# Advantages of a Statistical Estimation Approach for Clock Frequency Estimation of Heterogeneous and Irregular CGRAs

DENNIS LEANDER WOLF*, Computer Systems Group - TU Darmstadt
CHRISTOPH SPANG*, Embedded Systems and Applications Group - TU Darmstadt
DANIEL DIENER, Computer Systems Group - TU Darmstadt
CHRISTIAN HOCHBERGER, Computer Systems Group - TU Darmstadt

Estimating the maximum clock frequency of homogeneous Coarse Grained Reconfigurable Arrays/ Architectures (CGRAs) with an arbitrary number of Processing Elements (PE) is difficult. Clock frequency estimation of highly heterogeneous CGRAs takes additional factors into account, thus is even more difficult. Main challenges are the heterogeneous set of operators for each Processing Element (PE) and the irregular interconnect (connecting a CGRA's PEs). Multiple estimation approaches could be reasonable. We propose an optimized statistical estimator, which is based on our prior work. We demonstrate its superiority to state of the art neural networks in terms of accuracy and robustness, especially in situations with a sparse set of training data.

CCS Concepts: • **Hardware → Statistical timing analysis**; **Best practices for EDA**.

Additional Key Words and Phrases: Coarse Grained Reconfigurable Architecture, Heterogeneity, Design Space Exploration, Automation, Machine Learning

**ACM Reference Format:**
Dennis Leander Wolf, Christoph Spang, Daniel Diener, and Christian Hochberger. 2022. Advantages of a Statistical Estimation Approach for Clock Frequency Estimation of Heterogeneous and Irregular CGRAs. *J. ACM* 0, 0 (April 2022), 34 pages. https://doi.org/10.1145/3531062

## 1 INTRODUCTION

The increase in single thread computing power decreases every year [15]. In the beginning of the 2000s, the technological limitations [2] have urged engineers to focus more on improvements on the microarchitectural level. Besides the use of multiple processing cores, which boost the system performance on thread level, the use of hardware accelerators are a common choice to increase the overall performance of a computing system. Application-specific accelerators promise a huge performance gain, if the set of applications is known and will not change. However, this requires tight (and static) matching of the hardware accelerator to the individual application. As an

*Both authors contributed equally to this research.

Authors' addresses: Dennis Leander Wolf, wolf@rs.tu-darmstadt.de, Computer Systems Group - TU Darmstadt, Merckstraße 25, Darmstadt, Hessen, 64283; Christoph Spang, spang@esa.tu-darmstadt.de, Embedded Systems and Applications Group - TU Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289; Daniel Diener, Computer Systems Group - TU Darmstadt, daniel.diener@stud.tu-darmstadt.de; Christian Hochberger, Computer Systems Group - TU Darmstadt, Merckstraße 25, Darmstadt, Hessen, 64283, hochberger@rs.tu-darmstadt.de.


0004-5411/2022/4-ART $15.00
https://doi.org/10.1145/3531062

alternative, one can enhance the system with a reconfigurable accelerator, promising less but still significant speedups while offering more flexibility.

Field Programmable Gate Array (FPGAs) are a well-known form of reconfigurable accelerators. While being highly flexible and still allowing application-specific optimizations, they lack ease-of-use, if the provided FPGA vendor tools are used exclusively. They require experience in FPGA design. The realization of an application would need to be coded in a hardware description language making just-in-time synthesis infeasible. This would forbid an automated use of the FPGA. The consequence of these circumstances are tools for High Level Synthesis. Instead of easing their use, an alternative has emerged simultaneously. These are Coarse Grained Reconfigurable Architectures or Arrays (CGRAs). Due to a more coarse grained reconfiguration, they enable fully-automatic generation and mapping of new kernels within seconds. Therefore, they allow a significantly higher productivity. Instead of being based on lookup tables that are (re)configured on bit-level, CGRAs are based on Processing Elements (PEs), that are entirely configured on word-level. CGRAs are researched on three levels [22] [17]:
1. *System integration* (mainly software) deals with the incorporation of an accelerator. It is mandatory to prove that an accelerator can actually be used in a running system.
2. The process of *mapping an application* to a CGRA's PEs and scheduling its execution influences the time to the first accelerated run, but also the overall run-time performance gain.
3. A CGRA's *microarchitecture* defines the hardware of the accelerator. Together with the scheduling it impacts the performance and mapping of the accelerator.

### 1.1 Microarchitecture and Composition

We differentiate between the microarchitecture and the composition of a CGRA. The microarchitecture deals with functional and technology oriented decisions of a design. An example is the choice of the registerfile (RF) type (combinatorial or sequential read) or the type of the interconnect that is used - e.g. use of a network on chip or use of direct connections. In contrast, the composition of a CGRA can be understood as a specific parameterization of a CGRA instance. This mainly includes the number of PEs, their individual provision of operators and the topology of the interconnect. Consequently, a CGRA concept is based on one microarchitecture, but might allow arbitrary compositions. CGRAs are not necessarily regular in their structure. Instead, they can be completely heterogeneous with an irregular interconnect. In order to systematically research both fields (microarchitecture and composition), we have implemented a framework that allows modelling of arbitrary CGRA compositions. Among other features, our framework includes a scheduler, which respects the model of the CGRA. Also, both a legal schedule and a precise clock frequency estimation are required, for a reasonable wall-clock-time performance estimation.

### 1.2 Design Space Exploration

Although CGRAs enable dynamic mapping of new kernels, this is limited to the CGRA's capabilities. Heterogeneous CGRAs provide a higher number of optimization options than homogeneous CGRAs. Certainly, an optimal CGRA composition depends on the application that should be accelerated. A design space exploration (DSE) helps in finding reasonable CGRA compositions for a given set of applications, which most often will represent an application domain.

We want to realize a DSE that finds an optimal composition for a specific purpose in a fully automated fashion. For instance, this purpose can be maximizing the performance, minimizing the resource requirements or minimizing power consumption. Since CGRAs are used as hardware accelerators, we concentrate on maximizing performance. Nonetheless, the presented approach could be used for any other purpose as well. The metric of performance is usually the time to compute a given kernel or set of kernels. While the run-time in cycles can be derived from scheduling,

the more challenging part is the clock period. At this point we neglect effects of accesses to an external memory, since we are mainly interested in the effects of a composition onto the scheduling. Yet, we are aware of the challenges and potential issues of memory accesses [23] [14].

The clock frequency is the part that is more critical, since initial tests have shown that it varies by a factor of 2.5x depending on the composition. CGRAs with 2 or 3 PEs, with few supported operations and small memories (16 entries), easily exceed 150 MHz, while CGRAs with e.g. 30 PEs and more, wide support of operations and rather big memories (256 entries), only allow a frequency of about 60 MHz on a Xilinx Zynq-7000. Some examples of homogeneous compositions can be found in [30]. As a consequence, a methodology or heuristic is required to find an optimal composition in terms of application run-time. Simulated Annealing (SA) appears to be the most promising candidate. It promises high result quality and is easy to apply to our optimization problem. First tests resulted in solutions that were above average quality. The most reliable test compares the executions time of several compositions for the application ADPCM. Compositions found using SA performed about 2.0 to 2.5 times better than homogeneous CGRAs with 4, 9 and 16 PEs as well as the same homogeneous CGRAs with a manually reduced set of supported operations. In comparison to a genetic algorithm, SA performs 1.3 times better. While the genetic algorithm appeared to behave like a random walk, SA appeared to conduct a meaningful search as we analyzed the cool down velocity and the acceptance ratio. This strongly encouraged us to proceed with SA.

Especially SA, but many other heuristics as well, converge towards an optimal solution. Hence, they evaluate the run-time of vast numbers of potential solutions. In our case, SA might even evaluate millions of CGRAs. Therefore, it is infeasible to run a full synthesis each time. So we need an estimation of the clock frequency that is as accurate as possible in an acceptable run-time. The goal is to realize an estimator that estimates the theoretical maximum clock frequency that is limited in terms of synthesis. We do not consider further potential for overclocking that might be possible for specific applications when certain paths of the system might be neglected. Nui et. al. [21] present such an approach. However, we are interested in the composition in general and its impact on the clock period. Once SA has found a composition, such overclocking methods could be used in addition.

### 1.3 Contribution

We have generated a database of over 12k CGRA instances that are based on our CGRA microarchitecture, which was originally presented in [31]. These instances vary from 2 to 49 PEs for a Xilinx Zynq 7000 device. The database revealed that there are 43 different types of critical paths. A detailed analysis of relations between the composition of a CGRA and its critical path showed that there are only few and weak correlations. Hence, there is no basis for an analytical approach [33]. For solving challenges in classification and regression, machine learning including neural networks are a modern well-promising black box approach. Their main selling point is that a user is not required to deeply understand input to output relationships. Therefore, neural networks provide a promising approach to the given challenge of clock frequency estimation.

However, a simple thought revealed a methodology that is promising as well and easily comprehensible: *Similar compositions should lead to a similar critical path (length).* This is the basis for the statistical estimator presented in [33]. It consists of two phases. First, a group of similar CGRAs are chosen by an Euclidean distance function. Second, the final regression is carried out solely on the selected group by using a weighted kernel density estimator. In this contribution, we extend the options for both phases. We also consider the Mahalanobis distance function in phase one. In phase two, several variations of the regression are discussed - starting with the most similar CGRA over a simple mean value to a weighted kernel density estimation used in [33].

The additional value of this contribution is an in-depth discussion and evaluation of an optimized version of the statistical estimation [33] and a state of the art neural network that is implemented and optimized in TensorFlow [8]. We argue that the statistical estimator is as accurate as a modern neural network, while providing a better mean error in certain regions. Most importantly, we demonstrate that the statistical estimator has significantly better accuracy, when only a sparse reference set is given. This is a familiar situation for hardware engineers, since the generation of a data base is usually time intensive and costly.

The paper is structured as follows: The microarchitecture of the CGRA and the framework are presented in Section 2. It includes the main characteristics that influence the clock period of a CGRA and the generation process of the data base for the estimators. Section 3 deals with the coverage of the design space and the generation of the data base. Section 4 and 5 discuss the neural network and give detailed presentation of the statistical estimation. Section 5 also includes an evaluation of several distance functions and choices of parameterization like the group size for the regression. An in-depth evaluation of both approaches is given in Section 6. Related work is discussed in Section 7. Finally, Section 8 concludes with a comparison of both approaches.

## 2 MICROARCHITECTURE AND FRAMEWORK

This chapter introduces the microarchitecture. In order to allow a widely generic CGRA, we implemented a CGRA core and added individual wrappers that realize communication interfaces to the host system. The presented CGRA could be used to accelerate many different systems. We already integrated it into two systems [31, 32]. As part of this contribution, an integration into a control engineering environment is described (Section 2.1). The architecture is explained in a top down manner, starting with the system integration, followed by the explanation of the CGRA core. Eventually, the framework is explained that is used to model compositions and to generate HDL code. This framework also provides a mechanism to generate a reference set.

### 2.1 System Integration

As stated before, our CGRA can be integrated into two systems. One is an adaptive microinstruction driven processor. Adaptivity is realized by utilizing the CGRA autonomously on changing requirements of applications [32]. The second system is used in a project called UltraSynth [30, 31]. A combination of an ARM Cortex A9 with the CGRA is used in a control engineering environment. Both systems are implemented as an overlay on an FPGAs.

As described in the introduction, the overall goal is to find an optimal composition. Because the CGRA core needs to satisfy certain requirements for both integrations, the discussed host wrapper and the CGRA core cannot be fully decoupled. Therefore, we will focus on one of the systems. Since the DSE should be done for an entire application domain, the UltraSynth project [31] is a perfect environment, because it represents the domain of control algorithms.

The integration of the CGRA core is done for a Xilinx Zynq 7000 SoC. Figure 1 shows the setup of the UltraSynth project. A more detailed overview of the host wrapper is shown in Figure 2. Green modules are used for communication and brown modules have buffers, which intermediately store incoming or outgoing data. An Advanced eXtensible Interface Bus (AXI) interconnect is used to transfer configuration and run-time data from the host processor to the CGRA. Since the CGRA computes control engineering algorithms, accurate timing needs to be ensured. Therefore, a configurable hardware cycle counter (yellow) provides a heartbeat for the overall system in terms of periodic *sync_in* pulses. Two different buffers coordinate the data transmission to the host. The Result Buffer stores data that is crucial for the control. This data is sent to the On-Chip-Memory, which can be read by the host processor with low latency. The Log Buffer stores data that is sent to the DDR memory and then forwarded to the control terminal by the host. This data is typically

used for tracking, or constitutes the basis to observe a control algorithm. Because humans have a relatively slow reaction time, this data is not highly time-critical. Hence, the Result Buffer data is sent via a High Priority AXI-Master and the Log Buffer data is sent via a Low Priority AXI-Master. A state machine controls all modules. For a better arrangement it is not shown in Figure 2.

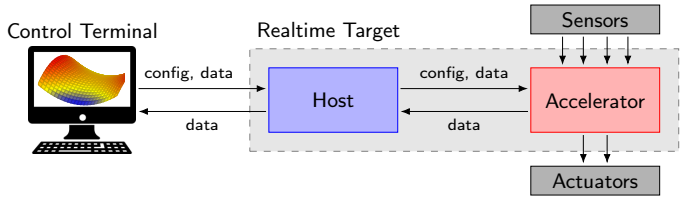



Fig. 1. Setup of the *UltraSynth* system. The CGRA is used as a hardware accelerator.

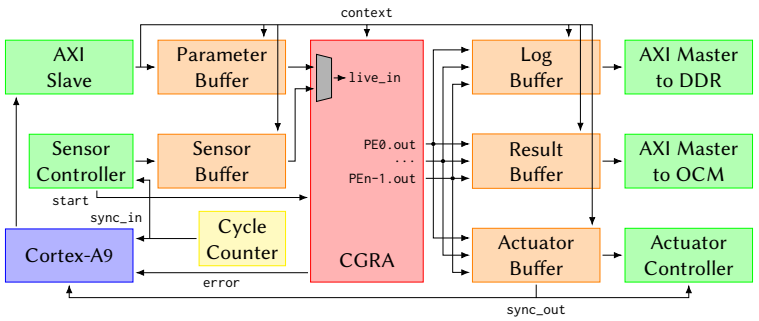


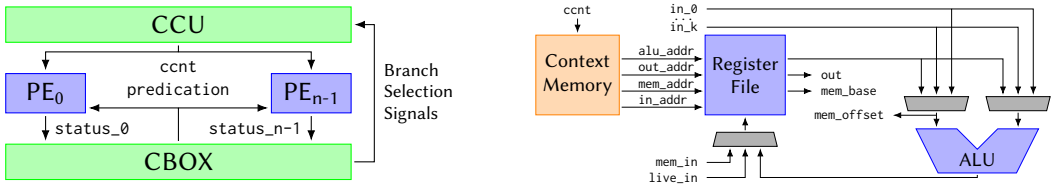

Fig. 2. Communication interface and control units in the wrapper.

## 2.2 Microarchitecture

Our primary goal is to design a CGRA towards three additional requirements besides high performance. First, allowing a wide range of applications that can be mapped onto the CGRA. Therefore, the CGRA needs to be capable of handling complex control flow. Second, enabling big applications and those which process or generate large amounts of data. This demands large data and reconfiguration memories. Thirdly, we want to allow just-in-time mapping of applications. Hence, the time required for scheduling and reconfiguration of the CGRA should not exceed a few seconds. This doesn't allow complex hardware structures and routing of data. The run-time of scheduling benefits from relatively simple structures and a modest interconnect.

*2.2.1 Overview.* Figure 3a shows an abstract view of the CGRA core. The heart of the CGRA core are the Processing Elements (PEs). We implemented a direct and latency free interconnect between PEs. Synchronization and the implementation of control is realized with two additional modules - the Condition Box (CBox) and the Context Control Unit (CCU). Relevant critical paths are identified and discussed in [33].

*2.2.2 Processing Elements.* The run-time of the application mapping procedure should be kept short. Thus, the structure of the PEs is straightforward. They consist of an Arithmetic Logic Unit (ALU), a Register File (RF), and a Context (Configuration) Memory, as shown in Figure 3b. The context control counter (*ccnt*) indexes the current configuration in each Context Memory. For




(a) CGRA Core - abstract overview.

(b) Structure of a PE.

Fig. 3. CGRA microarchitecture.

maximum flexibility, it is loaded each clock cycle. Besides multiplexer settings, e.g. ALU operand selections, a context defines the addresses for the read and write ports of the RF as well as the operation that is carried out by the ALU. Thus, one operation can be triggered per clock cycle per PE.

In the ALU each operator is realized in an individual module. This might hinder synergy effects of one fused ALU, but significantly eases the generation of a CGRA with heterogeneous PEs. Data can be passed from one PE to another without any delay. We have chosen a direct interconnect and a register file that is read combinatorially. This combination allows to instantly load data via $in_{0\,..\,k}$ in one PE that is required in another PE within the same clock cycle. Run time data from the host can be passed to a PE via the $live_{in}$ input.

*2.2.3 Handling Control Flow.* The strategy to handle applications with intensive control flow is speculation and predication. Considering an if-else construct, both branches are computed (speculation), but only the results of the valid branch are written to the eventual RF slot (predication). The CCU computes the *ccnt* in each cycle. If a conditional jump occurs, the next *ccnt* can be computed depending on the Branch Selection Signal (BSS). This signal will be of importance for the later timing discussion. The predication signals as well as the BSS are driven by the CBox. Both (predication and branch selection signals) are derived by linking incoming and intermediately stored status signals from the PEs. These status signals result from a compare operation. The structure of the CBox is shown in Figure 4.

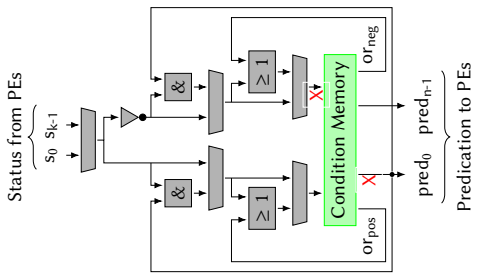



Fig. 4. Condition Box overview. Branch Selection source options marked with a *red X* can origin before or after the Condition Memory.

The complexity of the circuit is due to the required short-circuit evaluation, which is implemented in a Java framework [32] that is discussed in Section 2.3. Short circuit evaluation refers to a processing model of boolean expressions. It means that when boolean expressions are evaluated, the evaluation can be stopped once the first condition is found that ultimately determines the expression. As the CBox is required to evaluate these expression within one cycle, it likely becomes

part of most critical paths. The source of the BSS plays a major role for the achievable clock frequency. We differentiate between a *combinatorial* and a *sequential* BSS. Depending on the composition of a CGRA, the BSS can origin combinatorially from the input of the Condition memory or is sequentially and explicitly loaded via the memory's $or_{pos}$ output. The amount of predication outputs can be user-defined. In this contribution we limit the number of predication outputs to one, thus avoiding further expansion of the design space.

### 2.3 Framework for Composition Modelling

The main tool-flow is implemented in Java, as portability and reusability are considered more important than the tool run-time at the moment. It is divided into two interconnected toolchains for generating the CGRA composition bitstream (i.e. Steps 1 to 3, described in Section 5.1) and the application-specific CGRA configuration (i.e. Steps 4 to 9).

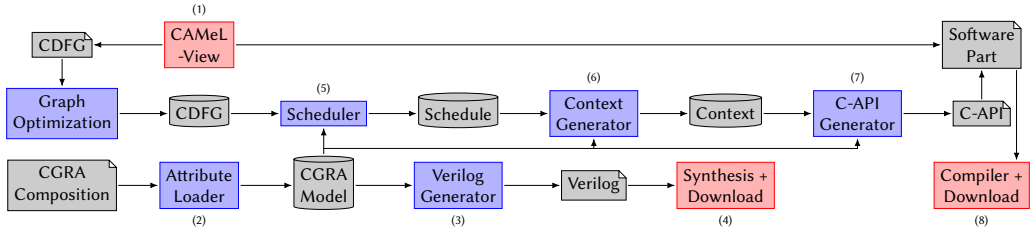



Fig. 5. Tool-flow of the framework for scheduling and hardware generation.

Figure 5 illustrates the main tool-flow of our framework. A model is used to represent the composition of a CGRA, which can be passed to a Verilog Generator (3) to generate HDL code. The HDL code can be used with any synthesis tool (4) that is capable of Verilog. Models can be stored on disk as a JSON description and loaded using a parser (2). This detailed model (including scheduling and clock frequency estimation) allows us to systematically research CGRA-structures and analyze the impact on scheduling quality without implementing the resulting hardware each time. The CGRA model itself is an attribute description of the CGRA, exploiting the object orientation of Java and its inheritance structures. [31] and [30] provide in-depth discussions of the tool and the system integration. Applications are implemented using an integrated design environment (IDE) for mechatronic systems. They can be mapped onto the CGRA using the CGRA tool chain (5-7). Depending on the realized CGRA, the related attribute description is loaded and scheduling (6), context generation (7) and the generation of a C-API (7) can be performed. The C-API builds a front end to configure the CGRA and to handle data transfers. All these steps (6-8) derive required information from the model. Eventually, the software including the C-API is compiled and downloaded onto the device (8).

## 3 DESIGN SPACE COVERAGE AND DATA BASE

This chapter gives an introduction to the design space we are dealing with and how it is systematically covered for reference data generation. Note that the common nomenclature of the underlying set to train a neural network is called training set. As the main focus of this work lies on the statistical estimator, the training set will also be referred as reference set.

### 3.1 Challenges of the Design Space

We limit the design space to the boundaries listed in Table 1. "Global" and "local" indicates whether the parameter can be set once (globally) or can be set for each individual element.

Table 1. List of main parameters of a composition

| Parameter | Global/Local | Supported Range | Selected boundaries |
|---|---|---|---|
| # PEs | - | 2 - ∞ | 2 - 49 |
| Interconnect | - | any feasible | any feasible |
| Set of Operators | local | 1 - 128 | 1 - 128 |
| Context size | global | 2 - ∞ | 64 - 4096 |
| RF size | local | 1 - ∞ | 1 - 256 |
| Local buffer size | local | 0 - ∞ | 0 - 256 |
| Global buffer size | global | 0 - ∞ | 32 - 256 |
| CBox BSS | - | 0 - 5 | 0 - 5 |

Despite the limitations, we are still confronted with a huge design space as shown in Eq. 1, even only considering the number of PEs, their individual provision of operators and their interconnect.

$$\begin{aligned} \mathbf{size}_{\mathbf{designspace}} &= 2^{\#PEs\times(\#PEs-1)} \times 2^{\#PEs\times\#Ops} \\ &= (2^{49\times(49-1)})2^{49\times128} = \mathbf{1.21\times10^{2596}} \end{aligned} \tag{1}$$

For the presented region of the design space, we have found up to 43 variations of critical paths. Most of them are permutations of the same path type. For a more detailed view, we recommend reading [33]. Due to the size of the design space, reference data has to be generated thoroughly.

### 3.2 Generation of the Reference Set

The implementation of the CGRA as an overlay architecture for FPGAs allows us in the first place to automatically generate a reference set with a sufficient amount of CGRAs. In theory, the proposed approach could be transferred to every other CGRA architecture, if it can be mapped onto an FPGA and if a tool is provided to systematically generate CGRA instances. A correctly sampled design space is crucial for this intention. This section discusses how the design space is sampled uniformly while avoiding static parameter sweeps that are too strict. There are three main demands to the generator: The generated reference set has to be representative, the generation process should be relatively fast, and the generation process should be immune to run-time errors.

Hence, we have implemented a tool that systematically covers the design space using a given grid, which segments the design space. Using a Random Composition Generator (RCompGen), the reference set generator iteratively and randomly generates compositions. The grid limits the number of accepted compositions per segment. The combination of the random generation and the static grid avoids problems such as periodic patterns and a biased coverage.

The generation process is fully automated by using a Vivado Runner that implements the desired logic on a given FPGA and returns the captured runs of Vivado including all relevant information. Moreover, it can search for the highest possible clock frequency. The generation process is also highly parallelized with multiple concurrent synthesis runs. Dying threads are supervised and can be automatically restarted. However, periodic supervision is required to eventually react on OS-related run-time problems and to evaluate logs. In the following, the segmentation is discussed. Afterwords, the tool-flow using the RCompGen and Vivado Runner are presented in detail.

### 3.3 Segmentation Grid - Covering the Design Space

As introduced in Section 3.2, the Random Composition Generator (RCompGen) first creates arbitrary compositions, which are then accepted or rejected by the grid. Both components, the RCompGen and the grid, can be parameterized to control the distribution and density. We configure the RCompGen to evenly distribute its randomness, and we configure grids with evenly sized partitions. By coarsening the grid, run-time is controllable, while still ensuring that all regions of the design space are covered uniformly.

The reference generator is designed to use multiple threads for its search. Once a limit of unsuccessful tries has been reached, the generation finishes. The limit as well as the number of threads can be defined by the user. We have usually chosen 16 threads for server class machines (2x Intel Xeon E5-2690 and AMD EPYC 7501). Based on empiric evaluation, we configure a thread to finish if 2000 random compositions in sequence are denied by the grid extension. The generation can be run multiple times while it reads-in all existing references at the beginning of each run, meaning it picks up where a previous run has left off. This mechanism proved to be highly valuable, since our compute-servers usually do not run several months without crashing, being updated, or being restarted.

Table 2. Grid Paramaterization for the Reference Set Generation (# of segments)

| Dimension | Range | Full | Sparse 1 | Sparse 2 | Sparse 3 | Sparse 4 |
|---|---|---|---|---|---|---|
| # Entries per Segment | 2 | 2 | 1 | 1 | 1 | 1 |
| # PEs | 2-50 | 49 | 49 | 49 | 49 | 49 |
| $^{\varnothing}$ # OPs per PE | 2-18 | 2 | 2 | 2 | 1 | 1 |
| Standard dist. # OPs per PE | 0-8 | 2 | 2 | 2 | 1 | 1 |
| Interconnect Density | Min-100 % | 3 | 2 | 2 | 2 | 2 |
| Standard dist. # Inp. per PE | Min-100 % | 3 | 2 | 2 | 2 | 1 |
| # LUTs | Min-Max | 22 | 22 | 11 | 3 | 2 |
| Total # possible Entries | - | 77,616 | 17,248 | 8,624 | 588 | 196 |
| # Feasible Entries | - | 11,176 | 2,746 | 1,723 | 261 | 174 |
| Percentage Feasible | - | 14.39 % | 15.92 % | 19.97 % | 44.39 % | 88.78 % |

The structure of the grid, which we used to segment the design space, is shown in column Full in Table 2. We have listed 5 sets. For the Full reference set, the chosen resolution leads to $2^3 \cdot 3^2 \cdot 49 \cdot 22 = 77,616$ segments (see column Full in Table 2), with the majority (85.61 %) of them being physically unfeasible combinations. Three examples: First, models with many PEs and many OPs often do not fit onto the FPGA. Second, CGRAs with 2 PEs only can have 100 % interconnect density. Third, certain combinations of # PEs, and # OPs per PE will always stay within a certain range of numbers of LUTs.

Using the random generator, accepting only two entries per segment, results in a reasonable coverage of 11.176 feasible entries. By coarsening the grid, thereby gradually reducing the number of segments, Sparse Sets 1 to 4 were reduced from the Full set. Together with the Full set, this gives five alternative differently-sized sets, which will later on be used to demonstrate each estimator's quality with fewer references being available. To avoid any direct dependence between the estimator's input and output, we generated another Test set from scratch. Instead of using the Full set's grid resolution, the Sparse 2 grid generation settings were used, and the stop-criteria was reduced by 25 % from 2000 to 1500 consecutively denied compositions. The resulting independent Test set contains 1,531 feasible entries out of 8624 possible entries at a percentage of 17.75 %.

### 3.4 Tool-flow

The framework provides a large number of tools that are used for automated synthesis and implementation. Figure 6 illustrates the simplified tool-flow of the automated synthesis for a given CGRA model (1). It is passed to the Vivado Runner (2), which uses a Verilog Generator (3) to create the Verilog Code for the model and generates a TCL-script that is run with Vivado 2019.2 (4) in TCL mode via a run-time object in Java. The Vivado Runner is a tool that we have implemented. A model can be passed to it and full synthesis and implementation are run including a heuristic search for the highest clock frequency. This search is iterative. It stops when a clock period is found for which the worst negative slack is around 0 ns +/- 0,1 ns without any other slack violations. Vivado itself is configured to use default synthesis and implementation settings. However, a user could select from a set of predefined settings in order to optimize the design towards a certain property. These settings are a collection of configurations that work well based on our experience. Synthesis and implementation feedback is tracked via the process as well. TCL printouts are processed and warnings and errors are analyzed. Should an error occur or a warning be suspicious, synthesis or implementation is aborted automatically. Fortunately, the Verilog generation has been heavily tested. No syntax or semantic errors have ever occurred during the generation of the reference set. Once synthesis and implementation have finished, the utilization and timing reports that were generated by Vivado are parsed and all required information is extracted using a Report File Reader (5). The critical paths can be extracted from the reports as they are. We assume that there are no false paths in our design that need to be excluded. This is due to the fact that our designs are synthesized without I/O bounds and without multi-cycle paths.

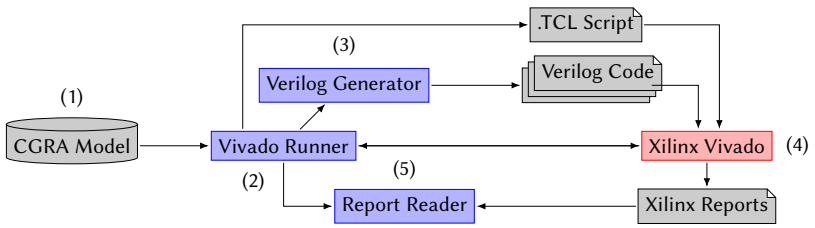



Fig. 6. Tool-flow of automated synthesis.

An automatized tool for generating CGRA models constitutes the basis for building a reference set. Ideally, this tool is bias free and can generate arbitrary instances throughout the whole design space. RCompGen was implemented for this purpose. It requires an initialization, which is an agreement on a fixed set of operators that can potentially be added to PEs. If none is given, the RCompGen autonomously searches the operator library and collects all feasible operators (currently 128 in total for the most widely used formats fixed and floating point - both in 32 and 64 bit).

The generation of a composition is computed based on three parameters: One is the number of PEs that should be contained in the CGRA and the other two are density probability distributions. Both dictate the spread and density of operators in PEs and the interconnect of PEs. Besides a uniform distribution, there are further functions to steer the RCompGen towards certain corner cases. For instance, using a Gaussian function for the operator density, *all* PEs will most likely have a medium-sized set of operators, while a high density function will most likely lead to heavily packed PEs. The same principle is used for the interconnect. This allows us to have a relatively low bias when creating a reference set while still being able to fill corners of the design space without having to hope for sheer coincidence.

## 4 NEURONAL NETWORKS ESTIMATION APPROACH AND THEIR LIMITATIONS

Statistical estimation can be done in numerous ways. One of them are Artificial Neural Networks (ANN) [19], which gained high popularity in the last decade and led to a nowadays commonly used method. As described before, estimating the clock frequency is a complex task, especially with regard to the size of the design space and the amount of possible critical paths. ANNs provide a promising approach to this challenge. This chapter explains all different methods we employed to create an ANN, which is competitive to our statistical approach. It is intended to be a meaningful baseline comparison to later-on validate the strengths (and weaknesses) of our own approach in Section 6.

Firstly, we describe the problems occurring in neural networks with respect to multiple dimensions of the design space. This limits the possible representation of CGRAs on an ANN. We discuss the initially straight forward representation of a CGRA and a manually reduced representation. It is basically a compressed representation, which also discards certain features but yields far better results as it reduces the input complexity of the network. At this point, it is important to mention that the statistical estimator uses a third representation of a CGRA model. It has been used for the ANN as well but yields worse results in contrast to the ANN specific reduced representation. In addition, autoencoding, that is commonly used to overcome such issues, is evaluated.

After settling down on a representation, we explain the chosen ANN topology and discuss other possibilities. Lastly, the training is presented. The evaluation of the ANN can be found in Section 6.

### 4.1 Curse of Multidimensionality - Picking the right Representation

As mentioned in Section 3, our design space is huge and contains multiple dimensions. Under these conditions difficulties arise. One is the distribution of data: with increasing number of dimensions, the spread of the possible samples strongly increases. This has to be kept in mind to create a sample set suitable for training. ANNs *should* estimate well on data never seen before, therefore it is important to put data with relevant information about the task into the reference set. ANNs need to see patterns and learn autonomously how to estimate. If every element of the reference set contains the same information or pattern, an ANN cannot learn important relations of the design space and thus does not perform well on new data. Solving this multiple dimension problem was done by using our sampling method described in Section 3.

*4.1.1 Representation with all information.* Another problem is the increasing complexity of the function to approximate. From the discussion of last section, it is clear that the representation of the CGRA on the ANN is a critical step. The goal is to use as few features as possible but enough to provide the information needed to estimate the clock frequency properly in terms of small error margins. Choosing the features is done by considering the important values of a CGRA regarding the clock frequency. The initially taken representation trivially includes all information (Figure 7).

As shown, the first features match the CGRA-global characteristics. Following those, every PE is listed with its values, the sizes of memory, if it is connected to another PE, and if the operation is available. The latter two are one-hot-encoded. Advantages of this representation are the abundance of information, especially in terms of interconnect and availability of operations. At the same time, this wealth of information is a disadvantage, as a lot of features are needed. We want to estimate up to 49 PEs with 128 available operations. Thus, looking only at the PEs, $49 * (128 + 49) = 8673$ features are needed to represent their interconnect and operations. Another drawback is the size-dependency of the CGRAs. In our case we estimated CGRAs with a size up to 49 PEs. In general, even larger CGRAs are possible and should also be estimable. With this representation, we would need to create a new network capable of estimating larger CGRAs, create new data and train it

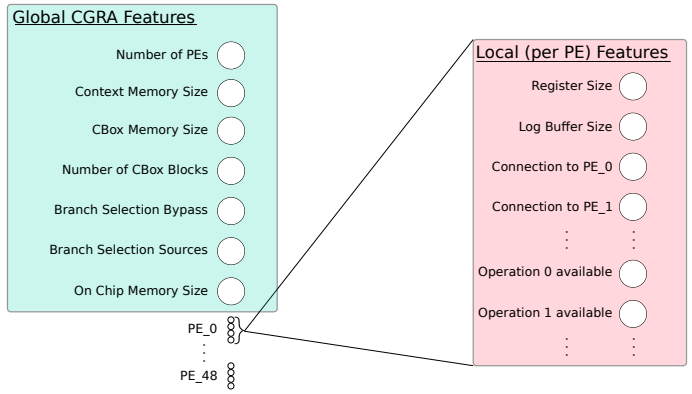



Fig. 7. First approach of feature representation of the ANN.

from scratch. Desirable are networks independent of the number of PEs, so even larger CGRAs can be estimated without training new networks.

As a result, training an ANN with this representation and the given 12k training data did not deliver reasonable accuracy because of the sheer amount of features and the lack of data. Another representation had to be taken, which is discussed in the next section. Still, this representation can be useful, since all important information is encoded. In Section 4.1.3, this representation is used to train an autoencoder and automatically decrease the amount of dimensions to achieve better estimations.

*4.1.2 Manually reduced representation.* One way to tackle the disadvantages of the representation discussed in the last section is to conduct a manual reduction of features. Doing so, information needs to be summarized with the hope of achieving a better estimation because of fewer dimensions, even though some lost information might have been helpful. The resulting representation is shown in Figure 8.

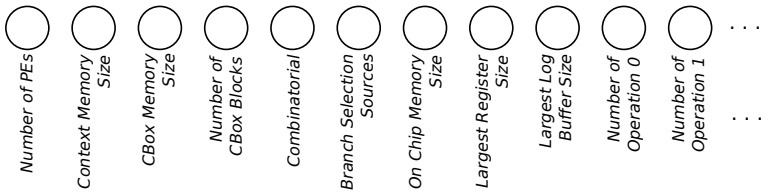



Fig. 8. Manually reduced approach of feature representation of the ANN.

The first features are still the global characteristics of the CGRA. Instead of listing all PEs, we only pass the largest register and buffer size to the ANN. These values have given us the best results in the evaluations. After that, all available operations are listed, denoted with the overall number of an operation. As can be seen, the information of the interconnect is completely discarded. At a first glance this might be contra-intuitive as the interconnect is an important piece of information of the critical path and therefore of the delay. The thought is that the information of the operations is enough for the ANN to get an idea of the critical path. For example, if it is known that a logical operation is implemented, the ANN might know that the critical path involves the CBox; if a complex arithmetic function is implemented, the ANN might locate the critical path in that operation. Even

though the interconnect is lacking, the ANN (and even a human) can have an idea of the critical path. Especially for small CGRAs this works well, but for larger CGRAs the critical path is not dominated by an ALU operation anymore. [33] discusses this in more detail.

Contrary to this drawback there are two advantages: the network is now independent of the size of the CGRA and only 133 features are used instead of over 8000. In theory, we could decrease this number even more by calculating aggregated values such as a percentage representing the interconnect density or using the amount of only complex operations. Such a representation is used for the statistical estimator, as explained in Section 5.5. We trained an ANN using these features as well. The results were around 10% relative mean and 30% - 40% maximum error, which are worse than the results later seen in the evaluation. We evaluated various kinds of other representations as well, but in the end all resulted in loss of accuracy. Therefore, we decided on the shown feature representation and tweaked the ANN accordingly.

*4.1.3 Autoencoding.* Another way to reduce the amount of features of the representation in Section 4.1.1 is the usage of autoencoders [1]. These are neural networks, which can automatically learn a more advantageous representation of the features using considerable less neurons. This is done by providing all the features (in our case over 8000) at the input but also at the output. Figure 9 depicts the basic topology of such a network. The hidden layer in the middle of the network contains less neurons than the input. Neurons from the input to the middle layer are called encoder, from the middle layer to the output are called decoder. The output of the encoder can be used as input of a different neural network. To train this network, the weights of the already trained encoder need to be frozen. Data to train an autoencoder can easily be obtained in our case, since the inputs are the same as the outputs. One does not need to know the wanted value to estimate while training the autoencoder. Creating labeled data requires us to synthesize the model many times, which is the most time consuming step in the estimation pipeline.

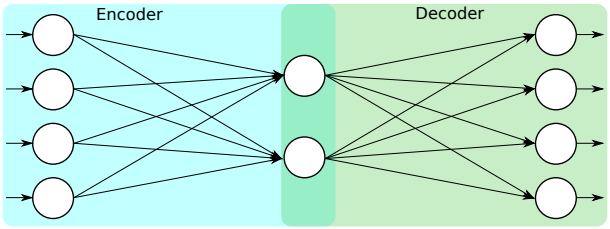
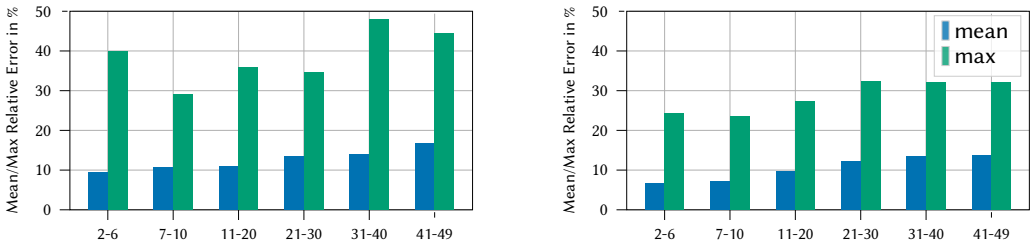



Fig. 9. General topology of an autoencoder.

For this purpose, we can create millions of unlabeled data in our framework. Training an autoencoder with this amount of data in a timely manner requires fast hardware. Our infrastructure is infeasible to do this, thus we evaluated an autoencoder trained on 150k data and one trained on 500k data. Both encoders were used as an input of a Multi Layer Perceptron (MLP). Results are depicted in Figure 10. They are better than an ANN using the naive encoding directly but worse than the ANN specific representation. Providing more unlabeled data results in better results, despite not adding a single new labeled data. Still, 500k data are not enough to encode the design space and dimensions by an autoencoder. If we increase the amount of data, training on an AMD EPYC will take too long, as we would need millions of instances. Because of the missing computing capacities we decided to use the manually reduced representation in the evaluation. With available computing power, autoencoders are a promising technique to solve a complex task when unlabeled data can be created much faster than labeled data.




(a) Autoencoder trained with 150k data.

(b) Autoencoder trained with 500k data.

Fig. 10. Mean/max error over PE amount of different autoencoders.

### 4.2 Topology

The previous section discussed the representation of the inputs on the ANN. We used a MLP for the ANN. This is one of the basic possible ANNs characterized by its fully connected layers, e.g. every neuron of layer $n-1$ is connected with every neuron of layer $n$. Even though a single neuron only calculates a weighted sum and compares if a bias is passed, the large amount of neurons still allows approximation of complex functions. We used one hidden layer (discussed in the next section) with 250 neurons. Inputs are given by the manually reduced representation with 133 neurons. One output neuron is used because we want to conduct a single regression. This results in 33,751 weights (excluding bias) to adapt. Figure 11 shows the whole network. This is also the topology used in the evaluation.

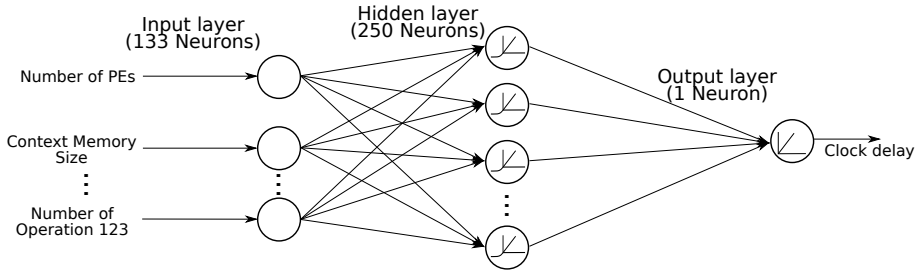



Fig. 11. MLP topology of the ANN used in the evaluation.

Currently, other ANN topologies like Convolutional Neural Networks (CNN) [16] exist. Primarily used in image classification, it can be adapted to different problem domains. A CNN could be used to estimate clock frequency by using graph CNNs. These networks operate directly on graph data to which a CGRA can be transformed. One advantage would be the autonomous feature reduction and the incorporation of the interconnect because of the graph structure. These advantages could lead to a high gain of accuracy. However, the given heterogeneity and irregularity would lead to new challenges in representation, since each composition would lead to a slightly different graph. At the time of training, no suitable graph neural networks library was available. Future development might be able to handle irregular graphs that vary immensely from another. Open-source libraries like Spektral [9] with TensorFlow integration already exist. They represent an interesting trend, but are not yet suitable for our problems.

### 4.3 Workflow and Training

The implemented workflow with ANNs is presented in Figure 12. First, all reference CGRAs are parsed into the framework. After that, data is prepared for TensorFlow by writing it into CSV files according to the representation. These file are read into Python with the Pandas library. Before creating the ANN, some preprocessing has to be done. It includes the splitting of the features and labels but also the scaling of the given data. By dividing all features with the highest existing value, all features were normalized on the range of (0, 1]. We also evaluated a scaling of inputs to a mean of 0 and a standard deviation of 1. The first scaling method achieved better accuracy. The prepared data is consequently used for training, which is done through the Keras Wrapper [4]. We can use the model either directly in TensorFlow or in Java. Latter one is achieved through the Deeplearning4j library [29] with its Keras import function.

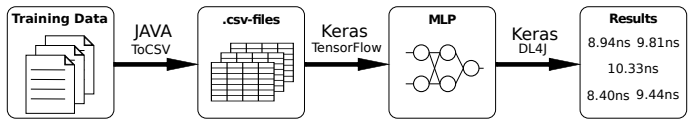
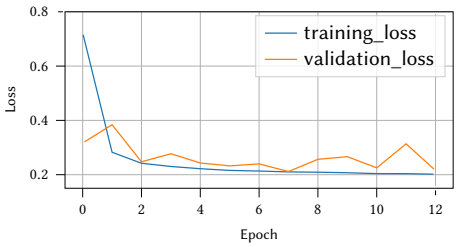


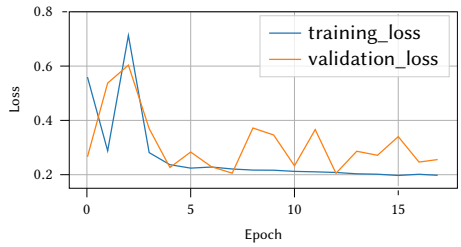

Fig. 12. Workflow of ANN estimation.

Training of the MLP was done by the Nadam-Algorithm [7] with a learning rate of 0.01. Nadam is an adaptive algorithm with momentum, which achieves good convergence on the reference set. Algorithms like Stochastic Gradient Descent (with Nesterov), Adam and RMSProp were also evaluated. All achieved almost the same accuracy, but Nadam had the shortest training-time. For a more detailed overview of these algorithms, [24] can be consulted. For the activation functions we use for all layers but the last Exponential Linear Units (ELUs). According to [5], ELUs alleviate the vanishing gradient problem and can improve learning characteristics and training time. The last layer uses the identity function to be able to map the regression value. Using these activation functions determines the first initialization of the weights, which in this case is the He-Initialization [11]. For other parameters like the batch size a value of 16 was taken. This did not have much influence on the end result, but numbers higher than 32 have resulted in worse results. To prevent overfitting, two approaches are used: one is to create a large MLP (around six hidden layers) and apply regularization (Dropout, l2-norm), the other is to manually change the size of the MLP. Both used an early stopping mechanism to prevent overfitting without slowing down the convergence or training time peculiarly. [10] favors the first method, since the network will certainly be large enough to cope with the data. Drawbacks are longer training and estimation times, whereby the second approach has to be done manually. In our case both techniques worked well and achieved the same accuracy. In Figure 13, the loss function of a small and a large MLP are plotted against the epochs. Both converge around the same value and since early stop is used, the validation loss is minimal in the end. We use the small MLP in our evaluation especially for the sake of faster estimation times.

One remarkable setting is the usage of the error function. We evaluated two, the mean squared error and the huber loss [12]. Even though the first is more common, the second achieves two to three percent better results on the maximum error and half a percent improvement on the mean error. Compared to the squared mean, the huber loss function is less sensitive to outliers, which seems to have a positive influence on the sparse reference set. Table 3 summarizes all characteristics of the small and large MLP.




(a) Small MLP with one layer.




(b) Large MLP with six layers.

Fig. 13. Loss functions (Huber-Loss) of a small and a large MLP.

Table 3. Characteristics of the small and large MLP.

| | Nr. of layers | Neurons per layer | Activation function | Loss function | Weight init. | Overfitting preventions |
|---|---|---|---|---|---|---|
| Small MLP | 1 | 250 | ELU | Huber loss | He init | Early stop |
| Large MLP | 6 | 500 | ELU | Huber loss | He init | Early stop, Dropout, L2-Norm |

Training was done in Google Colab on a NVIDIA V80 with all data in memory. All networks ran 5-15 epochs (early stopping). One epoch equals one iteration over the whole reference set. Large networks with regularization took 7-10 minutes to train, small networks took 1-3 minutes or less.

## 5 STATISTICAL ESTIMATION

This section discusses the statistical estimation approach. Before elaborating on the actual chosen estimation algorithm, alternative approaches, which we have explored, are discussed first. This builds a better basis to understand the challenges of the given task. Next, the selection phase is presented. The user can choose between two different distance functions - Euclidean distance and Mahalanobis distance [18]. Then, the actual regression is discussed. The most important parameters of the estimator and their impact on estimation quality are analyzed.

We decide on using the *Sparse 1* reference set as it provides a good trade-off between the effort for generating the reference database and the resulting estimation accuracy. Compared to the Full set, its coarser grid suits well to demonstrate the estimator's characteristics.

### 5.1 Alternative Approaches and LUT Estimation

Prior to the realization of the statistical estimator, we manually analyzed the presented reference database for noticeable structures. The goal was to find sufficient correlation of the CGRA's parameters to its critical path, especially it's critical path's length, such that an analytical estimator could be implemented. The attempt was to develop a formula expressing the strongest correlations comparable to our trivial LUT-estimator, which adds-up a reference LUT-amount per CGRA-feature (Equation 2) and uses a linear correction factor depending on the CGRA's size (Equation 3). The LUT-estimator reaches a maximum relative error of below 10 % and is later on used.

$$LUTsum = \left( \sum_{0}^{\#Features} Value_{feature} \cdot Weight_{feature} \right) \tag{2}$$

$$\#LUT_{est} = LUTsum \cdot CorrFactor(LUTsum) \tag{3}$$

We implemented a tool, which dynamically creates spreadsheets and visualizations. It provides insights to this multidimensional problem. The tool separates the included CGRA-features and thereby locates each CGRA in multidimensional space. With manual analysis and different points of view, we indeed found relations. Unfortunately, most findings are rather local to corners of the design space and do not cover the vast majority of more centered data points [33]. We identified single prominent dimensions in the corners of the design space like a complex arithmetic operation on a small CGRA's critical path. But when analyzing central parts of the design space, we could only do vague assumptions on the critical path. We assume the imprecision/noise of the correlations to be caused by two effects:

First, the underlying place and route algorithm's reaction on a small change is hard to predict and depends on the former level of stress. Here, the term "stress" correlates to a high (local) FPGA resource utilization, which leads to local overutilization and congestion. The FPGA's main resources are logic, memories, flip-flops, and wires. In case of high stress, the place and route algorithm must take additional measures to avoid resource conflicts e.g. by using longer wires to reroute a signal around a local congestion. These measures (e.g. longer wires) can delay the affected signals, thereby possibly impacting the design's overall critical path length. The actual amount of additional stress induced by changing a single dimension can potentially depend on all other dimensions. For a detailed estimation of the algorithm's behaviour, the level of (local) stress is a relevant information that is not available without synthesis. The lack of this knowledge leads to noise-alike imprecision.

Second, our tool considers a fixed and relatively large set of dimensions. Its 19 elements contain the number of PEs, the interconnect density, and further CGRA parameters. The complete set will be discussed later in Section 5.5 and is included in Table 4. Even with high effort, humans are limited in analyzing high-dimensional problems. For the exploration of dominant meaningful features, we iteratively fixated some dimensions and analyzed the remaining dimension's behaviour. This idea is comparable to feature selection in machine learning. By limiting the amount of dimensions, the excluded dimension's impact does not vanish, but becomes harder to understand and eventually behaves like additional noise. Algorithms are not limited by the amount of dimensions, but typically face related problems.

By using a Java package [6], we realized a Multiple Linear Regression (MLR) estimator. Whereas a Linear Regression algorithm models an independent input variable onto a dependent output variable, MLR does the same for higher numbers of independent input variables. The result is a formula, which can be used for estimation. Unfortunately, when estimating a CGRA's critical path, MLR resulted in unreasonable relative estimation error peaks with more than 100 %. We assume this to be caused by MLR's key requirements (linearity and low noise) being violated by our estimation problem, thereby reducing robustness. Higher order polynomials require more parameters to be defined and are expected to further decrease robustness against noise.

### 5.2 Concept

Our analysis showed that although two CGRAs, which are similar in multiple characteristics, can have a different critical path route, their delay is often still comparable. Based on the idea that similar CGRAs should lead to a similar length of the critical path, we constructed a two-stage estimator. First, the estimator uses clustering to find a small group of closely related CGRAs. Second, regression determines the final estimation.

Algorithm 1 first coarsely reduces the number of references based on the CGRA's type of BSS (reminder: BSS stands for Branch Selection Signal). The reference CGRAs within the selection

**Algorithm 1:** Statistical estimation algorithm

**Data:** Reference CGRAs, CGRA to estimate, Boolean: Euclidean of Mahalanobis distance, Selection: Type of the regression algorithm
**Result:** Estimation of the critical path's clock period
Load CGRA to estimate;
**if** CGRA has combinatorial BSS **then**
  Load reference CGRAs in Selection Window PE-Range with combinatorial BSS;
**else**
  Load reference CGRAs in Selection Window PE-Range with sequential BSS;
**for** all loaded CGRAs and the CGRA to be estimated, **for each** relevant parameter **do**
  Compute normalized dimension value;
Based on distance: Select group of N closest references;
Perform regression estimation algorithm and output estimated delay;

PE-window range are loaded and placed within a normalized multidimensional space. This allows to compute relative distances between each reference and the CGRA, which is to be estimated. These distances are dual-purpose. Firstly, the distance-information helps finding a small group of highly relevant CGRAs. Secondly, a regression algorithm uses the distance to weight each reference via predefined metrics, to finally estimate the critical path delay. With growing complexity, the estimator can either simply pick the delay of the closest CGRAs, or compute a Mean, Weighted Wean, or Weighted Kernel Density Estimator (KDE) regression between the closest candidates.

### 5.3 Distance Functions - Euclidean and Mahalanobis Distance

We first insert a general discussion concerning two distance functions, as we will later on evaluate both. Typically, an Euclidean distance function is used to compute the distance between two points in n-dimensional space. Our empiric testing showed competitive estimation accuracy for the Mahalanobis distance function [18]. Mahalanobis distance typically is used to measure the distance between a point $\vec{x}$ and a distribution $D$. By rescaling all axes, the scale-invariant and unit-less function becomes a statistical distance function between point $\vec{x}$ and the mean $\vec{\mu}$ of $D$ (Equation 4).

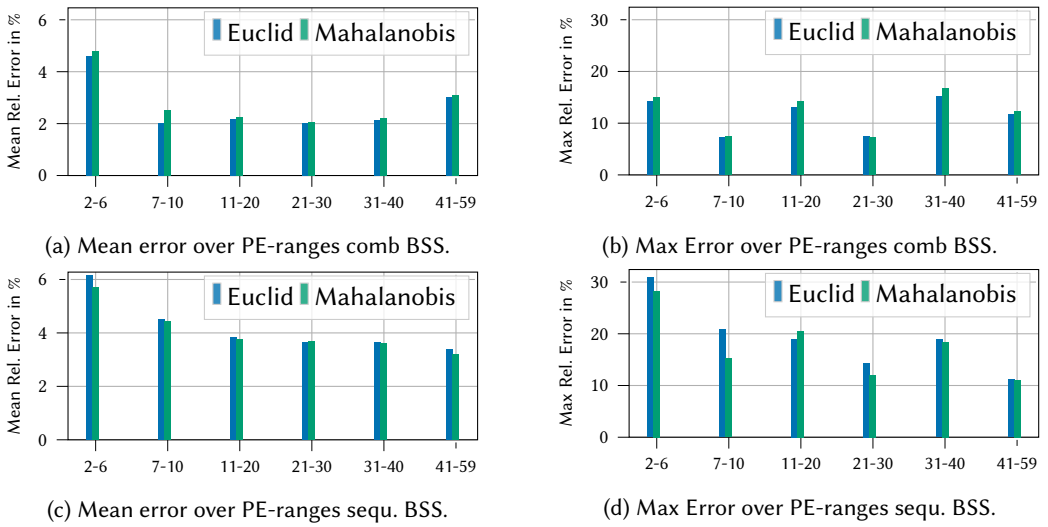
$$D_M(\vec{x}) = \sqrt{(\vec{x} - \vec{\mu})^T S^{-1} (\vec{x} - \vec{\mu})} \tag{4}$$

The value D can be interpreted as the likeliness of point $\vec{x}$ belonging to the set $D$, with $S$ being the covariance matrix. By replacing $\vec{\mu}$ with $\vec{y}$, the Mahalanobis rescaled coordinate system can also be used to compute a statistical distance between two points $\vec{x}$ and $\vec{y}$ (Equation 5).

$$d(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T S^{-1} (\vec{x} - \vec{y})} \tag{5}$$

Two points with constant Euclidean distance will have a small Mahalanobis distance, when they are likely to be part of distribution D. With growing dissimilarity to D, the Mahalanobis distance between the two points grows. Whereas the Euclidean distance requires all data to be in the same units of length, which we realize by normalizing all dimensions, the Mahalanobis distance is immune to differently-scaled dimensions and independent units of length. For the Mahalanobis distance normalizing and rescaling of dimensions has no effect. We verified this both mathematically and practically for testing parts of our implementation.

The resulting estimation quality by the chosen distance metric depends on the structure of the estimation problem and the distribution and correlation of reference points. If, for each CGRA to be estimated, a homogeneous group of similar CGRA references can be found, Euclidean distance performs better in terms of Mean (Figure 14a) and Maximum error (Figure 14b). If the group members lack in similarity, Mahalanobis distance is a better choice, as it further reduces the impact of multivariate outliers on the estimation (Figures 14c and 14d).




(a) Mean error over PE-ranges comb BSS.

(b) Max Error over PE-ranges comb BSS.

(c) Mean error over PE-ranges sequ. BSS.

(d) Max Error over PE-ranges sequ. BSS.

Fig. 14. Comparison of estimation accuracy, when using Euclidean or Mahalanobis distance functions over different design space areas. We use the *Sparse 1* set. Other parameters are set to the later discussed optimum. Euclidean distance performs slightly better for CGRAs with combinatorial BSS. Mahalanobis distance improves the global worst relative error and is typically more accurate for CGRAs with sequential BSS.

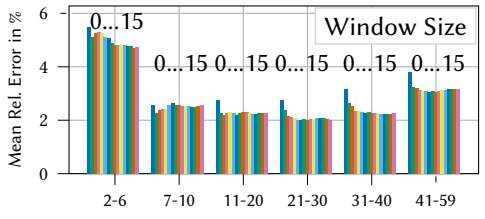
### 5.4 Pre-Selection Phase - Selection Window over PE Amount

As we will demonstrate in Section 6.3, the ANN generally profits from an increasing amount of reference data. It autonomously interprets the data to find patterns helping with the estimation.

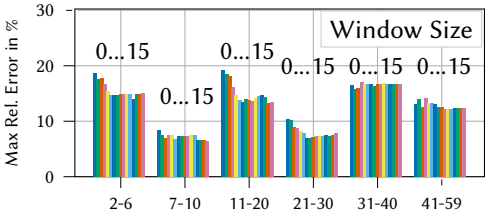
In contrast, the statistical estimator is based on the only assumption that similar CGRAs contain a related critical path. Therefore, an early exclusion of unrelated CGRAs, which likely cannot be used for estimation, does not only come at no significant loss of accuracy and potentially reduces the required reference set's size, but also speeds-up the estimation process.

As a consequence, in the algorithm's (Algorithm 1) first step, the pre-selection phase, the size of the reference set is reduced. This is done by excluding CGRAs, which have a very different size and structure when being compared to the desired CGRA. For our estimation approach, such instances are deemed useless and likely would be excluded from the subsequent estimation process, anyway. One parameter is the type of the CGRA's Branch Selection Signal (BSS), which can be combinatorial and sequential. All references with a different type of BSS (combinatorial/ sequential) and a strongly deviating PE-amount are excluded. In contrast to our previous work [33], which implemented windowing for static ranges, a Selection Window (SW) over the CGRA's PE-Amount returns only CGRAs with a PE-Amount in-range. For example, given a CGRA with 15 PEs a SW of ±2 would return all CGRAs with 13 to 17 PEs.

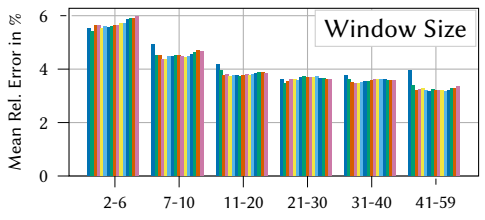
Choosing the Selection Window size effects the output quality. In the following example the SW is varied from ±0 to ±15 PEs per evaluated PE-range from left to right in Figure 15. If the SW is chosen too small, error spikes within estimation accuracy can be observed. A large SW increases run-time, as more references are taken into consideration by the subsequent estimation steps. Widening the Selection Window leads to smoothing of the estimator's output, thus can be used to limit the mean, and often also the maximum error. However, over-smoothing can result in error spikes and should be avoided. Based on Figure 15 and by not focusing on noisy artifacts, we decided to fix the windowing size to ±10 PEs. This is often, but not always, a sweet spot for low mean and maximum estimation errors.



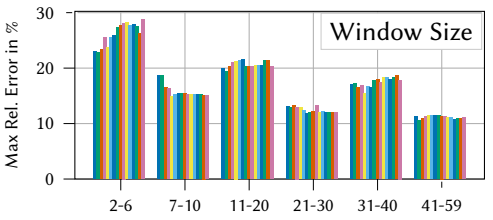

(a) Mean error over PE-ranges comb BSS.




(b) Maximum error over PE-ranges comb BSS.




(c) Mean error over PE-ranges sequ. BSS.




(d) Maximum error over PE-ranges sequ. BSS.

Fig. 15. Variation of the Selection Window (SW) size from ±0 to ±15 PEs. We use the Mahalanobis distance function, and the *Sparse 1* reference set. Group Size (GS) (explained in Section 5.6) is set to 10. Also, noise-alike artifacts can be observed. They likely relate to local congestion (stress) of FPGA resources (Section 5.1).

### 5.5 Statistical Estimator Input Properties and Distance Computation

After the optional pre-selection phase, which uses the BSS type and PE-amount to perform a rather quick and coarse reduction of references, our estimation algorithm now focuses on a broader range of CGRA properties. It computes a distance value evaluating each reference CGRA's similarity to the CGRA, which is to be estimated. This distance is later on used to further reduce the reference set and also is an input to the estimation's final computation — the regression.

Similar to the ANN, the statistical approach needs a carefully chosen input set to compute a meaningful distance value. There are multiple arguments against the direct use of CGRA properties: An unreasonably high number of inputs causes noise and compatibility to variable-sized CGRAs would be complex as the amount of inputs would vary. To overcome these issues, additional algorithms for input reduction and symmetry detection would be required. Section 7.2 refers to some possible approaches. In the following, we introduce the term CGRA "parameter" to describe such derivations or aggregations of one or more CGRA properties.

Picking the right CGRA parameters can be non-trivial and automatically deriving the most important parameters for (other) estimation problems would be recommended. As part of our early

future work, we implemented Simulated Annealing to automatically perform an early selection of relevant CGRA parameters. By iteratively mutating weights, this algorithm adapted the influence per CGRA parameter. Optionally, this approach could be used to tune the estimation quality for local areas within the design space. Unfortunately, our automatically derived "perfect" smaller property subsets were prone to overfitting, thereby affecting estimation accuracy for new CGRAs.

Instead, we have manually chosen a larger set of important parameters based on the insights of the previously conducted analysis of the reference set. All considered parameters are listed in Table 4. They are subsets or aggregations of subsets of the total properties discussed previously in Sections 2 and 3. Some of these parameters explicitly include the CGRA's size, which is the case for PE-Amount and LUT-Amount. Most others express a density of logic or structure — their normalized values are independent from the CGRA's size. As an example, 100 % interconnect relates to a CGRA with all PEs fully connected — independent from its total amount of PEs. For equalizing the error on both reference and desired CGRAs, the amount of LUTs are always estimated using the previously described LUT estimator (Equation 2 and 3).

Using a different dimension per property type, these parameters are then repurposed as coordinates to locate each CGRA within a multi-dimensional space. Coordinates are computed for the reference CGRAs and the CGRA to be estimated. Finally, the desired CGRA's distance to the given reference CGRAs is computed. Mahalanobis or Euclidean distance can be either used, with the latter requiring a prior normalization ensuring all dimensions fitting in an equal range.

In order to adapt our statistical estimator onto other types of CGRAs or other estimation problems, this set of parameters (Table 4) is to be adapted. Technically, this can be performed by editing the property-individual dimension value computations. Estimation quality will also depend on how well these parameters can assess major impacts on a different CGRA's critical path length.

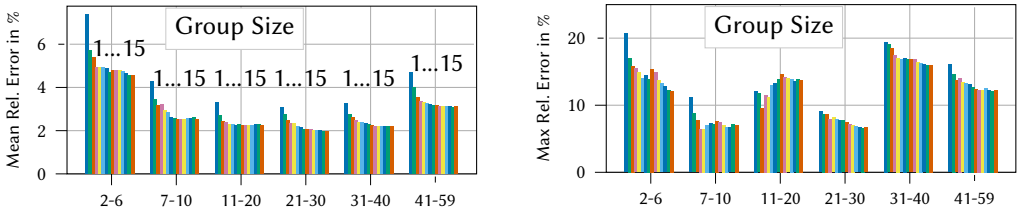
Table 4. The estimator's complete set of indirect CGRA description parameters

| | | |
|---|---|---|
| - # PEs | - Size of biggest RF | - # PE Interconnections |
| - Interconnect Density | - Sum of RF Sizes | - Width of widest Context Mem. |
| - Context Memory Size | - Set of PE-Operators | - Standard Deviation of RF sizes |
| - Global Result Buffer Size | - Max. Log Buffer Size | - # PEs with Control Flow OPs |
| - # Lookup Tables (LUTs) | - Local Buffer Size | - Standard deviation # LUTs per PE |
| - Max. # of PE Inputs | - Sum of Context Widths | - # LUTs of Largest PE |
| - Stand. Dev. Context Widths | | |

### 5.6 Selection of Nearest Neighbours

The preceding two steps assure that we now have a meaningful distance value (Section 5.5) indicating the similarity of the CGRA to be estimated and every remaining CGRA of the pre-selected reference set (Section 5.4). After sorting the reference CGRAs by their distance, only the `N` most related references are selected to be part of the group that is used in the later regression. Again, the manually-chosen size of this group affects the estimation's quality. A sweep of the group sizes is illustrated in Figure 16. There are several expectable effects on the estimation accuracy. Too few neighbouring CGRAs often result in higher error peaks. In contrast, including too many distant references can lead to an over-smoothed result, thereby potentially increasing the maximum relative error (Figure 16b 7 - 20 PEs). We examined different approaches towards a dynamic Group Size (e.g. distance limitation, limit first derivation of the distance), but finally decided for a fixed Group Size of 10 references. One reason for this decision is the regression step later on. It is immune to a

limited number of more distant references. In case of high distance, a reference's impact on the overall estimation result is reduced.




(a) Variation of Group Size (GS).

(b) Variation of Group Size (GS).

Fig. 16. Variation of the Group Size from 1 to 15 references: Error peaks in case of too few references. Too many references can lead to over-smoothing. Again, noise-alike artifacts can be observed. They likely relate to local congestion (stress) of FPGA resources (Section 5.1).

### 5.7 Regression

Now that we have a small set of the most similar CGRAs, the final step is the regression estimation. Different approaches are discussed.

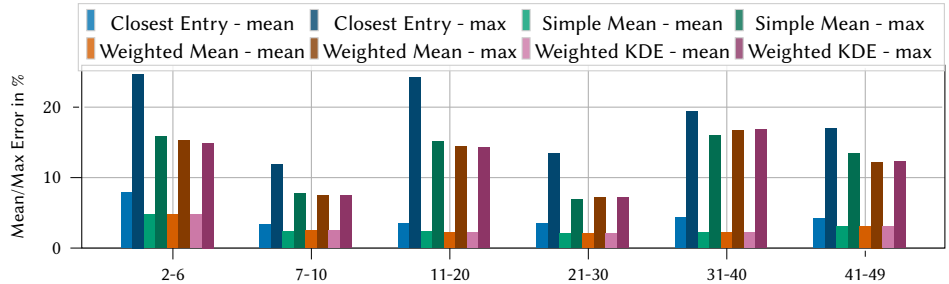
*5.7.1 Most similar CGRA's clock frequency.* In a first naive approach, one can take the nearest CGRA's critical path's length as a coarse approximation. As can be seen in Figure 17a and in Figure 16 with Group Size set to one, this often performs badly. However, when compared to the later-explained approaches it works best if the reference set contains only few and sparsely distributed reference points. This can be seen when comparing estimation results, which were generated with the larger reference set Sparse 1 (Figure 17a) to results from the smaller Sparse 4 reference set (Figure 17c). Nonetheless, if the nearest CGRA is too far away, there is a risk of larger error spikes (e.g. Figure 17b, 11-20 PEs).

*5.7.2 Mean value over closely related CGRAs.* Another approach computes the arithmetic mean value over the $n$ nearest CGRAs. As there is no weighting of references by their distance, the impact of a slightly more distant reference equals the impact of a very close reference. All neighbours are equally taken into account. This seems to be the best choice, if there are multiple equally good references, which are hard to distinguish concerning their quality. This holds especially for evenly-distributed and dense reference sets (Fig. 17a). Notice that the estimated value can be affected by even a single outlying reference. To avoid visual congestion, Figure 18 includes five instead of 10 unweighted dotted reference neighbours. Four dotted blue references are close-by. One dotted red reference is far-away outlying. Although one would expect the estimated value being located close-by to the most nearby references (4x dotted blue), the unweighted mean (dashed black) tends to be affected by to the right towards the single outlying unweighted reference (Unweighted Reference 5 - dotted red). Therefore, its estimation accuracy, when applied to a coarse reference set (green in Fig 17b and 17c 2-6 PEs).

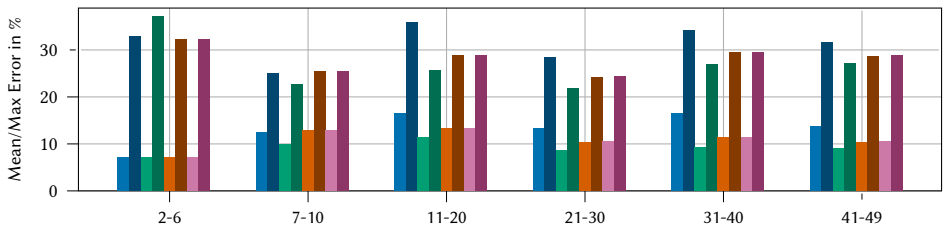
*5.7.3 Weighted Mean.* This is an approach for using the distance value to introduce a weighted mean, which accounts all neighbouring CGRA's, but varies their impact on the result according to the individual multidimensional distance. Each entry is weighted inverse to its distance. This approach is a mixture of the first two approaches (Most similar CGRA & Simple Mean). Against our intuition, it often is neither the best, nor the worst solution (Fig. 17c).

Depending on an optional weight factor, it behaves more like one or the other. In our experiments, we always set this factor to *1*. Taking the individual estimations into account, this value can be both, too high and too low. If the factor is too high, it behaves like taking the closest CGRA. If the factor is too low, it behaves like an unweighted mean (Fig. 17c, 41-49 PEs). We expect additional performance gains, if this factor was dynamically optimized based on the individual reference set's local quality (Fig. 17c, 11-20 PEs).

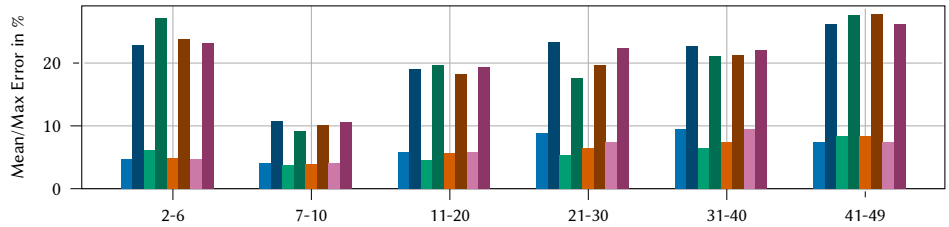
Besides the unweighted mean (dashed and dotted lines), Figure 18 also contains a similar visualization for the weighted mean (solid lines). Again, there are four solid blue references and a fifth outlying solid red reference. Compared to the unweighted references (dotted), each weighted reference's impact (solid red/blue) is weighted and therefore reduced with growing multidimensional distance. The weighted mean (solid black) is less affected by a single outlying reference (solid red). Yet, it is still affected. To overcome this issue, the weight factor of close (solid blue) neighbours can be more emphasized. Again, if the number of low-weighted distant entries strongly increases, also their impact grows, and the problem remains. Instead, if the weight factor is chosen too strong, results become comparable to our first solution presented in section 5.7.1, as only the closest entry is taken into consideration. This leads us to the next and last approach.




(a) Comparing different regression approaches when using 898 references (Sparse 1, Comb BSS)




(b) Comparing different regression approaches when using 104 references (Sparse 4, Seq. BSS)




(c) Comparing different regression approaches when using 70 references (Sparse 4, Comb. BSS)

Fig. 17. Comparison of estimation accuracy for different regression approaches (Closest Entry, Simple Mean, Weighted Mean, Weighted Kernel Density Estimator) with a decreasing number of references (898, 104, 70).

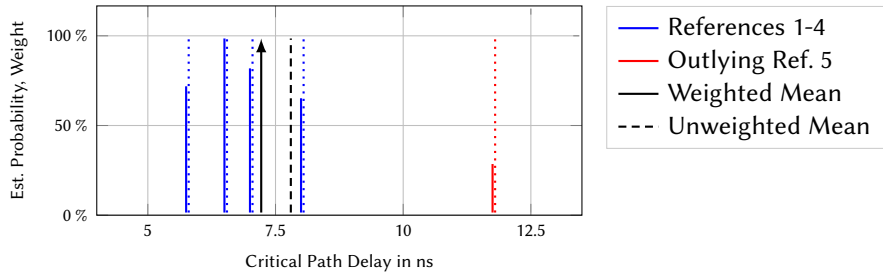



Fig. 18. Comparing (solid black) Weighted Mean to (dashed black) Simple Mean: The blue references are accompanied by an outlying red reference. Whereas the Simple Mean's (dotted blue/red) references are all equally strong (high), each weighted mean's (solid red/blue) reference impact (height) is reduced according to its distance. Whereas the dashed black Unweighted Mean tends to be distorted towards the right direction of the outlying red reference, the solid black Weighted Mean strongly limits the impact of the red outlier.

*5.7.4 Weighted Kernel Density Estimator (KDE).* As the Statistical Mode points to the reference values, which occur most often, it will be resistant to outliers. However, the Statistical Mode relies on the existence of multiple instances of the same value, which is not the case for our huge and sparsely populated reference set. A workaround would be to first form discrete clusters and then compute the Statistical Mode to let it point to a cluster. However, result quality will depend on the width and thereby size of these clusters. Statically configuring the clustering width is difficult and likely introduces additional errors.

As an alternative, and based on our previous experience, we implemented a mixture of the discussed ideas - a weighted Kernel Density estimator. The approach is visualized in Figure 19. Comparing the weighted mean's computation first, all references (blue and red) are placed on the x-axis. Next, they are weighted with the inverse of their distance to the CGRA to be estimated. Finally, their curves are accumulated together. This results in the black dashed and dotted curve. The point of highest kernel density is then picked and its x-axis value (at the dotted vertical line) is returned as the estimated delay value.

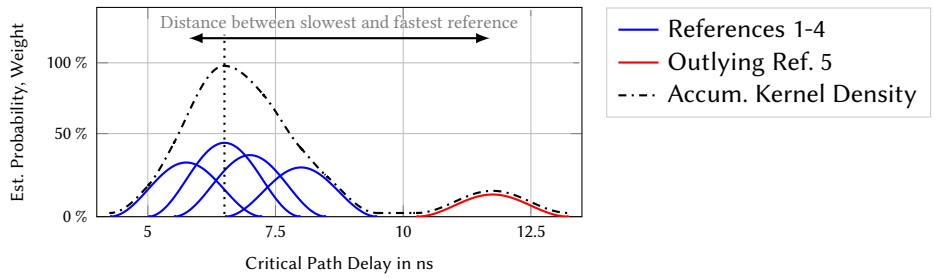



Fig. 19. Regression with a weighted KDE algorithm, using a group of similar CGRAs, thereby perfectly excluding the red outlier.

After evaluating various static and dynamic widths for the reference curve, we set the width of the curve to half the distance between slowest and fastest reference point. When choosing wider curves, the estimator behaves more like a weighted mean. Choosing tighter curves, behaviour is similar to picking the nearest CGRA. Therefore, reduced width again tends to decrease the mean error at cost of higher estimation error peaks.

On one hand, a close-by reference point does not solely dictate the result and on the other hand its potential noise impact is reduced by few other close neighbours. Single outliers are excluded and cannot degrade estimation quality. In our eyes, this approach is most suitable for the analyzed problem. It often is not the optimal solution, but has limited error peaks (Fig. 17a, 17b & 17c). The choice of a weighted KDE for the step of regression does not always lead to the highest accuracy, but promises the best overall robustness.

## 6 EVALUATION

There are two major metrics to evaluate estimation quality. These are relative mean and maximum error, which reflect the accuracy of an estimation. In our case there is a third crucial metric, which is the run-time of an estimation. This is important, since we need a fast estimation due to its use in SA. Both estimators are evaluated with these three metrics. Firstly, we go into the reference sets. The grid for their generation has been introduced in Section 3. Secondly, a comparison of the best possible accuracy is presented. Thirdly, the accuracy of both estimators is evaluated when only a sparse reference set is available. Eventually, the run-times of both estimators are compared.

For the statistical estimator, we split the design space in half by distinguishing between CGRAs with a combinatorial BSS and CGRAs with a sequential BSS. For best performance, the neural network is trained for the complete reference set and the design space is not splitted.

### 6.1 Recap: Reference Sets

Referring to Table 2 in Section 3.3, there are five sets available for training of the neural network or as the reference set for the statistical estimator. One set contains all instances and four sets are sparse as shown in Figure 20. The full set contains more than 11k entries (fully-synthesized CGRAs) and required high effort during a long creation process. It is used for the evaluation of the best possible accuracy. The four sparse sets vary between 174 and 2,746 entries in total. The smallest set contains 70 CGRAs with a combinatorial BSS. Considering the range of 2 to 49 PEs, there is only one point of reference with the same amount of PEs for many CGRAs.

The full set is generated by the tool described in Section 3 using a relatively fine grained segmentation. Each sparse set is generated using a more coarse segmentation. However, instead of actually synthesizing new CGRAs, the full set is loaded and unnecessary entries are discarded. This still ensures the best possible coverage of the design space for the sparse sets, but allows us to reuse the previously generated instances. A completely exclusive run is again used to generate the test set, which builds the basis to determine the accuracy of both estimators. The separate set avoids a bias of the test set as good as possible. Additionally, it ensures good coverage of the whole design space for the test set and therefore strengthens the resilience of the presented results.

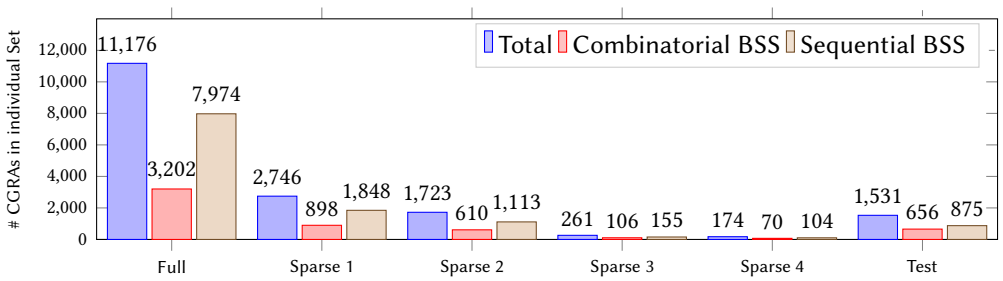


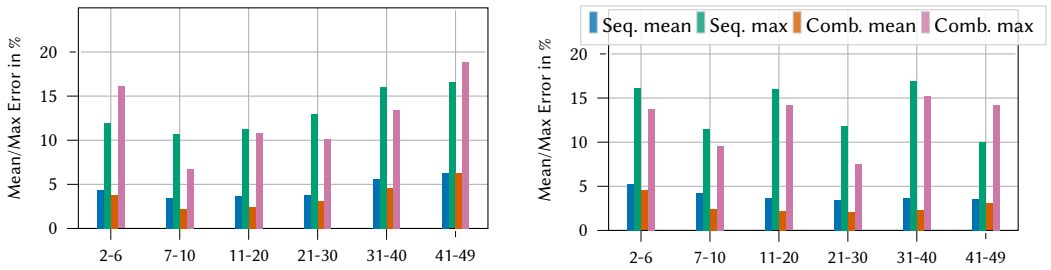

Fig. 20. Reference and Test Sets together with their BSS type.

### 6.2 Best Possible Accuracy

We present the best possible accuracy of both estimators to show their potential under the best available circumstances. Both estimators use the full set as a reference set. For the statistical estimator, we use the Mahalanobis distance function for selection and a selection PE-window size of ±10. The regression is realized by a weighted kernel density estimation with a Group Size of 10.

As mentioned, the results are split into two groups, one for a combinatorial and one for a sequential branch selection. Figure 21 shows the results for both approaches. The overall worst estimation error is comparable and below 20 % for both approaches. Depending on the area within the design space either one or the other approach is more accurate. The statistical estimator performs better in terms of maximum error for CGRAs with 41-49 PEs and partly for 2-6 and 21-30 PEs, whereas the ANN's maximum error is smaller within the remaining PE ranges. The given number of references is sufficient for the ANN, to effectively limit its maximum error. However, for larger CGRAs this comes at the cost of an increased mean error. For CGRAs with 21 and more PEs, the statistical estimator's mean error is clearly better than the ANN's mean error. To sum up, although there are different strengths, both approaches are comparable.

In comparison to our prior work [33], mean and maximum error are mostly similar. Caused by an increased number of estimations, some PE-regions face a slightly increased maximum error. These are limited by the use of a reference selection PE-window and the use of the Mahalanobis distance function. Concerning the path analysis, CGRAs with a combinatorial branch selection are easier to estimate, since they appear to have fewer critical path types in general and more CGRAs have the same critical path type. When analysing the results itself, one can see that the absolute estimation error for both groups are in the range of 0 ns to 1 ns. The total clock period of CGRAs with a combinatorial branch selection is higher, leading to a smaller relative error for the same absolute estimation error.




(a) ANN (b) Statistical Estimator

Fig. 21. Best possible accuracy: mean/max error over PE amount - combinatorial/sequential.

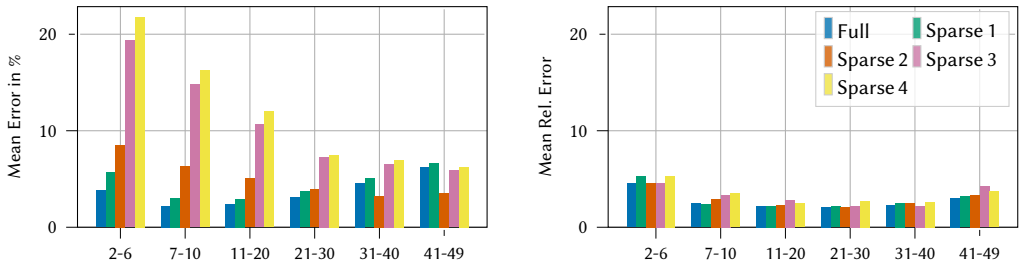
### 6.3 Handling a Sparse Reference Set

The major challenge of our and many other estimation tasks is a high overhead of generating or even finding reference data of decent size. The greatest advantage of the presented statistical estimator is its robustness towards the size of the given reference set and its good accuracy in general. The following section demonstrates how well the proposed statistical approach performs for a sparse reference set. We use the sets Sparse 1 - 4 as a reference set and for training of the neural network. The test set is used for evaluation as in the chapter before. First, the mean error is discussed and second the error mass function is presented. We decided to show the trend of

accuracy using only the mean error to spare space, since the max error is implicitly contained in the error distribution functions illustrated in Figures 24b and 25b and the related Tables 5 and 6.

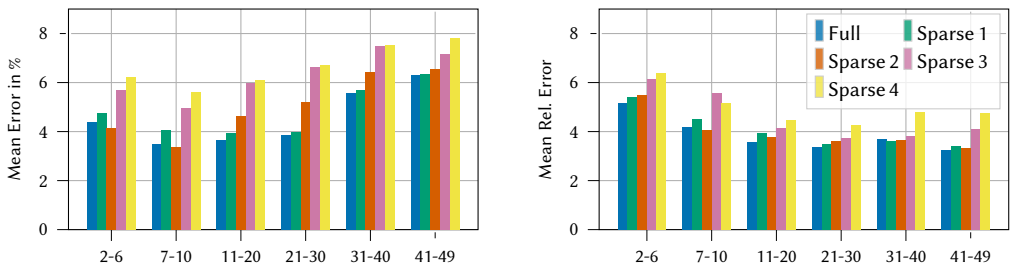
The results for a combinatorial branch selection are illustrated in Figure 22. Results for a sequential branch selection can be found in Figure 23. The best possible accuracy using the full set is added for orientation. The most important finding is the superior robustness of the statistical estimator to the amount of references. The accuracy decreases minimally with a decrease of references. In contrast, the accuracy of the neural networks becomes significantly worse for the sets Sparse 3 and 4. Considering a combinatorial BSS, the loss in accuracy for instances with less than 31 PEs is notably drastic. Yet, the decrease in accuracy is surprisingly acceptable for more than 30 PEs.

Two trends can be identified. The mean error decreases for both estimators and all sets with an increase in amount of PEs for a combinatorial branch selection. The accuracy for instances with a sequential branch selection has an oppositional behavior. In contrast to the other sets, the ANN seems to perform very well with Sparse 2, particularly for CGRAs with more than 31 PEs.




(a) ANN (b) Statistical Estimator

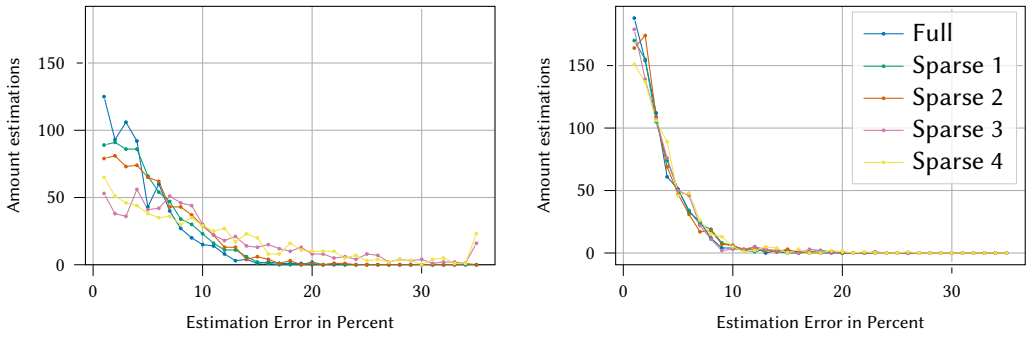
Fig. 22. Mean error over PE amount and different reference sets - combinatorial BSS.




(a) ANN (b) Statistical Estimator

Fig. 23. Mean error over PE amount and different reference sets - sequential BSS.

The arithmetic mean error is a good indicator for accuracy, but does not include peaks and error distribution. Figure 24 and Figure 25 show a visualization of the error distribution for all sets. They are based on the same data as the related Tables 5 and 6. Data is denoted in a fashion that error values are ceiled to the next integer. The error distribution of the statistical estimator shows that even with a sparse set of only 174 references, the absolute majority of CGRAs are estimated with an error of below 15 % for combinatorial branch selection and below 20 % for sequential branch selection. The neural networks still performs well, but over 36 instances have an estimation error of 30 % or higher for Sparse 4.




(a) ANN (b) Statistical Estimator

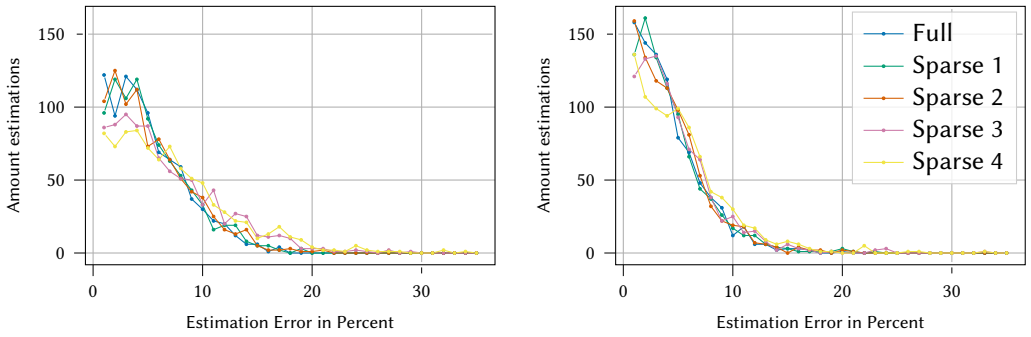
Fig. 24. The number of estimation errors is compared across differently-sized reference sets. CGRAs with combinational Branch Selection Signal (BSS) from the Full or Sparse 1-4 sets are used to estimate combinatorial-BSS CGRAs from the Test set.

Table 5. Error comparison for the sparse sets - combinatorial

| | Full | | Sparse 1 | | Sparse 2 | | Sparse 3 | | Sparse 4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Error in % | Stat | ANN | Stat | ANN | Stat | ANN | Stat | ANN | Stat | ANN |
| $[0-4)$ | 515 | 416 | 504 | 352 | 516 | 307 | 501 | 183 | 483 | 206 |
| $[5-9)$ | 125 | 190 | 135 | 231 | 120 | 250 | 131 | 224 | 148 | 174 |
| $[10-14)$ | 13 | 44 | 13 | 67 | 15 | 81 | 16 | 105 | 17 | 121 |
| $[15-19)$ | 3 | 6 | 4 | 3 | 5 | 14 | 6 | 63 | 5 | 63 |
| $[20-24)$ | 0 | 0 | 0 | 3 | 0 | 3 | 2 | 31 | 2 | 42 |
| $[25-29)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 24 | 1 | 16 |
| $[29,\infty)$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 26 | 0 | 34 |

Both Tables 5 and 6 reveal that even though the max error of both estimators are quite comparable for the full set, the neural network results in worse peak errors for sparse reference sets in case of a combinatorial branch selection. In case of a sequential branch selection, the peak errors are similar or the neural network performs slightly better. The mean error of the statistical approach is better for all sets as mentioned previously. Both estimators improve when the size of the reference set is increased. One can see that the quality of the neural network scales better for a combinatorial BSS, but starts off significantly worse for the smallest reference set Sparse 4. The high number of peak errors for the sets Sparse 4, Sparse 3 (and Sparse 2) is an indicator that the neural network requires around 600 training elements at least, in order to train the network properly.

In summary, one can note that the statistical approach has smaller mean error and appears to be more robust in the case of a combinatorial branch selection. When only a very limited number of references is given, we draw the conclusion that a statistical approach is the better choice. In terms of scalability, the statistical estimator seems to work well. Nevertheless, the trend of being worse for a sparse set but equally good for 12k references, the neural network is most likely the better choice if there is a well sized reference set.




(a) ANN (b) Statistical Estimator

Fig. 25. Error distribution - sequential BSS.

Table 6. The number of estimation errors is compared across differently-sized reference sets. CGRAs with sequential Branch Selection Signal (BSS) from the Full or Sparse 1-4 sets are used to estimate sequential-BSS CGRAs from the Test set.

| | Full | | Sparse 1 | | Sparse 2 | | Sparse 3 | | Sparse 4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Error in % | Stat | ANN | Stat | ANN | Stat | ANN | Stat | ANN | Stat | ANN |
| $[0-4)$ | 557 | 449 | 544 | 440 | 524 | 443 | 505 | 356 | 436 | 322 |
| $[5-9)$ | 265 | 325 | 268 | 325 | 286 | 308 | 288 | 309 | 331 | 318 |
| $[10-14)$ | 44 | 90 | 51 | 95 | 54 | 108 | 64 | 148 | 81 | 152 |
| $[15-19)$ | 9 | 11 | 7 | 15 | 8 | 13 | 12 | 48 | 19 | 61 |
| $[20-24)$ | 0 | 0 | 5 | 0 | 3 | 3 | 5 | 10 | 5 | 14 |
| $[25-29)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 2 | 5 |
| $[29,\infty)$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 3 |

## 6.4 Run-Time of the Estimators

*6.4.1 ANN.* While the accuracy dictates whether the quality of an estimator is sufficient, its run-time can be also crucial for different use-cases. We measured the setup and run-time of both estimators. The former denotes initialisation steps, such as copying reference data into memory, which only have to be done once before the estimator can start computing. The latter measures the average time in which exactly one CGRA can be estimated. Run-times were captured on an AMD EPYC 7501. Since graphic cards contain many cores and are commonly used for ANNs, the run-time on an RTX 2070 SUPER is also included for the ANN. Each approach was measured 100 times on all available test data. The mean is presented. Table 7 depicts the results of the ANN. Computation for one instance lies below 1 ms, which is more than satisfying if used in SA.

Table 7. Setup and estimation times for the ANN

| | ANN (CPU) | ANN (GPU) |
|---|---|---|
| Setup-Time | 49.62 ms | 39.86 ms |
| Run-Time | 50.47 $\mu$s | 26.12 $\mu$s |

*6.4.2 Statistical Estimator.* The run-time of the statistical estimator depends on multiple parameters. For a better overview, we decided to provide three different run-time benchmarks (A,B,C), which cover different areas of the design space (Table 8). We will first discuss the run-time benchmark estimation steps to then elaborate on run-time influencing parameters. Again, we use the *Sparse 1* reference set with a Selection Window size of 10 and a nearest neighbours size of 10.

Within a given and static area of the design space, the multidimensional grid containing the references can be established once, and then be reused for multiple estimations. First, the estimation algorithm loads and pre-selects relevant references. Compared to the ANN, the statistical estimator uses the original `JSON` files of fully-parameterized CGRAs. BSS type and window size parameters specify a subset of CGRAs to be loaded. Benchmark A estimates 19 CGRAs with 31 PEs each. After the initial loading the algorithm keeps 420 reference CGRAs within the range of 21 to 41 PEs (45.86 s). Second, the algorithm computes the coordinates of these reference CGRAs, which takes 5.07 s for benchmark A. After preparing the references, the estimator is setup by loading the CGRA for estimation (70 ms).

The following estimation step is three-fold. First, the coordinates of the CGRA to be estimated are computed (16.05 ms). Second, the distance computation towards the reference CGRAs and the selection of nearest CGRAs is performed (143ms). Third, the regression within this nearest group is computed (0.95 ms).

When comparing benchmark A to benchmark B and C, the run-time of the first two preparation steps depends on both, the number of reference CGRAs, and the number of PEs per CGRA. Based on caching, the setup and loading becomes quicker when estimating more CGRAs. Next, the run-time of the first estimation step, which is the computation of coordinates of the CGRA to estimate, depends on the CGRA's complexity in terms of # PEs, but likely also on the # operators. The second estimation step, which is the distance computation and selection of nearest CGRAs, depends on number of references and the non-constant-time distance sorting algorithm. Finally, the regression depends on the number of nearest CGRAs. With 10 nearest CGRAs for benchmarks A,B, and C, this results in almost constant time for this step, which is below one ms.

Table 8. Preparation, setup, and estimation times for three benchmarks using the statistical estimator

| Benchmark | A | B | C |
|---|---|---|---|
| # Benchmark References | 420 CGRAs | 256 CGRAs | 162 CGRAs |
| # Benchmark Estimations | 19 CGRAs | 14 CGRAs | 2 CGRAs |
| Benchmark size of reference CGRAs | 21 - 41 PEs | 37 - 49 PEs | 2-12 PEs |
| Benchmark size of estimates CGRAs | 31 PEs | 47 PEs | 2 PEs |
| 1. Once: Loading & pre-selection of all references | 45.86 s | 29,44 s | 15.94 s |
| 2. Once: Compute Coordinates of all references | 5.07 s | 2.66 s | 1.27s |
| Setup: Loading CGRA for estimation | 70 ms | 83 ms | 296.5 ms |
| 1. Estimate: Compute coordinates for CGRA to estimate | 16.05 ms | 10.27 ms | 7.5 ms |
| 2. Estimate: Compute distance, sort, select 10 nearest CGRAs | 143 ms | 97.86 ms | 188 ms |
| 3. Estimate: Regression within 10 nearest CGRAs | 0.95 ms | 0.93 ms | 0.97 ms |
| Estimation time per CGRA | 160 ms | 109.6 ms | 196.47ms |

The ANN clearly outperforms the statistical estimator in terms of run-time. The ANN is about 300 times faster than the statistical estimator on an AMD EPYC and about 600 times faster on a GPU. The time consumed to transfer data to the GPU and back to CPU is not included. Hence, if a GPU is used, one should consider transferring bunches of CGRA in order to cover up the transfer time. The gap between the ANN and the statistical estimator will never be closed, but it should

easily be possible to narrow it down. There are two things that should be taken into account: First, the ANN uses a highly optimized framework for computation — TensorFlow. Second, the statistical estimator was not implemented with the aim for having a short run-time. Rather it was optimized for accuracy, maintainability, and easy modifications. For instance, all phases are modularized. This results in iterating the reference set multiple times, even though it could be done once.

## 7 RELATED WORK

Related work can be discussed from two different points of view. First, publications that deal with the task of estimating hardware properties of computing architectures are discussed, since they set the standards. Second, related work proposing similar methodologies is discussed, since neural networks are non-specific estimators. Other estimation domains face the same challenges as we do. The focus lies on a sparse reference set and the representation of the instance that should be estimated. The representation of our system is a key aspect since CGRAs have individual typologies. This brings challenges, especially for the neural network.

### 7.1 Hardware Property Estimators

There are only a few publications that deal with maximum clock frequency estimation of designs. Not all of them actually attempt to estimate it. There are multiple related approaches that are relevant to be discussed. Table 9 summarizes the five relevant publications. The applied methodology and the accuracy in terms of mean and maximum error are illustrated. The column *Implemented* states, whether the estimation is done for a design that has already been implemented. There is one limitation to four of the five works: They are not able to estimate heterogeneous designs with an irregular interconnect. The proposed approaches exclusively deal with homogeneous structures.

Table 9. Overview of related works

| | Work | Methodology | Input | Implemented | Mean/Max Error |
|---|---|---|---|---|---|
| [21] | CGRA-ME | static timing analysis | model | yes | 9.6 % / n.a. |
| [34] | Yan et. al. | static timing analysis | model | yes | 8% / n.a. |
| [27] | Suh. et. al. | implicit / a priori | model | no | n.a. / n.a. |
| [26] | Sokolovic et. al. | analytical | VHDL | no | 0,89 % / 3,39 % |
| [20] | Monostiro et. al. | neural network | parameters | no | 22,22 % / n.a. |

As in our case, many DSE tools require timing or clock period information. CGRA-ME offers a generic mapper for an architecture that can be user-specified. An extension [21] offers performance estimation based on a profile, which has to be delivered by the user. In order to generate such a profile, an implementation of a design is required. Hence, the approach can not be used in a DSE as the major goal is to avoid pushing all potential solution through synthesis. However, the tool analyzes which paths of the CGRA are used, based on the mapping of an application. Then, a static timing analysis (STA) is applied and the clock frequency is adjusted. It is basically a application dependent overclocking of the CGRA. This approach has a mean error of about 9.6 %, the maximum error is not given. [34] presents an almost identical approach with a mean error of 8 %, a maximum error is not mentioned. [34] allows estimation for the presented architecture for the Xilinx Virtex, while [21] presents an approach that is suitable for arbitrary architectures and technologies as a profile is provided. Both approaches appear to be quite useful to speedup applications. However, the use of a static timing analysis requires a final implementation. We face the challenge of composing a system that has varying structure depending on the composition. One could transfer a static

timing analysis to our problem, but our analysis of our reference set indicates that this approach is most likely too error prone. Small changes in the design can lead to significant implementational differences. This makes STA not applicable for a DSE of a composition.

Suh et. al. manually explore several compositions and evaluate their performance for the Samsung Reconfigurable Processor in [27]. They also discuss the impact of heterogeneity. Knowledge of the architecture is implicitly used to find a composition that yields a high clock frequency, but clock frequency estimation is avoided purposely.

Sokolovic et. al. present a VHDL-Simulator in [26] that is capable of clock frequency estimation for synchronous circuits. The simulator requires a VHDL description of the design. The approach proposed by Sokolovic et. al. yields highly accurate estimations and offers huge flexibility. Unfortunately, the run-time of the simulator is not stated. We assume that the run-time is not feasible for our case, since a given design needs to be exported, loaded into the simulator, then elaborated and eventually simulated. Hence, the overhead of integration and the total run-time will probably take dozens of seconds in a best case scenario.

As in our case, [20] starts the parameterization of an FIR filter and a neural network is used to estimate the maximum clock frequency on a Xilinx Virtex II device. In contrast to the presented CGRA design space, the reference set is relatively big with about 225 instances for training, considering that the FIR is configured by only three parameters - depth limit to 7 *levels*, maximum width to 8 *components per level* and *input word-lengths* limit of 20 bits. The validation is done with 25 instances and results in a mean error of 22.22 %. A maximum error is not mentioned. Their accuracy is moderate, yet underlines that clock frequency estimation is feasible for neural networks.

### 7.2 Methodological Point of View

To our knowledge, only [20] uses ANNs to estimate hardware properties. However, the realization of ANNs face many challenges we face as well. Many properties of these challenges are common (few training data, noise, or a giant design space) and many approaches exist on how to deal with them. In this section we investigate these approaches.

The main problem while implementing our ANN is the high dimensional input combined with the sparse reference sets. These difficulties are common in the area of ANNs. In [13], a method is proposed with respect to a small reference set with many dimensions. They use a two-step approach: first selecting the important features and then finding a suitable architecture. While providing algorithms for these steps, judgement of a person is still necessary to make the final decisions. In our case we did this without an algorithm but trial-and-error.

Regarding dimensionality reduction, a wide variety of algorithms exists. Principal Component Analysis (PCA) uses eigenvectors and the covariance matrix to transform highly-dimensional inputs into a low-dimensional input vector. Other algorithms like Kernel PCA (KPCA) or Independent Component Analysis (ICA) also exist, a comparison is published in [3]. Dimensional reduction with a neural network is possible with the already mentioned autoencoders. Section 4.1.3 describes the topology of the most basic autoencoder. There exist different structures which could be adapted into our problem. A comparison between three types of autoencoders can be found in [28].

Another suitable approach for our estimation are Graph Neural Networks (GNNs) [25]. They are mentioned in Section 4.2. If we can map graph data directly onto ANNs, we would not have to first serialize the data into an input array. This step causes problems like the possible high-dimensional vector referring to the interconnect. GNNs map to every existing node in the graph a feature vector dependent on its neighbours. These vectors are transferred into an ANN. Even though this would solve the problem regarding the interconnect, our CGRA model contains heterogeneous nodes. A

solution for this is proposed in [35] with Heterogenous Graph Neural Networks. The area of GNNs is still actively researched and new achievements are continuously made, which is why we did not implement this approach yet.

## 8 CONCLUSION

Estimation of the maximum clock frequency is crucial for a reasonable design space exploration of freely composable CGRAs. In this article, we have presented an in-depth analysis and discussion of an ANN and an optimized version of a statistical estimator [33]. A reference set of about 12k synthesized CGRAs builds the basis for the evaluation. Using all available CGRAs as a reference and for training, the mean and maximum errors of both estimators are comparable (2 % - 6 % and below 19.5 %). If the reference set is thinned out, the statistical estimator shows robustness in accuracy even if a couple of dozens references are left — merely 0.7 % instances of the test set have an error above 20 %. The accuracy of the ANN on the other hand drops substantially — here 7.3 % of the of the test set have an error above 20 %. In terms of run-time, the ANN (25-50$\mu$s per instance) is considerably superior to the statistical estimator (108-198 ms). Non the less, the statistical estimator can still be considered as being blazingly fast when compared to multi-hour synthesis runs.

Future work will integrate the presented statistical estimator into Simulated Annealing for an automated design space exploration and evaluate heterogeneous and irregular CGRA compositions.

## ACKNOWLEDGMENTS

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity *ATHENE*.

## REFERENCES

[1] Pierre Baldi. 2011. Autoencoders, Unsupervised Learning and Deep Architectures. In Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27 (Washington, USA) (UTLW'11). JMLR.org, Bellevue, Washington, USA, 37–50.

[2] Mark Bohr. 2007. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. IEEE Solid-State Circuits Society Newsletter 12, 1 (2007), 11–13. https://doi.org/10.1109/N-SSC.2007.4785534

[3] L.J. Cao, K.S. Chua, W.K. Chong, Hp Lee, and Q.M. Gu. 2003. A comparison of PCA, KPCA and ICA for dimensionality reduction in support vector machine. Neurocomputing 55 (09 2003), 321–336. https://doi.org/10.1016/S0925-2312(03)00433-8

[4] Francois Chollet et al. 2015. Keras. https://github.com/fchollet/keras

[5] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2016. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).

[6] Div. 2020. Javadocs API 3.0 - Interface MultipleLinearRegression. https://commons.apache.org/proper/commons-math/javadocs/api-3.0/org/apache/commons/math3/stat/regression/MultipleLinearRegression.html

[7] Timothy Dozat. 2016. Incorporating nesterov momentum into adam. (2016).

[8] Martín Abadi et. al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[9] Daniele Grattarola and Cesare Alippi. 2020. Graph Neural Networks in TensorFlow and Keras with Spektral. arXiv:2006.12138 [cs.LG]

[10] Aurélien Géron. 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow - Concepts, Tools, and Techniques to Build Intelligent Systems. "O'Reilly Media, Inc.", Sebastopol.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. arXiv:1502.01852 [cs.CV]

[12] Peter J. Huber. 1964. Robust Estimation of a Location Parameter. Ann. Math. Statist. 35, 1 (03 1964), 73–101. https://doi.org/10.1214/aoms/1177703732

[13] Jen-Lun Yuan and T. L. Fine. 1998. Neural-network design for small training sets of high dimension. IEEE Transactions on Neural Networks 9, 2 (1998), 266–280.

[14] Lukas Johannes Jung and Christian Hochberger. 2018. Lookahead Memory Prefetching for CGRAs Using Partial Loop Unrolling. In Applied Reconfigurable Computing. Architectures, Tools, and Applications, Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and Pedro C. Diniz (Eds.). Springer International Publishing, Cham, 93–104.

[15] Karlrupp. [n.d.]. Karlrupp/microprocessor-trend-DATA: Data Repository for my blog series on Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data

[16] Yann Lecun and Y. Bengio. 1995. Convolutional Networks for Images, Speech, and Time-Series. The Handbook of Brain Theory and Neural Networks.

[17] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. ACM Comput. Surv. 52, 6, Article 118 (Oct. 2019), 39 pages. https://doi.org/10.1145/3357375

[18] Prasanta Chandra Mahalanobis. 1936. On the Generalised Distance in Statistics. Proceedings of the National Institute of Sciences of India 2, 1 (1936), 49—55.

[19] Manish Mishra and Monika Srivastava. 2014. A view of Artificial Neural Network. In 2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014). 1–3. https://doi.org/10.1109/ICAETR.2014.7012785

[20] Adam Monostori, Hans Holm Frühauf, and Gabriella Kókai. 2005. Quick Estimation of Resources of FPGAs and ASICs Using Neural Networks. In LWA. 210–215.

[21] K. Niu and J. H. Anderson. 2018. Compact Area and Performance Modelling for CGRA Architecture Evaluation. In 2018 International Conference on Field-Programmable Technology (FPT). 126–133. https://doi.org/10.1109/FPT.2018.00028

[22] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective. IEEE Access 8 (2020), 146719–146743. https://doi.org/10.1109/access.2020.3012084

[23] Johanna Rohde, Lukas Johannes Jung, and Christian Hochberger. 2019. Update or Invalidate: Influence of Coherence Protocols on Configurable HW Accelerators. In Applied Reconfigurable Computing, Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz (Eds.). Springer International Publishing, Cham, 305–316.

[24] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. CoRR abs/1609.04747 (2016). arXiv:1609.04747 http://arxiv.org/abs/1609.04747

[25] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009. The Graph Neural Network Model. IEEE Transactions on Neural Networks 20, 1 (2009), 61–80.

[26] M. L. J. Sokolovic and V. B. Litovski. 2005. Using VHDL Simulator to Estimate Logic Path Delays in Combinational and Embedded Sequential Circuits. In EUROCON 2005 - The International Conference on "Computer as a Tool", Vol. 2. 1683–1686. https://doi.org/10.1109/EURCON.2005.1630296

[27] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim. 2012. Design Space Exploration and Implementation of a High Performance and Low Area Coarse Grained Reconfigurable Processor. In 2012 International Conference on Field-Programmable Technology. 67–70. https://doi.org/10.1109/FPT.2012.6412114

[28] C. C. Tan and C. Eswaran. 2008. Performance Comparison of Three Types of Autoencoder Neural Networks. In 2008 Second Asia International Conference on Modelling Simulation (AMS). 213–218.

[29] Eclipse Deeplearning4j Development Team. 2016. ND4J: Fast, Scientific and Numerical Computing for the JVM. (2016). https://github.com/eclipse/deeplearning4j

[30] Dennis Wolf, Andreas Engel, Tajas Ruschke, Andreas Koch, and Christian Hochberger. 2021. UltraSynth: Insights of a CGRA Integration into a Control Engineering Environment. Journal of Signal Processing Systems 93 (05 2021), 1–17. https://doi.org/10.1007/s11265-021-01641-7

[31] Dennis Wolf, Tajas Ruschke, Christian Hochberger, Andreas Engel, and Andreas Koch. 2019. UltraSynth: Integration of a CGRA into a Control Engineering Environment. In Applied Reconfigurable Computing, Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz (Eds.). Springer International Publishing, Cham, 247–261.

[32] D. L. Wolf, L. J. Jung, T. Ruschke, C. Li, and C. Hochberger. 2018. AMIDAR Project: Lessons Learned in 15 Years of Researching Adaptive Processors. In 2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). 1–8. https://doi.org/10.1109/ReCoSoC.2018.8449384

[33] D. L. Wolf, C. Spang, and C. Hochberger. 2020. Towards Purposeful Design Space Exploration of Heterogeneous CGRAs: Clock Frequency Estimation. In 2020 57th ACM/IEEE Design Automation Conference (DAC). 1–6. https://doi.org/10.1109/DAC18072.2020.9218649

[34] Leipo Yan, Thambipillai Srikanthan, and Niu Gang. 2006. Area and Delay Estimation for FPGA Implementation of Coarse-grained Reconfigurable Architectures. SIGPLAN Not. 41, 7 (June 2006), 182–188. https://doi.org/10.1145/1159974.1134677

[35] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. 2019. Heterogeneous Graph Neural Network (KDD '19). Association for Computing Machinery, New York, NY, USA, 793–803. https://doi.org/10.1145/3292500.3330961