Journal of
Low Power Electronics and Applications

MDPI

Article

A Framework for Ultra Low-Power Hardware Accelerators Using NNs for Embedded Time Series Classification

Daniel Reiser [1,*], Peter Reichel [2], Stefan Pechmann [3], Maen Mallah [4], Maximilian Oppelt [4], Amelie Hagelauer [3,5], Marco Breiling [4], Dietmar Fey [1] and Marc Reichenbach [6]

1 Chair of Computer Science 3 (Computer Architecture), Friedrich-Alexander University Erlangen-Nuernberg, 91058 Erlangen, Germany; dietmar.fey@fau.de
2 Fraunhofer Institute for Integrated Circuits (IIS), Division Engineering of Adaptive Systems EAS, 01187 Dresden, Germany; peter.reichel@eas.iis.fraunhofer.de
3 Chair of Micro and Nanosystems Technology, Technical University of Munich, 80333 Munich, Germany; stefan.pechmann@tum.de (S.P.); amelie.hagelauer@emft.fraunhofer.de (A.H.)
4 Fraunhofer Institute for Integrated Circuits (IIS), 91058 Erlangen, Germany; maen.mallah@iis.fraunhofer.de (M.M.); maximilian.oppelt@iis.fraunhofer.de (M.O.); marco.breiling@iis.fraunhofer.de (M.B.)
5 Fraunhofer Institute for Microsystems and Solid State Technologies (EMFT), 80686 Munich, Germany
6 Chair of Computer Engineering, Brandenburg University of Technology (B-TU), 03046 Cottbus, Germany; marc.reichenbach@b-tu.de
* Correspondence: daniel.g.reiser@fau.de

Abstract: In embedded applications that use neural networks (NNs) for classification tasks, it is important to not only minimize the power consumption of the NN calculation, but of the whole system. Optimization approaches for individual parts exist, such as quantization of the NN or analog calculation of arithmetic operations. However, there is no holistic approach for a complete embedded system design that is generic enough in the design process to be used for different applications, but specific in the hardware implementation to waste no energy for a given application. Therefore, we present a novel framework that allows an end-to-end ASIC implementation of a low-power hardware for time series classification using NNs. This includes a neural architecture search (NAS), which optimizes the NN configuration for accuracy and energy efficiency at the same time. This optimization targets a custom designed hardware architecture that is derived from the key properties of time series classification tasks. Additionally, a hardware generation tool is used that creates a complete system from the definition of the NN. This system uses local multi-level RRAM memory as weight and bias storage to avoid external memory access. Exploiting the non-volatility of these devices, such a system can use a power-down mode to save significant energy during the data acquisition process. Detection of atrial fibrillation (AFib) in electrocardiogram (ECG) data is used as an example for evaluation of the framework. It is shown that a reduction of more than 95% of the energy consumption compared to state-of-the-art solutions is achieved.

Keywords: hardware architecture; neural networks; non-volatile memory; wearable AI; RRAM

Citation: Reiser, D.; Reichel, P.; Pechmann, S.; Mallah, M.; Oppelt, M.; Hagelauer, A.; Breiling, M.; Fey, D.; Reichenbach, M. A Framework for Ultra Low-Power Hardware Accelerators Using NNs for Embedded Time Series Classificatione. *J. Low Power Electron. Appl.* **2022**, *12*, 2. https://doi.org/10.3390/jlpea12010002

Academic Editors: Aatmesh Shrivastava, Vishal Saxena and Xinfei Guo

Received: 30 October 2021
Accepted: 27 December 2021
Published: 31 December 2021

1. Introduction

Atrial Fibrillation (AFib) is one of the most common forms of heart arrythmia. It causes the upper chambers of the heart to contract unrhythmically and beat out of sync with the lower chambers of the heart [1]. In recent years, the spread of the disease has been steadily increasing, especially among the elderly population. The early identification of AFib is crucial, as patients suffering from such a heart condition are very likely to be affected by severe heart diseases later in life. AFib can be detected by analysis of an electrocardiogram (ECG). Because the symptoms vary in severity and do not occur regularly, it makes sense to monitor this condition over a longer period of time. Such a long-time monitoring can be performed using a wearable device.

Commonly available hardware detectors identify symptoms using statistical properties of ECG signals defined by medical experts [1,2]. Neural networks can help detect abnormalities that manifest themselves on different time scales. They also generalize very well and can therefore handle data recorded with different measurement methods [2–4]. Since portable devices are generally battery powered, they have a strict power budget. Low-power hardware for realizing the NN and the data acquisition process is therefore necessary.

While many generic NN ASIC accelerators have been investigated in recent years, they often sacrifice higher power consumption for more flexibility [5]. For portable and other embedded devices, this higher power consumption can have a significant impact on battery life. Therefore, an adapted architecture for a system that has minimal energy overhead for a specific classification task is necessary. In addition, usually only individual parts of such a system are considered for optimization, rather than choosing a holistic approach. Minimizing energy consumption only works if the optimization of all system components is coordinated with each other, from NN algorithm design to static power dissipation at transistor level. To solve this problem, we present a novel low-power hardware generation framework for embedded time series classification tasks using NNs. It contains all existential points necessary for a holistic optimization of the entire system, which are:

- An optimized hardware architecture for minimal energy consumption, where the data flow is implicitly given by the architecture to save energy overhead in control logic and data routing.
- A power-down operation mode with minimal energy consumption of the system during data acquisition.
- Non-volatile, multi-level memory in form of RRAM cells as embedded storage, to avoid external reloading of parameters after a power-down.
- A Neural Architecture Search (NAS), which optimizes the NN configuration for accuracy and energy efficiency at the same time.
- An architecture generator, which translates the NN into a complete accelerator hardware based on our architecture, which can be manufactured as an ASIC.

This paper is structured as follows: Section 2 gives an overview of related work, such as NN optimization strategies and hardware implementations of NNs and AFib detectors. In Section 3, important properties of time series classification tasks are identified and the hardware architecture concept is derived from them. We present our framework for designing the embedded classification system with a NN in Section 4. In Section 5, the framework is used to create an AFib detection system and an ASIC is implemented. An evaluation of this AFib detection ASIC is carried out in Section 6. Finally, Section 7 concludes the observations and presents future work.

2. Related Work

The automated detection of abnormalities using recordings of the electrical activity of the heart, such as arrhythmias or morphological changes, is extensively studied and is already commercially available on clinical devices [2,6,7]. However, the rise of wearable technologies and the associated changes of acquisition technology, as well as the increased number of recordings in daily routine, pose new challenges to the research community. In contrast to architectures based on predefined feature extraction, NNs generalize very well and can therefore be used for classification using data that is noisy or measured using different measurement methodologies. NNs can also be retrained and improved using new data distributions from different patients, hospitals, or even novel ECG devices with different electrode placement.

Implementing such a NN for an embedded ECG use case in an energy efficient way is quite challenging, as the available energy budget can be really low. Some approaches circumvent this problem by sending the raw data from the sensor to a central processing unit using wireless transmission [8]. However, in practice this approach is limited by the availability and range of the wireless connection. Burger et al. present an approach for an

embedded ECG classification use case realizing NNs on small FPGA devices [9]. However, inherently the reconfigurability of the applied FPGA devices adds overhead in hardware and limits the optimization potential to apply beneficial improvements like power gating in parts of the design or using novel memory technology. Loh et al. presented a DSP-based approach [10]. While a significant reduction of the energy per classification was achieved, the data acquisition process in a wearable use case has not been considered. We will later show that this contributes a huge amount to the overall power consumption. Therefore, the available concepts do not cover all the requirements for an efficient low-power system.

On the other hand, a lot of research has been done in recent years regarding efficient hardware implementation of NNs. One important factor is the optimization of the NN itself in order to reduce the amount of data to be processed and thus save energy [11,12]. Optimizations such as quantization or pruning can often be applied without significant loss of accuracy. However, this is only possible because NNs, especially unoptimized ones, contain a high degree of redundancy. To avoid this redundancy from the beginning, the framework we present uses a so-called NAS during the design phase of the NN [13]. This helps to find an optimized NN with a high accuracy and a low number of parameters. Since such a NN has far less redundancy, post training optimizations, such as pruning, are no longer necessary. In contrast to state-of-the-art NAS, we have extended it with features that are specifically adapted to our framework and architecture. These include strong quantizations per layer and minimization of the energy consumption for our hardware architecture as a second optimization goal. More details of our NAS are described in Section 4.1.

On the hardware site, emerging non-volatile memories such as RRAM devices are a good fit to be used as base for embedded low-power NN accelerators [14–16]. Calculations can be conducted in an analog fashion by sending voltage pulses through a crossbar array of RRAM cells which store the weight values of a NN. This approach often yields very low-power consumption. However, it has some disadvantages when considering integration into an overall system [15]. To perform these calculations, control logic is needed to manage the flow and sorting of the data. In addition, only individual layers can be calculated per crossbar array. Therefore, the results must be buffered in digital form and converted several times from the analog to the digital domain and vice versa for a more sophisticated NN with multiple layers. Furthermore, not all components of a NN can be mapped to such a crossbar array. Therefore, such arrays are often embedded in surrounding CMOS logic. However, the analog-digital/digital-analog conversion and control logic impose a significant energy overhead on an embedded application. To achieve an efficient overall system, we developed an architecture which calculates all values digitally and uses memory blocks containing RRAM cells for storing the weights. Thus, all components of a NN can be mapped individually without the need for conversion to another domain, while retaining the advantages of multi-level capability and non-volatility of the RRAM cells. This also means that such an accelerator can be easily integrated into a larger digital system.

3. Architectural Concept

To create an efficient architectural concept, the most important properties of an embedded time series classification task must be determined. We identified three main characteristic properties, from which we derived three main optimization strategies.

3.1. Reduction of External Memory Access

This strategy addresses the low energy budget available in an embedded device. For this reason, external memory access should be avoided using novel non-volatile memory technologies. For example, the usage of resistive random-access memory (RRAM) is promising, due to its ability to store more than one or two states per cell (multi-level-cell) [17]. This enables the storage of all parameters (e.g., weight and bias values of the NN) locally on-chip. At the same time, heavy quantization can be used in combination with a customized training process of the NN to adapt it exactly to the target hardware (e.g.,

all weight values can be discretized to the three states −1, 0 or +1, and each weight value can be stored in exactly one RRAM cell). Moreover, the quantization also leads to smaller bit widths of operands, which results in trivial multiplications (e.g., with −1, 0, +1) and therefore minimizes the energy requirements for arithmetic operations [18].

3.2. Minimization of Control Flow and Buffering

This strategy exploits the times series property by employing a dataflow driven architecture, which minimizes the required control overhead. Similar to the principle of a systolic array, where a sequence of arithmetic operations is coded into the data path, no instruction set or programming is necessary for such an approach [19]. In contrast to systolic arrays, our architectural concept does not target generic circuits (e.g., matrix multiplication with arbitrary size). Rather, the developed framework introduces a tool, which synthesized the arithmetic operations to be executed into the data path per layer of the NN (e.g., for one convolutional layer).

3.3. Power-Down Operation Mode

This strategy utilizes the property of a low sampling rate that most embedded applications possess. This means that classification by the NN does not need to be performed continuously, as data is collected at a low rate. A more efficient solution (especially regarding the static power) is to collect data for a period of time and then classify it at once. In common architecture approaches this is often not useful, because activating the chip after a power-down (e.g., loading NN weights) requires more energy than the power-down saves. RRAMs are non-volatile and do not lose their stored value after a power-down. Therefore, using RRAM cells enables us to cut the power to the chip and save energy. This concept is also used in non-volatile processors and can contribute to significant energy savings, especially in the embedded context [20].

3.4. Combination of the Strategies

Using and interleaving these three optimization strategies leads to our generic architectural concept, which can be used to derive an application specific accelerator for a classification task using NNs. The key feature here is that the three strategies are not just considered and used separately but are cleverly combined. Figure 1 shows a schematic overview of this architectural concept. The design is partitioned into two power domains: the Data Control Core and the Processing Core. The Data Control Core handles the communication with the outside, the preprocessing and the buffering of the input data. This part of the design is always powered by a 1.2 V voltage rail (VDD1V2) for data acquisition. Since this core is always on, it also contains all the necessary controllers. This includes the RRAM manager, which controls read and write access to the memory blocks containing RRAM cells and thus enables reprogramming. The central control unit is also part of this core. Its main tasks are controlling the buffering of the data and the power modes. It puts the processing core into a low-power state by switching off its power supply (VDD1V2 and VDD3V3 voltage rails). The Processing Core contains the dataflow driven architecture for the NN, including multiple quantized processing elements as well as memory blocks containing RRAM cells. The power supply of this core can be cut to reduce the overall energy consumption while data is collected. After re-enabling the power supply, the parameters stored inside the non-volatile RRAM cells are available without having to be reloaded from an external memory. The multi-level property of the RRAM cells reduces the memory footprint and increases the storage density.

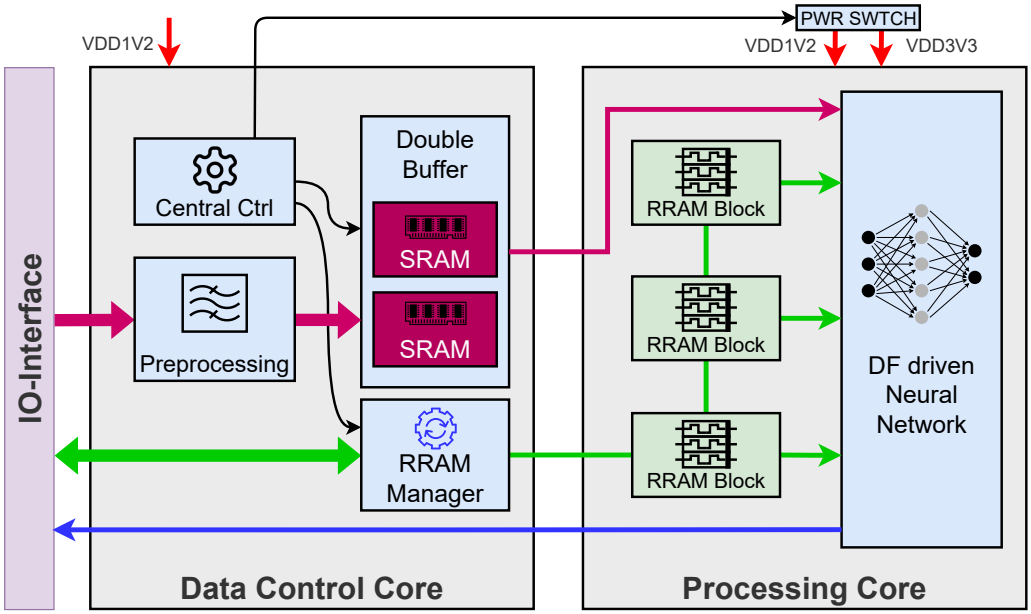

Figure 1. Schematic overview of our architectural concept.

4. Hardware Generation Framework

As discussed in Section 3, we derived a generic architectural concept for embedded classification tasks using NNs. To design a complete system for a specific classification problem and realize an ASIC using this architectural concept, a framework has been created. An overview of this framework can be seen in Figure 2. The flow can be divided into three main phases: *NN design*, *hardware mapping* and *hardware implementation*. This allows for the finding of an optimal implementation regarding accuracy and energy consumption of the final system. Red boxes in Figure 2 represent customizable, application specific constraints, such as the training data and the desired accuracy for the classification task. Green boxes represent fixed components specific to our proposed architectural concept. Concrete design steps in our approach are marked with blue boxes. Yellow boxes represent results obtained after each phase. This framework could be extended to include additional architectural concepts than the one presented in Section 3.

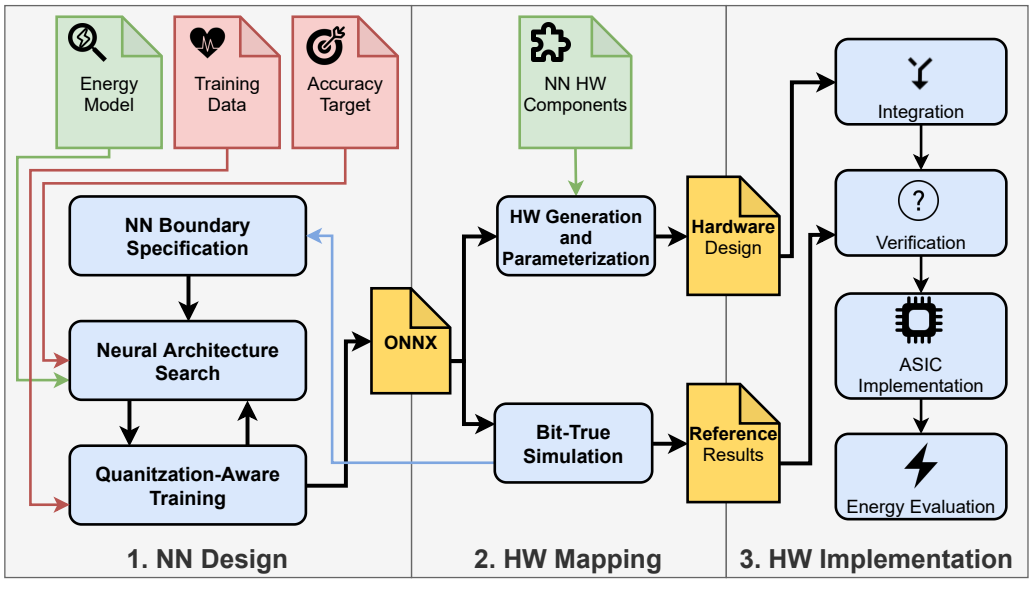

Figure 2. Overview of our hardware generation framework for problem mapping and implementation.

At the start of the design process, boundary conditions for the NN can be specified by the designer. This allows for the constraining of the design space of the NN and to integrate prior knowledge about beneficial NN configurations into the NN design process. Parameters include the type of layers and activation functions to be used as well as possible quantization levels for parameters and partial sums of different layers. In the next step an NAS is used, which finds a NN with minimal energy consumption for the desired accuracy. To evaluate, if the desired accuracy is reached, a full quantization-aware training of the NN is done. When an optimal network regarding a certain goal is found, the trained network is exported in *ONNX format*. This format provides a common standard for NN specifications [21]. In the second phase, the network defined this way can be imported into a self-developed *Bit-True simulator* written in Python. It uses a fixed-point library to exactly model the arithmetic behavior of the hardware on bit level, which leads to the aforementioned name. With the help of this tool an evaluation of overflow and quantization effects on the accuracy can be conducted. From this, possible options for the NN boundary conditions can be identified, especially suitable quantization levels. This flow greatly reduces the time needed for a design space exploration of different NN configurations, as up to this point no hardware needs to be generated. The results of the *Bit-True simulation* also act as reference for hardware verification later on. The *ONNX model* is also used in our *hardware generation and parametrization tool*, which parses and analyzes the network definition and creates an HDL design based on hardware components that we designed. In the third phase, all hardware components are integrated, and the functionality is verified by testbenches. Then a hardware implementation can be performed using commonly available EDA tools. Our framework is not limited to a specific toolset, because the hardware components are available as HDL description or hardware libraries. After the implementation, an energy evaluation of the whole design can be carried out (with tools such as *Synopsys PrimePower*).

4.1. Neural Network Design

In our framework, the *NN design* plays an important role, as optimizations on the algorithm level have the biggest impact on the energy consumption of the final hardware implementation. Therefore, a lot of effort was directed towards creating a generic approach for finding an energy efficient NN and optimizing its so-called hyperparameters. Many hyperparameters of a NN have a big influence on the energy consumption of a hardware implementation. These hyperparameters include, but are not limited to:

- Length of the NN input (window)
- Number of convolutional layers and for each layer:
  - Filter length
  - Number of filters
  - Stride of each filter
- Number of Fully-Connected layers and number of neurons in each layer
- Activation function used
- Quantization of the weights
- Quantization of the partial sums and activations

However, these hyperparameters also directly affect the accuracy of the classification task and the time a classification takes. As the design space for creating a NN and defining theses hyperparameters is very broad, a trade-off between energy efficiency, accuracy and duration of the classification must be found. For this task an NAS can be used [13]. An NAS helps in finding an optimal NN configuration regarding certain optimization criteria in a defined search space. It can be distinguished by the search space that is covered, the search strategy that is used, and the performance estimation that is applied to compare different models.

In contrast to a state-of-the-art NAS, which often treats the target hardware as an abstract model, if at all, we have adapted our NAS specifically to our hardware architec-

ture [22,23]. We can therefore incorporate prior knowledge about time series classification tasks. As an example, we constrain our search space to reach shorter search times by limiting possible basic blocks of the NN to those that are well suited for time series classification while being efficiently implementable in hardware. These blocks include convolutional layers, Fully-Connected layers, ReLU activation functions and Max Pooling layers. With our framework, the search space can be constrained even further by setting possible quantization configurations, for weights, inputs and partial sums. These constraints can be specified as a configuration file and be adapted to other application needs by a designer. For the search strategy, we use a grid-based search to increase the speed of finding a promising model.

Unlike state-of-the-art NAS, our approach simultaneously maximizes classification accuracy and minimizes the actual energy of a hardware implementation based on our architecture. To find the optimal trade-off, we created an *energy model*, which helps to identify the impact of different hyperparameters on the overall energy consumption. The model specifies values for the power consumption of MAC operations (P_{MAC}) and RRAM cells (P_{mem}) for different configurations. These values are stored in a lookup table. They were extracted from hardware simulation of our architecture components and are not based on abstract energy models as in other approaches. The parameterization is done using the bit widths (i.e., quantization levels) of the input values (x_{bits}), weights (w_{bits}), and output values (o_{bits}), respectively. For MAC operations the consumed energy can be calculated by summing up the power estimates of all layers, parametrized with the respective bit widths and multiplying the sum with the duration of MAC operations, depending on the targeted clock frequency. The energy consumed by the memory can be calculated by summing up all power estimates and multiplying the sum with the overall duration of a classification. However, for comparing different models for the same application a fixed clock cycle and duration can be assumed and therefore, the sums of the power values can be compared directly. To compare the accuracy of different models, we further divide it into sensitivity (proportion of positive cases that are correctly identified) and specificity (proportion of negative cases that are correctly identified).

Our NAS is described by Algorithm 1. We initialize the search space with a model found by a random search and then search for better models iteratively. In each iteration, a small grid (set of adjustable hyperparameters) is defined, while the rest of the hyperparameters is fixed to limit the search space. The three metrics (estimated energy/power, sensitivity, and specificity) are then calculated for each candidate model after a full training. The best of these candidate models is then chosen, and a new grid is defined for the next iteration. The influence of the three optimization goals can be adapted towards a certain goal of the application by weighting them before comparison. To create an ultra-low-power accelerator, more emphasis can be given to the estimated power than to the sensitivity or specificity, for example. For applications, where it is critical to classify an event as soon as it appears for the first time, it can be useful to prioritize sensitivity over specificity as a false alarm might be better than missing the event. However, when aiming for long-term monitoring in a wearable device it can be even more important to optimize for estimated power, as thousands of classification runs will be done per day and a condition is likely to appear in multiple classification runs. The optimization is finished when no better NN configuration can be found.

It is difficult to compare the potential savings of our NAS against other NAS or optimization approaches from the literature since there are multiple degrees of freedom and NAS often targets generic hardware architectures. However, to get an idea of how much energy can be saved using our NAS, a fixed specificity and sensitivity can be chosen. Then the energy minimization is the only optimization criterion. In this case, energy savings of up to 70% compared to an initial NN configuration can be achieved. Therefore, this NAS is worthwhile and can yield far better energy consumption than by applying individual optimizations alone.

Algorithm 1 Our Neural Architecture Search.

```
// First NN model and grid are defined
Input: NN_start
Input: grid
Input: train_data, train_data, val_data

NN_model = NN_start
repeat
    metrics = analyze(NN_model, grid)
    NN_model = get_best_model(metrics)
    // define new gird for the new found NN
    Input: grid
until targeted metrics are reached

sens, spec = test_model(NN_model, test_data)
power_est = power_estimation(NN_model)
metric = {power_est, sens, spec}
Output: NN_model, metrics

Function analyze(model, grid):
    for hyperparams in grid do
        candidate = update_model (model, hyperparams)
        trained_model = train(candidate, train_data)
        sens, spec = test(trained_model, val_data)
        power_est = est_power(candidate)
        metrics.append({power_est, sens, spec})
    return metrics
```

Unlike the other hyperparameters of a NN, quantization needs special attention. It influences not only the arithmetic operations, but also the amount of memory required for the parameters of the NN. NNs usually allow for heavy quantization, because of their inherent redundancy. While general purpose accelerators, like CPUs or GPUs, use the same bit widths for all arithmetic operations, in an ASIC architecture it is useful to set this quantization per layer of a NN in order to achieve an optimal compromise between accuracy and energy consumption. We adapt the NN to the designed hardware by developing a training method where the reduction of precision is exactly modeled, and the network can be forced to only use efficiently implementable operations. Rather than just quantizing to an arbitrary number of states we look at characteristics of the used RRAM cells first. In addition to their non-volatile property, they can in general store multiple states. However, this requires additional hardware circuitry for reading and writing the cell states reliably, which results in a more power intensive circuit.

In our NAS, the energy required to read out the values stored in a RRAM cell is considered in the energy model (depending on the numbers of states). Therefore, the NAS can be performed for different quantization levels of the parameter values. However, we have found that a three-state configuration usually is best for minimal energy when using RRAM memory. The reason for this lies in the trade-off between parameter quantization and NN size. While fewer states of the NN parameters also mean less energy for storage and arithmetic operations, binary weights are not optimal. With binary weights only two operations (e.g., multiplication with -1 and $+1$ or multiplication with 0 and 1) can be realized and the accuracy of the NN might suffer. Ternary weights (i.e., -1, 0, $+1$) are beneficial, as layers can amplify, attenuate, or ignore certain features, which would not be possible with only binary weights [18]. This results in a small network topology that achieves a high classification accuracy. At the same time, the resulting operations can be realized in hardware very efficiently and all three states can be stored in a single RRAM cell. Other parameters of the NN, like *bias* values, can be stored into multiple (n) of these cells, resulting in 3^n possible states. Storing more than three states in a single RRAM cell is also possible but requires a more than linear increase in the required read and write energy.

Creating a quantized NN is not straightforward [11]. While inference of a NN using quantized operators can be realized without any implications, training an NN requires high precision calculations to find a minimum of a loss function. Using full precision weights and then quantizing them severely at the end of a training would result in significant accuracy degradation [24]. Therefore, we implemented a quantization aware training procedure that accounts for the quantization during training. In contrast to quantization after training, this achieves higher accuracy. Each training step consists of forward-propagation, backward-propagation, and weight update. The real-valued weights are quantized during the forward propagation which calculates the output and loss function. The gradients are then computed using the backward-propagation. Finally, the gradients are used to update the real-valued (not the quantized) weights. This way the real-valued weights can be updated slowly and eventually, and some of them might shift across the threshold and be quantized to a different value during the forward-propagation. This approach prevents the introduction of a bias in the gradient due to quantization [11]. At the end of the training, the quantized weights are used in our design. The real-valued weights are discarded, as they are only needed during the training process.

To make the final quantized NN more energy efficient, we have specified some further constraints, which we found to be beneficial for time series classification applications: We use a *Batch Normalization* layer after each *Convolutional* and *Fully-Connected* layer. This type of layer helps to extend the range of values of strongly quantized NNs by multiplying input values by a *scaling* factor and adding a *bias* value. Both parameters will be learned during training. To keep *Batch Normalization* energy efficient, we constrain the *scaling* factor to a power of two and only store the exponent. We call this type of layer *Shift-Based Batch Normalization* (SBBN). This enables the realization of the multiplication with a *scaling* factor using an energy efficient shift operation. The exponent as well as the *bias* value for the *Batch Normalization* layer are also quantized (e.g., to 3^n states). An example of a customized layer configuration extracted from our NN used for ECG classification is given in Figure 3. The weights of the *Convolutional* layer posess three states. The exponent of the *scaling* factor is quantized to 3^2 states ($n = 2$). For the *bias* value $n = 4$ is used. This enables a broad range of values to be used. On the other hand, we save up to $4\times$ and $2\times$ the amount of memory cells, respectively, compared to binary 8 bit values. Similar to training the weights, the *Batch Normalization* coefficients are quantized during the forward-propagation and the real-values are updated in the backward-propagation. However, two training runs are done to achieve the quantization of the *Batch Normalization*. During the first run the statistics for normalization (mean and variance) are collected. The second run continues with the model already trained in the first run. Before the second run, the collected statistics are fixed and quantized. These values are then used along with the trained coefficients to normalize the activations. This approach is necessary, as quantizing the mean and variance during collection is not possible because the accuracy hit introduced by this quantization would be too big to successfully train the network. In addition to the coefficients, we also quantize the activations and partial sums in the layers to save even more power for the arithmetic operations, data transfer and registers needed. For that, we use a saturated rectified linear unit (*ReLU N*) as the activation function. In contrast to the regular *ReLU* function, it not only cuts off values smaller than zero, but also cuts of values higher than a certain saturation limit (N). This limit depends on the quantization of the input values of the following NN layer.

Our quantization-aware training was implemented in the NN framework TensorFlow by extending already existing functionality. Based on the *Fully-Connected* (TF FC) and *Convolutional* (TC Conv) layer provided by the framework, new layer types have been created. We use the prefix 'quasi' for these layers as they implement the previously described quantization process during training. As defined by our constraints, each *Quasi Quantized Fully-Connected* (Quasi QFC) and *Quasi Quantized Convolutional* (Quasi QConv) is followed by a Quasi *SBBN* then a *saturated ReLU*. According to our tests, this order in combination with the strong quantization causes the least reduction in accuracy. Figure 4

shows exemplary for one layer the extension of the TF Conv, TensorFlow *Batch Normalization* (TF BN) and Tensorflow *ReLU* (TF ReLU) module to achieve the quantization-aware training. Here, Q_w, Q_{in}, Q_α and Q_β are the functions that quantize the the weights, inputs, the scale and bias of the *Batch Normalization*, respectively. The parameters W_R, α, β represent the real valued weights, scale and bias values that are to be quantized. These values are stored and updated during the training process, but only the quantized weights (W_Q), scale (α_Q) and bias (β_Q) values are used in the hardware implementation for inference. Finally, $Saturate(N)$ is a function implementing the saturation of the activations at a configurable level N, which is synonymous with the quantization of the activations. All arithmetic operations in the network are realized using fixed point value representation, resulting in smaller and more energy efficient hardware than a floating point implementation. Compared to generic NN accelerators, which use a fixed bit width for all operands, we can show that our individual quantization per layer can save a lot of energy using our energy model. Thus, compared to an accelerator using 16 bit operands, up to 90% of energy is saved for arithmetic operations and storage alone, without significant impact on the accuracy. In contrast to accelerators using 8 bit operands it is still as high as 60%.

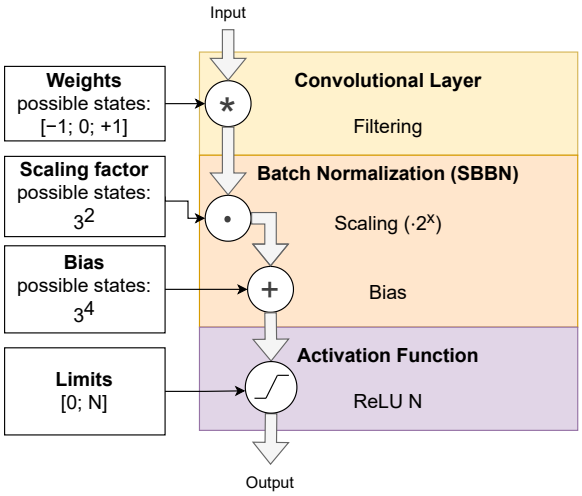

Figure 3. Example of a customized sequence of layers in our ECG classification NN.

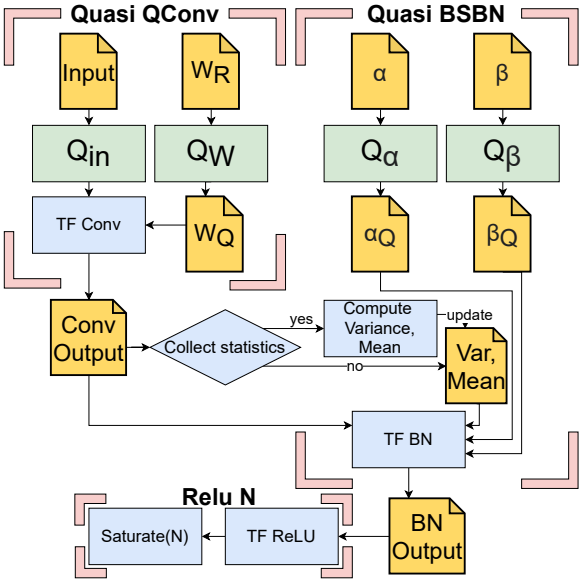

Figure 4. Overview of one Convolutional layer in our quantization aware training including SBBN and saturated ReLU.

4.2. Dataflow Driven Architecture

An essential part of the *HW mapping* step in the framework is the hardware architecture used to realize the calculation of the NN. While generic accelerators such as CPUs or GPUs can be used to run the calculations of a NN, they are far from optimal regarding energy consumption. They offer a high degree of flexibility, but require a lot of control overhead (i.e., instruction decoder, branch prediction unit etc.) which greatly contributes to the required energy. This accommodates different applications and flexible programming, but creates an additional energy budget for applications, where a fixed procedure is known beforehand. For NN-specific architectures this additional overhead can be omitted to reduce the energy consumption. Even when constraining to NNs and building specialized architectures like Eyeriss V2, a lot of energy is needed for the control logic [25]. Other approaches, like the TPU by Google, achieve a large reduction of the control logic, but still need large buffering structures, which occupy up to 29% of the chip area [26]. To solve this problem, a hardware architecture is required that is in complete contrast to freely programmable accelerators. Data paths should be defined at synthesis time and a data flow should be chosen which does not require buffering structures. For this, weight stationary architectures are already well researched and found to be a good approach for energy efficiency [5]. Especially when applying NNs with a small number of parameters, weight values can be kept in local storage and do not need to be reloaded. This saves memory bandwidth and energy. Therefore, we designed a completely new optimized hardware architecture based on the idea of systolic arrays, where part of the control logic is implicitly defined by the structure of the hardware architecture. It is derived from our optimization strategies for time series classification tasks discussed in Section 3 and designed to be generic and energy efficient. Different *hardware components* based on this hardware architecture are implemented and can be used with a hardware generation tool to create a complete hardware design. One of the unique features of our architecture is that it internally works completely without control logic and buffers for the calculation of the NN. The data flow is given by the connection of the components and is derived from the design of the NN during hardware generation.

Nevertheless, the goal of our architectural concept is furthermore to enable the use for different applications with different NN designs. It should be possible to use the *HW generation and parametrization* tool from our design flow to create and parametrize an optimal hardware depending on the NN configuration. Therefore, all hardware blocks designed are flexible regarding their (automated) mapping and arrangement. Figure 5 shows an overview of the hardware architecture of a *Convolutional* layer. As the share of the static power of the Processing Core design is small compared to the dynamic power and can be optimized by reducing the on-time, the hardware is designed to minimize dynamic power. Therefore, the data arrives at each layer in a systolic manner. This means that the first two input channels deliver values in parallel. The following channels deliver values with an offset of one clock cycle each. The layer itself consists of several multichannel (MC) filters, which correspond to the number of filters inside the layer defined by the NN. Each MC filter consists of multiple single channel (SC) filters, corresponding to the amount of input channels that the filter has. The SC filter is composed of a regular structure of multiple processing elements (PEs). Each PE has its own weight memory in form of a RRAM cell, which is physically not inside the PE, but in a RRAM block close to the actual PE. However, its output is routed directly into the PE. The input values are passed from one PE to another and a multiplication with the weight is carried out. Each arithmetic operation is enabled by a simple counter-based controller, depending on the stride value defined for the filter in the NN. The output values of each PE are summed using an adder tree. Each SC filter possesses a simple 1D topology of PEs. The output of multiple of these 1D structures is summed up inside the MC filters, which are themselves grouped to a regular 2D structure inside one layer. In our design, filter coefficients that are moved over the input signal are directly accounted for and do not need to be stored multiple times. This design also significantly reduces dynamic energy, as the stride value is realized using clock gating

and no unused partial sums are calculated. The dataflow between the layers is optimized in a way that multiple layers can be directly concatenated without the need for buffering in between.

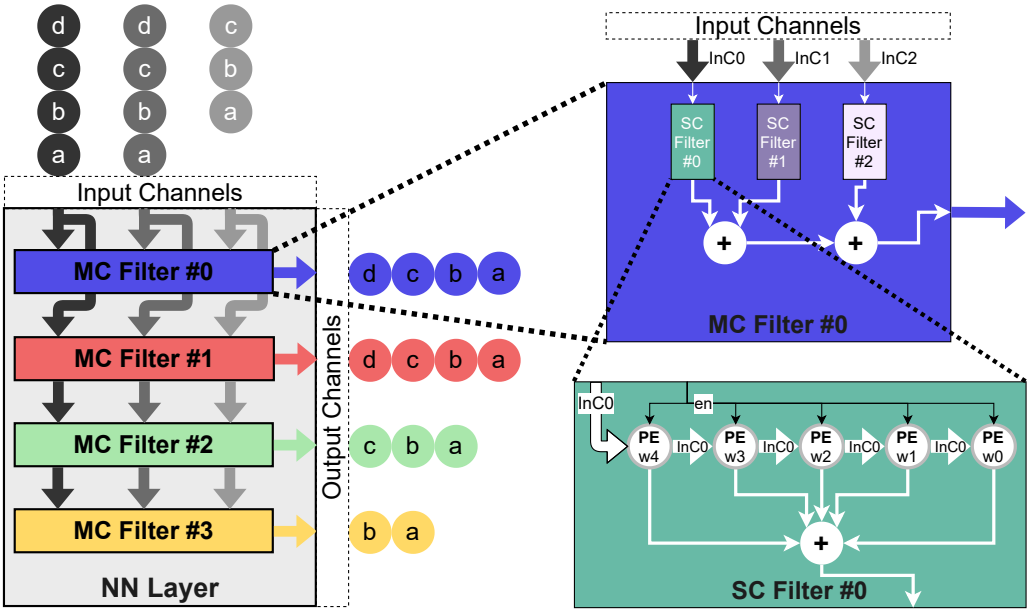

Figure 5. Hardware architecture of one *Convolutional*/*FC* layer including MC filters and SC filters.

The architecture of the *Fully-Connected* (*FC*) layer is the same as the *Convolutional* layer. The length of the filter is mapped to the amount of input values as well as the stride value. This results in a very low duty cycle. However, it was tested to be more energy efficient than mapping multiple weights into one PE. The reason is a much lower dynamic power requirement caused by less toggling, while the higher static power requirement for inactive PEs has an insignificant overall impact. Other hardware modules, like the *ReLU* activation function, the *scaling* and the *bias* addition of the *BatchNorm* layer as well as the *Max Pooling* layer are built of a vector of PEs. Each PE operates on its own channel. One PE in a *scaling* layer shifts the input value by the amount defined in the NN. Each PE in a *bias* addition layer adds the trained *bias* value to all inputs of one channel. The overall dataflow is delayed by one clock cycle by all of these modules, keeping the dataflow intact and enabling a concatenation of all layers in any order.

Implementing an architecture, which calculates all outputs of a layer fully parallel is not possible when designing a chip that should be integrable into a wearable device or a similar embedded environment. Even with a small network, a fully parallel configuration, extrapolated from the components of our design, reaches a chip size of over 500 mm^2. This is not practical for embedded use cases, therefore, our architecture does not calculate each individual output value in parallel, but one value of each channel in parallel. This is a good compromise to achieve a realistic chip size of a few multiples of 10 mm^2, depending on the NN size. It needs more clock cycles to calculate a classification, but the individual PEs are utilized with a high duty cycle, depending on the layer and the stride used. In comparison to a time multiplexed architecture, where multiple operations are mapped to the same PE, our design achieves an overall reduction of the average power. Mapping multiple operations to the same PE results in frequent changes of the operands in the PE and therefore a higher dynamic power. We implemented such a design for exemplary FC layers and compared the energy after synthesis. We could show that even in these FC layers, where the PEs possess only a low duty cycle, mapping multiple operations to the same PE and switching the weight values requires more energy than not using some PEs for multiple cycles.

4.3. Non-Volatile on-Chip RRAM

Another important part of the *hardware components* designed by us is the embedded RRAM memory. For a classification task in an embedded device, it is usually necessary to collect multiple seconds of data sampled at a low rate, while no actual classification must be carried out. In our architecture the classification is the task of the Processing Core, which could be in a sleep state during this data acquisition process. However, in traditional architectures, mainly volatile memory is used to store the NN parameters (e.g., NN weights, scaling factors) locally in accelerators. This means after every power cycle of the chip these parameters must be reloaded from external non-volatile storage. Reloading data from external memory requires a huge amount of energy. This contradicts the goal of reducing energy and prevents an efficient design with a power-down mode. To overcome this problem, we use non-volatile memory in form of RRAMs to store data in an embedded way. This RRAM cells can be read in parallel and allow for a quick availability of all parameters after power is (re)supplied. The division into two power domains, as seen in Figure 1, enables us to put the Processing Core into a deep sleep mode where it draws almost no power and after waking up the chip, all the parameters of the NN are still present.

The embedded memory blocks for local storage demand efficient read and write circuitry to be integrated in the architecture. The research on the integration of RRAM cells is very recent and due to the manufacturing process, they often possess considerable variations per cell. These can have an influence on the interpretation of the values stored in them [27]. The RRAM design used in our system is provided by IHP as part of the MEMRES module, integrated into their 130 nm SG13S-process line [27]. Previous research has been done on the device characteristics of these cells in IHP technology and their system level influence on NN accelerators. The typical device variations have been considered and it has been shown that the accuracy of the NN does not decrease by more than 1% despite the inaccuracy introduced by these variations [28].

In our architecture the memory cells are grouped into blocks of 32 and all blocks are connected like a long shift register, as shown in Figure 1. At the same time, the values of all cells are available to the accelerator in parallel. As discussed in Section 4.1, three states seem like an optimal solution for most NN configurations. Therefore, we consider three resistive states of the RRAM cells (two low resistance states (LRS1 & LRS2) and one high resistance state (HRS)). However, the use of more states is also possible. The design of this RRAM memory block has been optimized specifically for use in low-power accelerators for NNs [29].

For reading and programming the RRAM cells, an analog circuit block was designed that is used for each memory block. A diagram of this memory block and the surrounding circuitry is shown in Figure 6. It contains 32 1T1R RRAM cells, consisting of one transistor and one resistive switching element (hence the name 1T1R), each equipped with a read circuit, so all cells can be read simultaneously. The red digital control signals are provided by the RRAM manager (shown in Figure 1), while the green signals are analog signals used inside the memory block for interactions with the 1T1R cells. Each interaction with the RRAM cells is based on analog voltage pulses of specific height, in Figure 6 indicated as "V_{pulse}", which are buffered by an operational amplifier, that can provide sufficient output current for reading and programming the cells in parallel. The read method is based on a voltage divider and is done with low voltage pulses in order to prevent unintended programming of the cells during a read operation. For programming, voltage levels above the 1.2 V supply voltage (VDD1V2) of the digital core are necessary, therefore, the analog circuitry is also provided with an additional 3.3 V supply rail (VDD3V3). A reference block provides the necessary analog voltage levels for all operations. Besides the previous mentioned pulse height of the pulses, each operation (reading, writing LRS1, LRS2 or HRS) needs specific voltage levels at the selection transistor of the 1T1R cell, in Figure 6 shown as "V_{WL}". "V_{ref}", which is used by the comparators to determine the state of the RRAM cells during read operations, is also provided by the reference block. The external operation bits provided by the RRAM manager are then processed by an internal control logic which

applies the necessary signals to apply the voltage levels from the reference block that are needed for the corresponding operation. Furthermore, the "cell_sel" bits determine which cells are to be read from or written to, respectively. The power supply control is triggered by the "pwr_en" signal, which is also provided by the RRAM manager and can detatch the block from the supply rails, therefore saving energy. Compared to the overall design, the read-write circuitry of the RRAM cells possesses a high static power consumption and reading the cell state multiple times costs additional energy. Therefore, in our design the data from the RRAM cells ("comp_out") is read into latches once after each power up phase of the Processing Core. After reading from the non-volatile cells, the RRAM block is put to sleep again (controlled by the "pwr_en"-signal), thereby preventing the relatively high power consumption of the analog circuitry compared to the static dissipation of the latches. The output values of the latches are routed into the PEs of our design by providing the saved parameters for the NN. Our design was created using Cadence Virtuoso with the IHP SG13S process PDK and its functionality was verified by simulation [30].

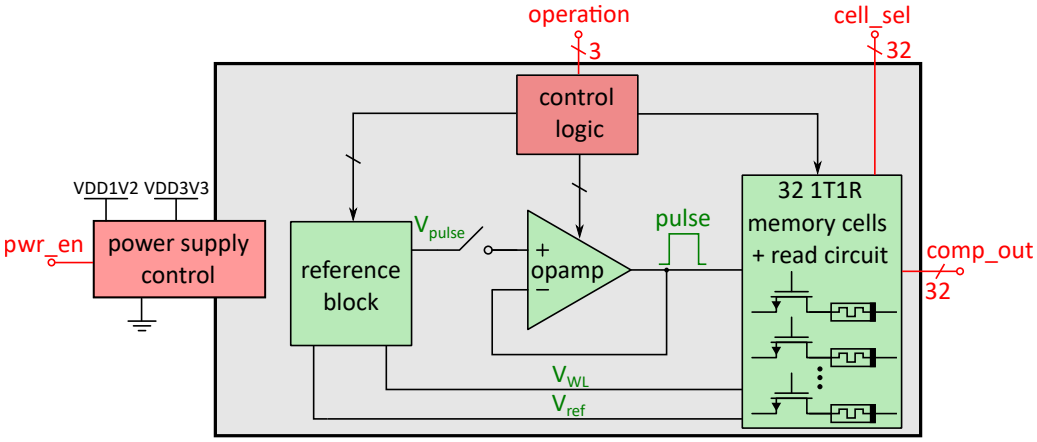

Figure 6. Block diagram of an RRAM memory block containing 32 1T1R memory cells and the necessary peripheral circuitry for reading and writing the cells [29].

5. Further Optimizations

To implement our approach and create a system for a time series classification application, a designer can use our framework and perform the design steps described in Section 4. On top of these steps, some additional tweaks can be done to save even more energy using domain- specific knowledge. In this section, we will point out possible further optimizations as an example of the ECG classification application in a wearable device. We will also state some ideas about how these optimizations can be transferred to other embedded time series applications.

5.1. Data Preprocessing

When minimizing the power consumption of a classification application, it can be important to preprocess the raw data after data acquisition. Thereby, the amount of data that the NN must process can be limited, and unwanted frequency components can be removed from the raw data. However, when implementing preprocessing of the data in hardware, it must be optimized to make sure its benefits, in terms of reducing the data rate and improving the classification accuracy with regard to outweiging the energy costs. We have not shown the preprocessing as part of our design flow in Figure 2, even though it is included in our architectural concept (see Figure 1). This is due to preprocessing usually requiring domain specific knowledge per classification application, (e.g., which frequency bands of the input data are relevant). For some classification applications using NNs it might not be required at all, depending on the data acquisition. For the example of the ECG classification application, we will show approaches with regard to how to derive very energy efficient preprocessing which is adapted to the NN.

One aspect of the optimization of the preprocessing is the data rate at the input of the NN. For our ECG classification application, a state-of-the-art external sensor is used, which samples the raw data at 512 Hz with a resolution of 12 bit. To minimize this data rate, quantization on the raw input data can be applied. For ECG data we have tested different NNs during the *NN design* phase of our design flow and have shown that quantizing to a resolution of 8 bit is sufficient for the raw input data without significantly affecting the classification accuracy. Another option for reducing the data rate is to lower the sampling rate by downsampling. Our research showed that when using a NN, reducing the sampling rate down to 128 Hz does not measurably reduce classification accuracy. Therefore, a downsampling by four can be applied. For lower energy consumption, the raw ECG data can be reduced even further. ECG data is usually sampled from two electrodes in two channels to achieve a high signal-to-noise-ratio (SNR). When using only a single channel, energy can be saved in the preprocessing, buffering and in the first layer of the NN. To reach the same accuracy using only a single ECG channel, a slightly larger NN is required. However, reducing the amount of data by omitting one channel saves more energy than the larger NN needs. As an additional benefit, measuring a single channel ECG signal in a wearable device can lead to an easier design of the device. In summary, we can see that only by applying the quantization (from 12 bit to 8 bit), downsampling (by 4) and use of a single channel (instead of 2) can a data compression of a factor of 12 be achieved. This does not only reduce the energy consumption of the preprocessing itself, but also of the NN processing, while adding almost no hardware overhead and therefore only a negligible additional energy consumption.

This is not all that can be done to optimize energy consumption. For ECG classification not all frequency components contained in the raw data are relevant. The most important frequency components are to be expected in a frequency band up to 50 Hz [31]. To reduce the dynamic energy introduced by unneeded frequencies and to combat aliasing effects introduced by downsampling, the raw data can be filtered by a bandpass filter. Nonlinear algorithms, such as NNs, may not be negatively affected by these effects and may even extract information from them. So instead of using a hand-designed filter, we came up with a solution that is also integrated into our framework: during the *NN design* (as described in Section 4) we add an additional filter without any non-linearity at the beginning of the NN as the first pseudo layer. During training, the parameters of this filter will be learned by the NN. The result is a preprocessing filter design with a frequency characteristic that does not remove information relevant to the NN from the input data. To minimize the energy of this filter we evaluate the frequency characteristic and map it to an efficiently implementable Cascaded Integrator Comb (CIC) filter [32]. This enables maintaining most of the learned frequency characteristic, while at the same time reducing the energy consumption of the hardware implementation of the filter. The frequency response of the trained and the mapped filter are shown in Figures 7 and 8. The block diagram for the efficient implementation of the CIC filter is shown in Figure 9. For a hardware realization, only a few arithmetic operations and registers are needed, while the dynamic energy needed in the NN can be reduced. For other classification use cases the same strategy for a preprocessing filter design can be applied, and the frequency characteristics can be mapped to similar optimized filter designs. In some cases, filtering and downsampling might not be needed. In this case, data preprocessing can be completely bypassed. Therefore, our architectural concept (see Section 3) is flexible regarding the preprocessing to address the requirements of different applications.

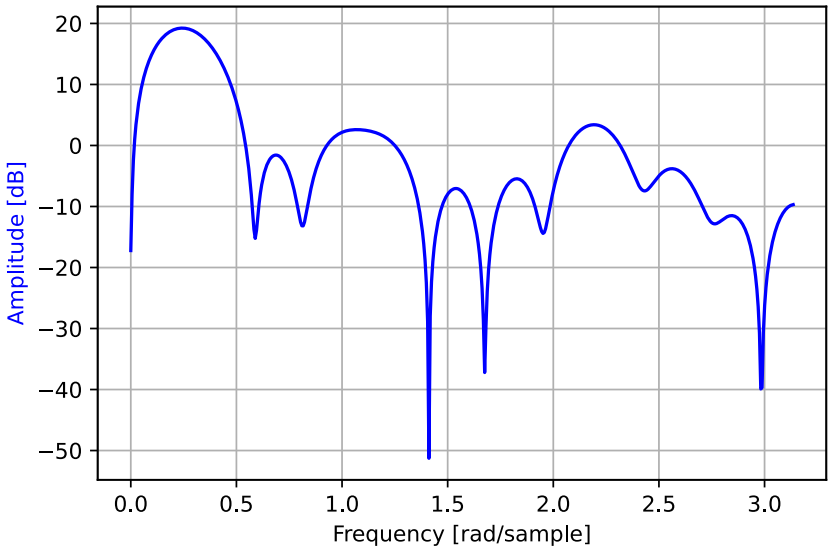

Figure 7. Frequency response of the trained preprocessing filter.

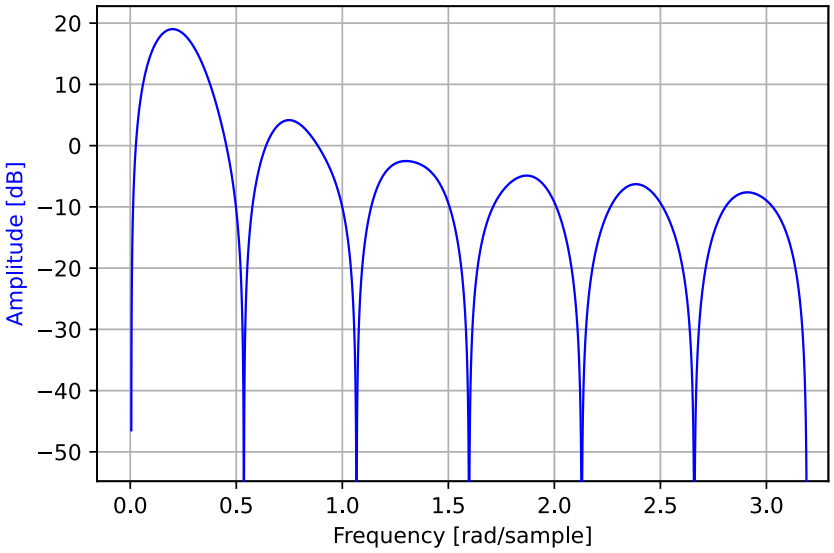

Figure 8. Frequency response of the the approximated CIC preprocessing filter.

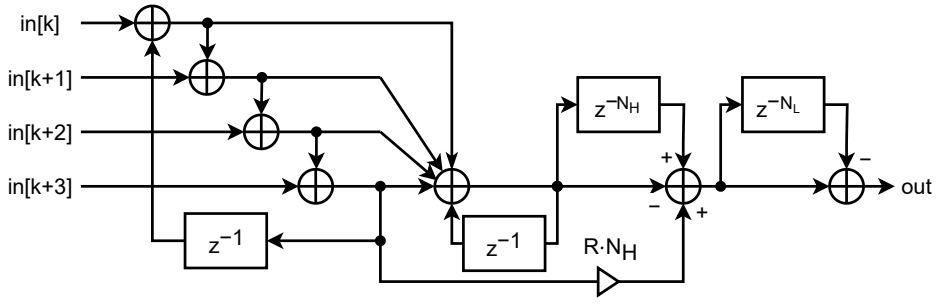

Figure 9. Block diagram of the CIC filter used for preprocessing the data (using $N_H = 4$ and $N_L = 3$).

5.2. Optimized Dataflow

In an embedded classification application, data has to be sampled in real time. Common ECG devices used in medical facilities use a frequency of 512 Hz for this purpose, which corresponds to a significant oversampling. Other applications might use similar low sampling rates. However, for classification the data must be grouped into sections of a certain length (T_{samp}). These so-called windows must be buffered until all the data

has arrived. Since data needs to be collected continuously, even when one window is processed, a double buffering is applied. A tradeoff for the length of the input window must be found, since a larger window benefits the classification accuracy, and a smaller window benefits the energy needed for buffering. In general, when choosing a very short length of only a few seconds, the accuracy (sensitivity and specificity) will decrease heavily. With increasing window length, the quality of the classification will improve. Depending on the application, the improvement will hit the point of diminishing returns at some point after a certain window length. This effect can be seen in Table 1 at the example of our NN for ECG classification. For each entry of the table a model was trained from scratch independently for each window length to avoid any bias. As the minimization of the loss function during training can result in local minima, these accuracy values are subject to change for each training run and serve as approximate values to demonstrate the effect of the window length.

Table 1. Accuracy Results (Measured Using Different Window Lengths).

Window Size	5115	5797	6479	7161	7843
Sensetivity	90%	91%	**96%**	94%	96%
Specificity	85%	82%	**89%**	87%	90%

When increasing the window length, more hardware resources for buffering the data and energy for preprocessing as well as processing the first layer of the NN is needed. This window length is part of the NAS described in Section 4.1. For our ECG application, the size of the input window was found to be 6479 samples (12.65 s). Therefore, we group and classify the incoming data in windows with a sampling duration $T_{samp} = 12.65$ s.

An overview of the dataflow at relevant points of our design is given in Figure 10. The dataflow at different points of the architectural concept as seen in Figure 1 are shown. The top of the figure shows the data at the input of the Data Control Core and the windowing scheme that is applied. As seen in Figure 1, double buffering is used to separate the preprocessing from the Processing Core. Incoming windows after preprocessing will be written into the two buffers in an alternating fashion. This can be seen in the middle section of Figure 10. From the buffers the data is read in the same alternating fashion and will be processed by the NN.

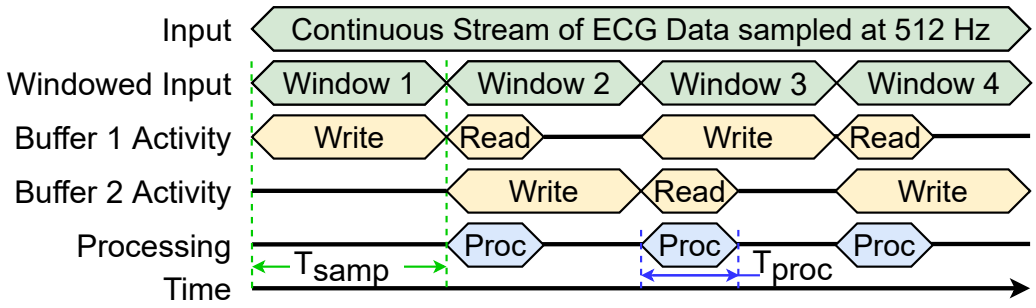

Figure 10. Overview of the optimized dataflow.

The long sampling duration causes the static power of both of our cores to dominate the dynamic power by more than one order of magnitude. As the classification of one window is significantly faster than the sampling duration ($T_{proc} << T_{samp}$), we turn off the Processing Core for most of the time to reduce the overall static power. The dynamic power for a given CMOS circuit is mostly dependent on the executed operation. The frequency of the Data Control Core is defined by the sampling rate of the input data. The frequency of the Processing Core, on the other hand, can be increased to reduce the power-on-ratio of this core and minimize the average static power. However, a higher clock frequency

results in more toggling in the clock tree. As the clock for the Processing Core has to be routed through the Data Control Core to synchronize the memory access to the double buffer, the dynamic power of the clock tree increases with a higher clock frequency, even when the Processing Core is turned off. An optimized clock frequency of the Processing Core can be found after generating the final design, using common EDA tools.

6. Evaluation

In this section we give an overview of the results we achieved when applying our architectural concept to the ECG classification task in a wearable device. Our framework discussed in Section 4 was used to generate an ASIC implementation for the task and the optimizations discussed in Section 5 were also realized. We evaluate the energy consumption of this implementation and demonstrate the savings that our architectural concept introduces compared to state-of-the-art approaches. At the end we also present a comparison of our framework with those from the literature.

6.1. Training Data

To train the NN for AFib detection a dataset was used that consists of 16,000 ECG recordings of two minutes in length. The data was sampled on two channels with 512 Hz. One half of the recordings contains AFib, and the other half contains a healthy heart rhythm. As in the final NN, smaller window sizes for the input data were used, the two minute recordings were split into multiple smaller segments. All the segments were used during training, and therefore an augmentation of the data was achieved. The segments were partitioned into training (70%), validation (15%), and testing (15%) datasets. The training dataset was used for training of each candidate model during *NN design* (see Section 4.1). The validation dataset was then used to calculate the sensitivity and specificity metrics used during the NAS described in Section 4.1. Finally, the testing dataset was reserved unseen by the training and NAS. This unseen dataset was used to report the sensitivity and specificity metric of the final model to ensure unbiased results.

6.2. Final NN Model

For our ECG classification task, a minimum sensitivity of 95% and a minimum specificity of 85% were used as target values. This makes the resulting system accuracy comparable to the accuracy reached by state-of-the-art non NN-based classification devices [33]. Setting higher thresholds for sensitivity and specificity is possible at the expense of energy consumption. However, since several thousand classifications are done per day, some rare misclassifications do not matter much. A graphical overview of the resulting model including the NN layers and the dimensions of their operators as well as the result values can be seen in Figure 11. The number of parameters (i.e., weights, scaling factors and bias values) as well as the quantization chosen for these parameters and the result values for the final model are shown for each layer in Table 2. This model achieves a sensitivity of 97% and a specificity of 87% on the testing dataset and therefore exceeds our minimum threshold. Overall, only 1370 parameter values with different quantization are needed, which results in 1490 ternary values that need to be stored. In total we only need to incorporate 47 memory blocks containing RRAM cells (each containing 32 RRAM cells) into our Processing Core. All parameters are fully reprogrammable, enabling a retraining of the network at any given time. After the ASIC implementation, all hyperparameters such as the number of parameters per layer or the number of layers is fixed, but a different configuration can easily be derived using our framework, as described in the previous sections.

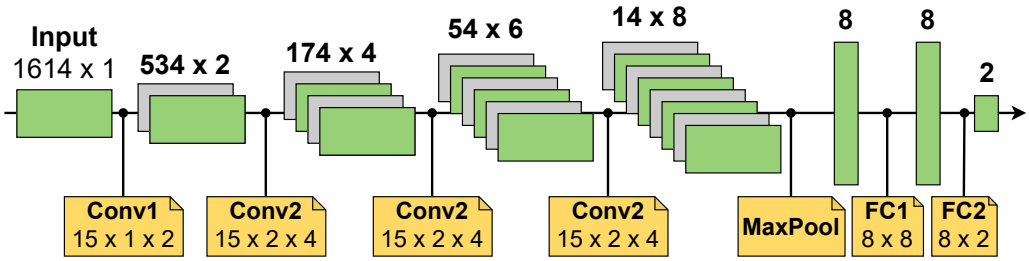

Figure 11. Overview of the final neural network model.

Table 2. Parameters of the Final Neural Network Model.

Type	No. Parameters	Parameter Quant. (States)	Result Quant. (Bit)
Conv	30	3	11
BatchNorm	2, 2	3^2, 3^4	19
ReLU	0	-	4
Conv	120	3	10
BatchNorm	4, 4	3^2, 3^4	18
ReLU	0	-	4
Conv	360	3	11
BatchNorm	6, 6	3^2, 3^4	19
ReLU	0	-	4
Conv	720	3	12
BatchNorm	8, 8	3^2, 3^4	20
MaxPool	0	-	20
ReLU	0	-	4
FC	64	3	8
BatchNorm	8, 8	3^2, 3^4	6
ReLU	0	-	4
FC	16	3	8
BatchNorm	2, 2	3^2, 3^4	16
ReLU	0	-	4
Overall	**No. Param.**	**Ternary Values**	**Memory blocks containing RRAM cells**
	1370	1490	47

6.3. ASIC Implementation

Following our architectural concept shown in Figure 1, the chip for ECG classification has been divided into two main parts, which have been synthesized independently and implemented in the IHP 130 nm standard cell library. Figures 12 and 13 show the layout of the implemented Data Control Core and Processing Core after place and route (with different scaling). The area of the Data Control Core of 363 × 950 µm^2 is mainly dominated by its SRAM blocks, which are necessary for double buffering and require an area of approx. 332 × 377 µm^2 each. In contrast, the memory blocks containing RRAM cells are controlled for programming and external testing by a digital circuit that is assigned to each block, which is also synthesized the same way. The combination of analog and digital subsystem requires an area of 155 × 960 µm^2, whereby the focus of the optimization of the blocks was exclusively on their reusability. These memory blocks containing RRAM cells were placed within the Processing Core 47 times in total and thus consume approx. 60% of its total area of 4870 × 2396 µm^2. The relatively high percentage for the memory blocks is a result of the analog trade-off that minimizes the active time and enables parallel processing at the expanse of the chip area. The processing logic of the NN was placed in between the memory blocks containing RRAM cells to minimize space and have a short route between the PEs and the RRAM cells. The value for the area of the Data Control Core lacks the

IO-pads as the chip has not been produced yet. On the other hand, we estimate that the area of the Processing Core can be reduced by half when optimizing for area. The size of the analog read and write circuit of the RRAMs. for example, can still be further minimized and thereby enable a tighter placement of the memory blocks containing RRAM cells. However, this was out of scope for the current state and will be done before manufacturing a chip. Nevertheless, it can already be seen that the area is small enough, so our design can realistically be used for an energy efficient wearable application. Compared to accelerators using classical memory cells, such as SRAM, our approach has the benefit that weights and other parameters are stored locally, near the point where they are needed. In theory, the RRAM cells also store more information in the same area. On the one hand this follows from the fact that, compared to a binary memory cell, more information (multiple levels) can be stored per cell, while on the other hand the cell itself possesses a significantly smaller area than SRAM, for example [34].

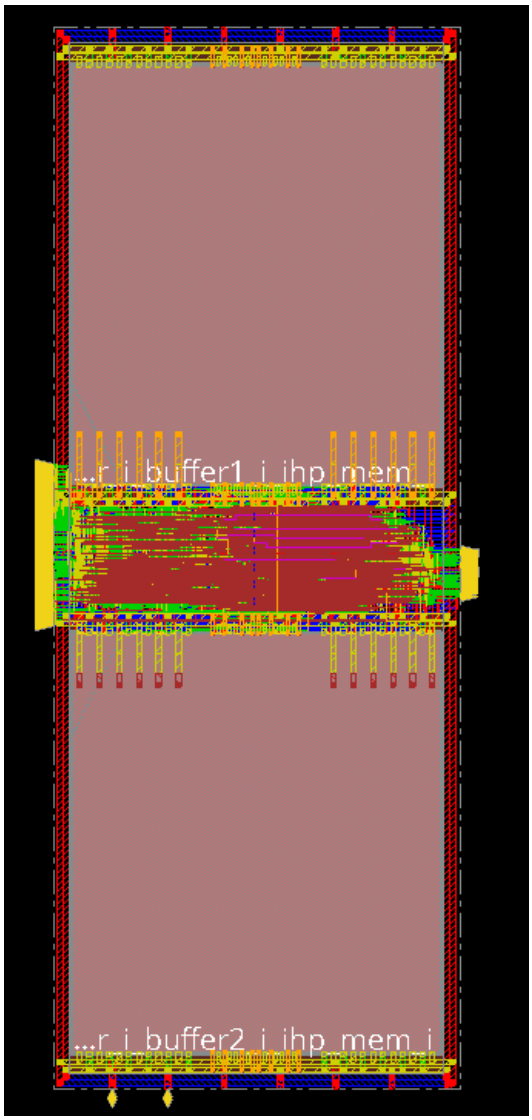

Figure 12. The layout of the Data Control Core with an area of 363 × 950 µm^2, dominated by the SRAM double buffer with the control circuits in between.

6.4. Energy Evaluation

To demonstrate the energy savings introduced by our framework, comparisons at different stages of our design flow have been conducted. The evaluation of the effect of the optimizations applied during *NN design* is done using the energy model (see Section 4.1).

We could show by using hard quantization alone to three state weights ([−1, 0, +1]) and 3^n state bias values and scaling factors as well as adapted quantization of the partial sums and activations per layer, energy savings as high as 60% can be reached. Integrated into the NAS proposed by us, another 70% can be saved, depending on how optimal the initial NN already is regarding the optimization goals. Combined with the RRAM cells which possess multiple possible states, less memory cells are needed to store the weights and other NN parameters, compared to an accelerator with binary storage cells. Therefore, it can be seen that the goal of vast energy savings by avoiding external memory access and applying heavy quantization was reached, due to the efficient use of local, on-chip RRAM storage and our NAS.

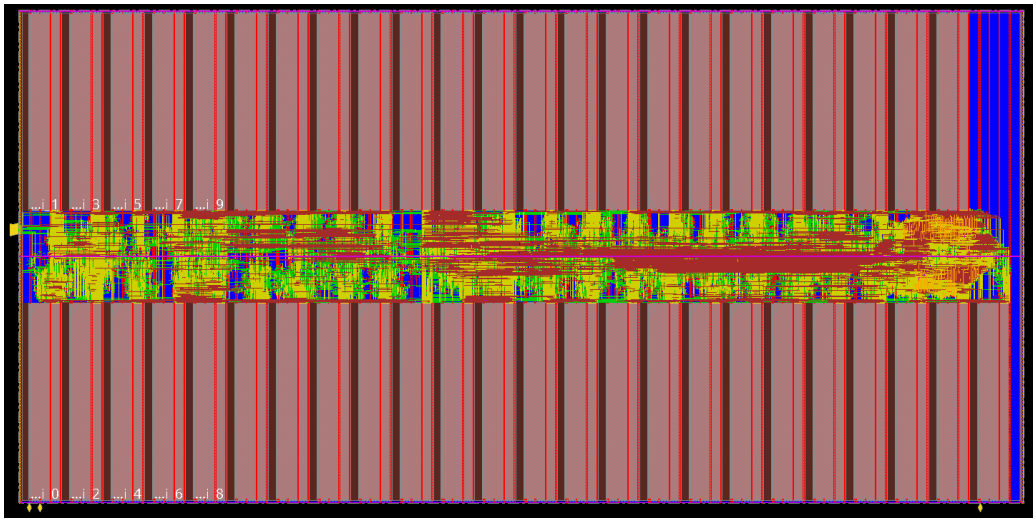

Figure 13. The layout of the Processing Core with the 47 memory blocks containing RRAM cells and the hardware for the NN in between. It takes up an area of 4870 × 2396 µm^2.

For the *hardware mapping* process, no direct energy evaluation can be made. However, in comparison to a low-power general purpose CPU, for example, based on the RISC-V architecture, our dataflow driven architecture has almost no control overhead, which can account for up to 95% of the energy consumption of such a CPU [35]. Minimization of control flow and buffering leads to a massive reduction of the energy consumption, even when compared to other devices suitable for an embedded application.

After the final step in our framework, (the *hardware implementation*) an estimation of the energy consumption of the digital components has been generated using the tools Synopsys PrimeTime and Synopsys PrimePower. For this estimation, the final post place-and-route netlist and the power models provided by the standard cell library in combination with toggle data extracted from several real simulation runs have been used. A simulation of the analog circuit block and the RRAM cells has been conducted separately using a RRAM model adapted for the IHP cells based on the Stanford-PKU Model and SPICE-based analog simulation tools [36]. The resulting energy for classifying a window of 12.65 s of ECG data as well as the average power consumption of our design is given in Table 3. The Data Control Core is running with a clock frequency of 512 Hz and the Processing Core is running at 70 kHz.

The energy of the Processing Core is minimized by the efficient dataflow driven architecture and all the optimizations included in our framework. The dynamic energy specifically is also reduced by the preprocessing and removal of unwanted frequencies in the raw data. Therefore, at this point, the biggest amount of energy can be saved by minimizing the on-time of the Processing Core and thereby the static energy. To demonstrate the impact of such an optimization, a comparison between an always-on operation and a power-down operation is shown in Table 3. For the power-down operation the parameter values must be provided by the RRAM cells after every power cycle. However, as the

energy needed for loading the data from the RRAM cells only makes up about 0.4% of the overall energy, this influence is negligible. In the always-on case the parameters need to be loaded only once. As the influence of a single loading of the parameters on the average energy is marginal and decreases with longer runtime, it has been ignored here. It can be seen that our power-down approach results in savings of 99.8% of the energy of the Processing Core. This is significant, considering that this effect is on top of the other optimizations. It is a direct effect of the on-time of this core being only 0.2% of T_{samp} and the drastic reduction of static energy. The average overall energy needed per classification of one window is 12.4 mJ. In wearable devices, such as a current generation smartwatch powered by a small battery (which can provide around 1.3 Wh of energy), this corresponds to an extremely long duration of almost two months and almost 400,000 classification results before the battery would need to be recharged.

Table 3. Energy Evaluation per ECG Window of 12.65 s.

	Always-On	**Power-Down**
Data Control Core	11.75 mJ	
Processing Core	220.5 mJ	0.55 mJ
Memory blocks containing RRAM cells (analog)	1.24 mJ	0.055 mJ
Overall	233.5 mJ	12.4 mJ
Average Power	18.46 mW	0.98 mW

6.5. Framework Evaluation

A direct comparison of our framework with other approaches from the literature is difficult since no one has yet integrated all system levels, as we have in our approach. In addition, the results are often not directly reproducible due to unpredictable outcomes of the optimization processes. This problem already concerns the subtopic of NAS and solutions to make different NAS approaches comparable have been described by Li et al. [37]. However, the consideration of the complete system view is also missing here. Nevertheless, we were able to make a qualitative comparison of our framework with other approaches found in the literature, which is given in Table 4. The table shows where these approaches fall short, and more research toward a complete system needs to be done. It can be seen, for example, that often only partial optimization goals are used, such as minimizing the size (i.e., number of parameter bits) of the NN. However, this does not necessarily lead to a minimization of the energy of the entire system if, for example, this results in more arithmetic operations having to be performed. Other approaches simply map the NN to the hardware without considering it during the NN design. Due to the unique combination of features of our framework, it is clear that a quantitative comparison is not really possible at this point. The other approaches would need to be extended to also cover an end-to-end design of an embedded system using NNs for classification. Only then could the results from the frameworks be compared. Since the optimizations of our framework are directly intertwined with a global optimization goal, an isolation of individual steps is not possible.

Table 4. Comparison of our framework with different approaches from the literature.

	Target HW Known	NN Design Using HW Metrics	Hardware Aware Training	Support for NVM and Power-Down	End-to-End System Design	NN Optimization Goal	Applied Optimizations
Our	Yes	Yes	Yes	Yes	Yes	Accuracy HW Energy	NAS Hard Quantization Energy Efficient Operations
[11]	No	No	Yes	No	No	Accuracy NN Size	Pruning Hard Quantization
[12]	Yes	No	Yes	No	No	Accuracy NN Size	Quantization
[38]	No	No	No	No	No	Accuracy NN Size	NAS
[39]	No	No	No	No	No	Accuracy NN Operations	NAS
[40]	No	No	No	No	No	Accuracy NN Operations	NAS
[41]	Yes	No	No	No	No	None	None
[42]	Yes	No	No	No	No	None	None

7. Conclusions and Future Work

In this work, we created a framework that helps to design an ultra low-power system for embedded time series classification applications using NNs. We have developed an energy efficient hardware architecture for this problem class which has a very low overhead for the control flow. To allow an easy end-to-end design of a system, we have incorporated this hardware architecture into a framework. This framework optimizes the configuration of the NN in terms of power consumption and accuracy using a NAS. A designer can parameterize how these two optimization goals should be weighted. A hardware generator extracts the data flow and the interconnection of the hardware components from the resulting NN configuration. We have shown that while many optimizations for NNs and their hardware implementations exist, essential considerations of an efficient overall system are often missing. With our framework a system designer is provided with an easy-to-use tool that generates very efficient hardware without the need to apply additional optimizations.

To validate the claims in practice, we evaluated the application at detecting AFib in ECG data using a NN. We could show that 95% of the energy consumption, compared to an unoptimized, always-on solution, can be saved with the help of our framework. To compare the energy savings provided by our framework with other approaches, we participated with the ASIC implementation of the AFib detection application at a national challenge between leading universities and research institutes in neuromorphic computing. Our hardware design won the first prize for energy efficiency [43]. However, the evaluation of this competition focused only on the energy required to classify an ECG signal of 2 min in length. The energy saving due to the power-down mode in a real wearable application (up to 95%) was not considered. This shows that the joint optimization of software and hardware, which has been incorporated into our framework, leads to a massive reduction in the energy consumption of the final system.

Nevertheless, further optimizations of our framework can be achieved. One point is to optimize the implemented hardware components with regard to their area, which was not the focus of this work. Minimizing the power required for reading and writing the RRAM cells is also still the subject of research and could allow for more power efficient data access in the future. The tools that make up our framework can be extended or included in a broader context to cover more NN related problems and NN designs. For example, further hardware architectures for other classification tasks could be included. The implemented NAS as well as the underlying power estimation model can also be extended further.

The more precise the different hardware components are modeled, the more accurate the results will be. This will lead to finding more efficiently implementable NN configurations. Other emerging non-volatile memory elements can also be integrated into the framework and targeted with the NAS. On-device training is an additional extension option. This is especially useful for applications where the NN must learn new information during runtime (e.g., reinforcement learning). For the implementation of on-device training, however, an extension of the hardware architecture by the appropriate components is necessary (e.g., memory and interface for training data, computation units for gradient calculation). Likewise, the extension of the tool flow of our framework is required to generate the components for the training hardware and to transfer the data required for training to the target system. Another important step for future work is to develop standardized methods to compare such end-to-end frameworks in a meaningful way, since a baseline must be created in order to create quantitative statements about the quality of such a framework.

Author Contributions: Conceptualization, D.R., S.P., B.M., P.R., D.F. and M.R.; methodology, D.R., M.M., S.P. and P.R.; software, D.R., M.M. and M.O.; validation, S.P. and P.R.; formal analysis, D.R., M.M. and M.O.; investigation, D.R., M.O., M.M. and S.P.; resources, D.F., A.H., M.B. and M.R.; data curation, D.R., M.M., M.O., S.P. and P.R.; writing—original draft preparation, D.R.; writing—review and editing, all authors; visualization, D.R., S.P. and M.M.; supervision, M.B., A.H., D.F. and M.R.; project administration, M.B. and M.R.; funding acquisition, S.P., M.B., A.H., D.F. and M.R. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded as part of the challenge for disruptive innovations in energyefficient AI hardware initiated by the German Federal Ministry of Education and Research (BMBF) under grant number 16ES1142K and 16ES1143.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Simulation and measurement data is available on request from the corresponding author.

Conflicts of Interest: The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

1T1R	One Transistor and One Resistor
AFib	Atrial Fibrillation
CMOS	Complementary Metal-Oxide-Semiconductor
EDA	Electronic Design Automation
ECG	Electrocardiogram
HDL	Hardware Description Language
HRS	High Resistive State
LRS	Low Resistive State
NAS	Neural Architecture Search
NN	Neural Network
ONNX	Open Neural Network Exchange
RRAM	Resistive Random Access Memory

References

1. Nattel, S. New ideas about atrial fibrillation 50 years on. *Nature* **2002**, *415*, 219–226. [CrossRef]
2. Jenkins, J.M.; Caswell, S.A. Detection algorithms in implantable cardioverter defibrillators. *Proc. IEEE* **1996**, *84*, 428–445. [CrossRef]

3. Sarvan, C.; Özkurt, N. ECG Beat Arrhythmia Classification by using 1-D CNN in case of Class Imbalance. In Proceedings of the 2019 Medical Technologies Congress, Izmir, Turkey, 3–5 October 2019; pp. 1–4.
4. Dhakshaya, S.S.; Auxillia, D.J. Classification of ECG using convolutional neural network (CNN). In Proceedings of the 2019 International Conference on Recent Advances in Energy-Efficient Computing and Communication, Nagercoil, India, 7–8 March 2019; pp. 1–6.
5. Chen, Y.; Emer, J.; Sze, V. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016; pp. 367–379.
6. Healthcare, G. *Marquette 12SL ECG Analysis Program: Physician's Guide*; GE Healthcare: Chicago, IL, USA, 2008.
7. Astrom, M.; Olmos, S.; Sornmo, L. Wavelet-based event detection in implantable cardiac rhythm management devices. *IEEE Trans. Biomed. Eng.* **2006**, *53*, 478–484. [CrossRef] [PubMed]
8. Bhat, T.; Akanksha; Shrikara; Bhat, S.; Manoj, T. A Real-Time IoT Based Arrhythmia Classifier Using Convolutional Neural Networks. In Proceedings of the 2020 IEEE International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics, Udupi, India, 30–31 October 2020; pp. 79–83.
9. Burger, A.; Qian, C.; Schiele, G.; Helms, D. An Embedded CNN Implementation for On-Device ECG Analysis. In Proceedings of the 2020 IEEE International Conference on Pervasive Computing and Communications Workshops, Austin, TX, USA, 23–27 March 2020; pp. 1–6.
10. Loh, J.; Wen, J.; Gemmeke, T. Low-Cost DNN Hardware Accelerator for Wearable, High-Quality Cardiac Arrythmia Detection. In Proceedings of the 2020 IEEE 31st International Conference on Application-Specific Systems, Architectures and Processors, Manchester, UK, 6–8 July 2020; pp. 213–216.
11. Fan, A.; Stock, P.; Graham, B.; Grave, E.; Gribonval, R.; Jégou, H.; Joulin, A. Training with Quantization Noise for Extreme Model Compression. *arXiv* **2020**, arXiv:2004.07320.
12. Demidovskij, A.; Smirnov, E. Effective Post-Training Quantization Of Neural Networks For Inference on Low Power Neural Accelerator. In Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN), Glasgow, UK, 19–24 July 2020; pp. 1–7.
13. Elsken, T.; Metzen, J.H.; Hutter, F. Neural architecture search: A survey. *J. Mach. Learn. Res.* **2019**, *20*, 1997–2017.
14. Ambrogio, S.; Balatti, S.; Milo, V.; Carboni, R.; Wang, Z.Q.; Calderoni, A.; Ramaswamy, N.; Ielmini, D. Neuromorphic learning and recognition with one-transistor-one-resistor synapses and bistable metal oxide RRAM. *IEEE Trans. Electron Devices* **2016**, *63*, 1508–1515. [CrossRef]
15. Choi, S.; Yang, J.; Wang, G. Emerging Memristive Artificial Synapses and Neurons for Energy-Efficient Neuromorphic Computing. *Adv. Mater.* **2020**, *32*, 2004659. [CrossRef]
16. Luo, Y.; Yu, S. Accelerating deep neural network in-situ training with non-volatile and volatile memory based hybrid precision synapses. *IEEE Trans. Comput.* **2020**, *69*, 1113–1127. [CrossRef]
17. Stathopoulos, S.; Khiat, A.; Trapatseli, M.; Cortese, S.; Serb, A.; Valov, I.; Prodromakis, T. Multibit memory operation of metal-oxide bi-layer memristors. *Sci. Rep.* **2017**, *7*, 1–7. [CrossRef]
18. Li, F.; Zhang, B.; Liu, B. Ternary Weight Networks. *arXiv* **2016**, arXiv:1605.04711.
19. Brent, R.P.; Kung, H.T. Systolic VLSI Arrays for Polynomial GCD Computation. *IEEE Trans. Comput.* **1984**, *C-33*, 731–736. [CrossRef]
20. Su, F.; Ma, K.; Li, X.; Wu, T.; Liu, Y.; Narayanan, V. Nonvolatile processors: Why is it trending? In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 966–971.
21. Bai, J.; Lu, F.; Zhang, K. ONNX: Open Neural Network Exchange. 2019. Available online: https://github.com/onnx/onnx (accessed on 25 November 2021).
22. Elsken, T.; Metzen, J.H.; Hutter, F. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv* **2018**, arXiv:1804.09081.
23. Yang, L.; Yan, Z.; Li, M.; Kwon, H.; Lai, L.; Krishna, T.; Chandra, V.; Jiang, W.; Shi, Y. Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks. In Proceedings of the IEEE 2020 57th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 20–24 July 2020; pp. 1–6.
24. Nahshan, Y.; Chmiel, B.; Baskin, C.; Zheltonozhskii, E.; Banner, R.; Bronstein, A.M.; Mendelson, A. Loss aware post-training quantization. *arXiv* **2019**, arXiv:1911.07190.
25. Chen, Y.; Yang, T.; Emer, J.; Sze, V. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2019**, *9*, 292–308. [CrossRef]
26. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* **2017**, *45*, 1–12. [CrossRef]
27. Pérez, E.; Zambelli, C.; Mahadevaiah, M.K.; Olivo, P.; Wenger, C. Toward Reliable Multi-Level Operation in RRAM Arrays: Improving Post-Algorithm Stability and Assessing Endurance/Data Retention. *IEEE J. Electron Devices Soc.* **2019**, *7*, 740–747. [CrossRef]
28. Fritscher, M.; Knödtel, J.; Reiser, D.; Mallah, M.; Fey, S.P.D.; Reichenbach, M. Simulating large neural networks embedding MLC RRAM as weight storage considering device variations. In Proceedings of the Latin America Symposium on Circuits and System, San José, Costa Rica, 25–28 February 2021; pp. 1–4.

29. Pechmann, S.; Mai, T.; Potschka, J.; Reiser, D.; Reichel, P.; Breiling, M.; Reichenbach, M.; Hagelauer, A. A Low-Power RRAM Memory Block for Embedded, Multi-Level Weight and Bias Storage in Artificial Neural Networks. *Micromachines* **2021**, *12*, 1277. [CrossRef]
30. IHP. IHP Offers Access to Memristive Technology for Edge AI Computing or Hardware Artificial Neural Networks Applications. Available online: https://www.ihp-microelectronics.com/news/detail/ihp-offers-access-to-memristive-technology-for-edge-ai-computing-or-hardware-artificial-neural-networks-applications (accessed on 25 November 2021).
31. Venkatachalam, K.; Herbrandson, J.; Asirvatham, S. Signals and signal processing for the electrophysiologist: Part I: Electrogram acquisition. *Circ. Arrhythmia Electrophysiol.* **2011**, *4*, 965–973. [CrossRef]
32. Lyons, R.G. *Understanding Digital Signal Processing*, 2nd ed.; Pearson: London, UK, 2004; pp. 550–566.
33. Chan, P.H.; Wong, C.K.; Pun, L.; Wong, Y.F.; Wong, M.M.Y.; Chu, D.W.S.; Siu, C.W. Head-to-head comparison of the AliveCor heart monitor and microlife WatchBP office AFIB for atrial fibrillation screening in a primary care setting. *Circulation* **2017**, *135*, 110–112. [CrossRef] [PubMed]
34. Yu, S.; Chen, P.Y. Emerging Memory Technologies: Recent Trends and Prospects. *IEEE Solid-State Circuits Mag.* **2016**, *8*, 43–56. [CrossRef]
35. Davide Schiavone, P.; Conti, F.; Rossi, D.; Gautschi, M.; Pullini, A.; Flamand, E.; Benini, L. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In Proceedings of the 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Thessaloniki, Greece, 25–27 September 2017; pp. 1–8.
36. Reuben, J.; Fey, D.; Wenger, C. A Modeling Methodology for Resistive RAM Based on Stanford-PKU Model with Extended Multilevel Capability. *IEEE Trans. Nanotechnol.* **2019**, *18*, 647–656. [CrossRef]
37. Li, L.; Talwalkar, A. Random Search and Reproducibility for Neural Architecture Search. In Proceedings of the 35th Uncertainty in Artificial Intelligence Conference, Tel Aviv, Israel, 22–25 July 2019.
38. Cao, S.; Wang, X.; Kitani, K.M. Learnable Embedding Space for Efficient Neural Architecture Compression. *arXiv* **2019**, arXiv:1902.00383.
39. Zoph, B.; Vasudevan, V.; Shlens, J.; Le, Q.V. Learning Transferable Architectures for Scalable Image Recognition. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 8697–8710.
40. Kandasamy, K.; Neiswanger, W.; Schneider, J.; Póczos, B.; Xing, E.P. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. *arXiv* **2018**, arXiv:1802.07191.
41. Wang, Y.; Xu, J.; Han, Y.; Li, H.; Li, X. DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family. In Proceedings of the 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016; pp. 1–6.
42. Baptista, F.D.; Morgado-Dias, F. Automatic general-purpose neural hardware generator. *Neural Comput. Appl.* **2017**, *28*, 25–36. [CrossRef]
43. BMBF. Pilotinnovationswettbewerb "Energieeffizientes Ki-System". 2021. Available online: https://www.elektronikforschung.de/service/aktuelles/pilotinnovationswettbewerb (accessed on 25 November 2021).