International Journal of Parallel Programming (2022) 50:243–266
https://doi.org/10.1007/s10766-022-00730-9

# Fine-Grained Power Modeling of Multicore Processors Using FFNNs

Mark Sagi[1] · Nguyen Anh Vu Doan[1] · Nael Fasfous[1] · Thomas Wild[1] · Andreas Herkersdorf[1]

Received: 15 April 2021 / Accepted: 16 March 2022 / Published online: 29 March 2022


## Abstract

To minimize power consumption while maximizing performance, today's multicore processors rely on fine-grained run-time dynamic power information—both in the time domain, e.g. $\mu$s to ms, and space domain, e.g. core-level. The state-of-the-art for deriving such power information is mainly based on predetermined power models which use linear modeling techniques to determine the core-performance/core-power relationship. However, with multicore processors becoming ever more complex, linear modeling techniques cannot capture all possible core-performance related power states anymore. Although artificial neural networks (ANN) have been proposed for coarse-grained power modeling of servers with time resolutions in the range of seconds, few works have yet investigated fine-grained ANN-based power modeling. In this paper, we explore feed-forward neural networks (FFNNs) for core-level power modeling with estimation rates in the range of 10 kHz. To achieve a high estimation accuracy while minimizing run-time overhead, we propose a multi-objective-optimization of the neural architecture using NSGA-II with the FFNNs being trained on performance counter and power data from a complex-out-of-order processor architecture. We show that relative power estimation error for the highest accuracy FFNN decreases on average by 7.5% compared to a state-of-the-art linear power modeling approach and decreases by 5.5% compared to a multivariate polynomial regression model. For the FFNNs optimized for both accuracy and overhead, the average error decreases between 4.1% and 6.7% compared to linear modeling while offering significantly lower overhead compared to the highest accuracy FFNN. Furthermore, we propose a micro-controller-based and an accelerator-based implementation for run-time inference of the power modeling FFNN and show that the area overhead is negligible.

**Keywords** Processor · Multicore · Power · Modeling · Estimation · Core-level · Artificial neural network · ANN · FFNN · Accuracy · Error · Overhead · Multi-objective-optimization · NSGA-II

Extended author information available on the last page of the article

## 1 Introduction

To take effective management decisions, both power and thermal (P&T) management of multicores depend on accurate run-time dynamic power consumption information at core-level. Due to the cost-prohibitive nature of actually measuring core power, such run-time power information is usually derived from predetermined power models [20] which use observable performance counters, operating frequency, and voltage as inputs. The performance counters are necessary to model the activity and thus indirectly the power consumption of each core. Apart from the spatial resolution (core-level), such dynamic power information also has to have a high time resolution ($\mu$s to ms) to be useful for P&T management of the processor [22, 23]. Most dynamic power models with such spatial and time resolution commonly assume a linear relationship between performance counters and dynamic power and a nonlinear relationship between changes of the voltage/frequency state and dynamic power [2, 4, 6, 14, 25, 29, 32].

Previous works have shown that the relationship between performance counters and dynamic core power can also be nonlinear at core-level at least for low time-resolutions (1 s) [19]. Furthermore, for server-level energy accounting with low time resolutions (0.5 Hz–1 Hz), multiple works have already proposed using ANNs for power/energy modeling to capture such nonlinear relationships [9, 17, 28, 34]. However, the increasing complexity of modern core architectures integrating hundreds of millions of transistors per core and the importance of effective P&T management increasingly necessitate fine-grained high accuracy power models capturing nonlinear performance counter/power relationships. We, therefore, investigate the use of FFNNs for high rate power estimations on core-level and the associated run-time inference overhead. With fast P&T management, e.g. [24], having decision epochs of 0.1 ms, we target a power estimation rate of at least 10 kHz.

A motivational example for using FFNNs is given in Fig. 1 showing core power on the *y*-axis and a time frame of 250 ms on the *x*-axis. PARSEC raytrace is executed and actual power is shown with the green line; estimated power based on a linear model is shown as the red line and estimated power based on an FFNN model is shown with the blue line. One can see that the FFNN model more accurately estimates the actual power consumption than the linear model thus minimizing estimation error which allows for more effective P&T management.

In previous work, we presented a methodology to generate FFNNs with high power modeling accuracy through a single-objective optimization with the goal of

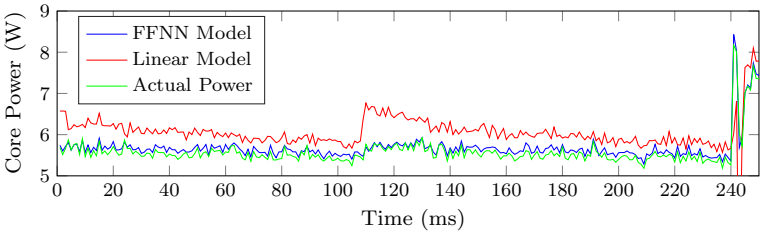



**Fig. 1** Actual core power compared to power estimations using an FFNN model and a linear model

minimizing the root-mean-square error (RMSE) of the power model [26]. For this heuristic optimization, the neural architecture space was constrained to a total of either 2, 3, or 5 layers and the number of neurons per hidden layer were step-wise increased from 1 to up-to an upper limit of 30 neurons. The step-sizes were 1, 3, and 7 for the respective layers. This approach *constrains* the number of investigated neural architectures as not all possible architectures for 2, 3, or 5 layers are evaluated while *exhaustively* evaluating those within the specified neuron number steps.

We extend that work [26] with multi-objective optimization of the neural architecture such that both the MSE as well as the run-time overhead of the FFNN is minimized. To allow for more possible solutions, the neural architecture space is less constrained by allowing all neuron combinations within the upper neuron limit, i.e. no step-wise reduction of the number of neurons per layer. To avoid an explosion in training time, we propose a metaheuristic multi-objective optimization of the neural architectures using a genetic algorithm based on NSGA-II [12].

With this paper we make the following contributions toward fine-grained run-time power estimation of multicores:

- We explore FFNNs for power estimation on core-level with an estimation rate of 10 kHz.
- We optimize the number of layers and the number of neurons per layer of the FFNNs with the objective of minimizing both power modeling error as well as run-time overhead.
- We show that relative estimation error for the optimized FFNNs are between 4.1% and 7.5% lower compared to a state-of-the-art linear modeling approach and between 2.1% and 5.5% lower compared to a multivariate polynomial regression model.
- We propose a micro-controller implementation as well as an accelerator implementation for run-time inference of the FFNNs which allow for minimal area overhead or very high estimation rates of up to 1.38 MHz.
- The FFNNs optimized for both accuracy and overhead offer on average 60% lower overhead and slightly smaller accuracy compared to the FFNNs optimized single-objectively for high accuracy.

## 2 Related Work

A range of different methodologies for run-time dynamic power estimation on core-level or even on core-component-level for multicore processors have been proposed. Works that focus on high estimation rates usually rely on linear models to describe the relation between performance counters and dynamic power and can be found for Intel or AMD multicore processors in [2, 4, 6], for IBM multicore processors in [14] and for embedded ARM multicore processors in [25, 29, 32]. Such linear models have the advantage of low run-time overheads but can suffer from lower accuracy compared to more complex models.

McCullough et al. [19] first identified non-linear power responses due to changing workloads in multicore power traces which have to be accounted for in

power models. They traced the power consumption of an Intel Core i7, broke down power consumption on core-level, and explored both linear and non-linear modeling techniques (polynomial regression and support vector regression). However, the sampling rate was comparatively low (1 Hz) and their non-linear models did not significantly improve upon linear models.

Further non-linear modeling techniques gained traction in the research area of power/energy modeling for datacenter systems and cloud servers. The first to propose ANNs for power modeling were Cupertino et al. [9]. They determined that an FFNN architecture with 2 hidden layers—20 neurons in the first layer and 5 neurons in the second layer—to provide accuracy improvements compared to state-of-the-art linear power modeling techniques. However, the very low sampling rate of 3 Hz for power and performance counter data limits the applicability to energy accounting and load balancing in datacenters. In contrast, our work focuses on high rate power estimations with estimation rates of 10kHz and provides a methodology to systematically generate FFNN architectures with both high accuracy and low run-time overhead.

In [15], ANNs were used to predict the power consumption of applications across different processor architectures with the underlying assumption that the linear power models for each processor architecture are sufficiently accurate. In [28], the so-called *additivity* of performance counters regarding energy modeling of multicores is explored where additivity denotes the robustness of reusing a performance counter as model input for a wide range of applications. To determine the additivity of performance counters for their energy model generation, the use of linear, tree-based, and ANN models was investigated. However, parametrization and optimization of the neural network models are not discussed and the focus was on full system energy modeling with low estimation rates.

Power modeling of multicores with multi-thread inference was explored in [7]. High modeling accuracies were found on the training data for a three-layered ANN with sigmoid activation function as well as for a linear model. No details were given on the hyperparametrization approach and due to comparatively similar accuracies for the ANN and linear model, the remainder of the work used the linear model. In contrast, our work provides a systematic approach for hyperparametrization of ANNs for power modeling and shows significant improvements compared to state-of-the-art linear power models.

Another work exploring three different ANNs (FFNN, Elman, and LSTM) for cloud server power modeling is given in [17] with BP and Elman having a single hidden layer with 25 neurons and the LSTM having 2 hidden layers with 10 neurons each. The resulting power model provides high accuracy at course-grained spatial (system-level) and time (1 Hz) resolution. In contrast to our work, core-level power models with high estimation rates (10 kHz) were not explored. Recurrent Elman neural networks have been proposed for coarse-grained power modeling of cloud servers where one hidden layer encodes power states and which exploits time-series performance information [34]. Finally, for system design, the development of P&T management algorithms and power modeling algorithms, power simulators are often used, e.g. McPAT [16]. Although the underlying principles of such power

simulators are highly accurate, they cannot be used for run-time power estimation due to their large computational overhead.

Multi-objective optimizations of neural network architectures based on NSGA-II [12] have been proposed in several works [8, 18, 31], usually for image classification tasks. To the best of our knowledge, we are the first to propose using NSGA-II to optimize neural architectures for multi-core run-time power modeling.

In contrast to previous work, this paper focuses on generating FFNN-based power models accounting for non-linear effects with fine-grained spatial (core-level) and time (10 kHz) resolution. Extending the power estimations to the core-level and increasing the time resolution by four orders of magnitude, make a thorough investigation of the needed FFNN complexity—regarding the number of hidden layers and neurons—and the overhead for run-time inference necessary. Our FFNNs are optimized both for accuracy and low run-time overhead with the resulting power estimations being applicable for run-time P&T management purposes.

## 3 Feed-Forward Neural Networks for Power Modeling

Power modeling for runtime power estimation is inherently a regression problem with the desired power information as a dependent variable and the performance counters as independent variables. In the following, we first present our previous single-objective [26] methodology on FFNN power modeling. We focus on dynamic core-level power information $P_{core,j}$ where $j$ denotes the $j$-th core of the multicore processor. The core power is estimated during run-time through $n$ performance counters $PC_i$ with $0 \leq i \leq n$ all related to their respective cores. With actual $P_{core,j}$ not being observable for each individual core, we use the following approximation for the model generation step when only the $j$-th core is active at a time:

$$P_{core,j} = P_{pack} - P_{idle}. \tag{1}$$

Package-level power $P_{pack}$ can be observed through instrumentation of the main-board, i.e. actual power sensors, and $P_{idle}$ is the idle power of the processor when no core is active. With only $P_{pack}$ being observable, we generate different models (FFNN, polynomial, linear) for $P_{pack}$ and subtract $P_{idle}$ to derive core-level power consumption. Therefore, the $PC_i$ and $P_{pack}$ data used for model generation is reduced to timeframes where only a single core is active at a time to capture the power response of that particular core. As we only investigate a homogeneous multicore processor in this work, the power models for the $j$-th core can be generalized to any core of the system by using the performance counters of those cores as model input, respectively. The error (cost) function for generating the subsequent power models is then:

$$P_{pack,error} = |P_{pack,act} - P_{pack,est}| \tag{2}$$

where the subscripts $_{est}$ and $_{act}$ indicate estimated power and actual observed power, respectively.

### 3.1 ANN Architectures and Hyperparameter Solution Space

There exists a multitude of ANN architectures, e.g. FFNN, Elman, LSTM, for modeling and predicting of non-linear functions and systems. With most fine-grained power models using linear regression models, we keep our analysis to comparatively simple ANN architectures. Our goal is to achieve higher estimation accuracies than with linear modeling techniques while adding as little additional modeling complexity and run-time overhead as possible. For this reason, we choose well-known feedforward networks which can theoretically model any nonlinear function according to the universal approximation theorem [13]. Similar to previous works, we do not use any input delay on the $PC_i$ inputs, i.e. we do not generate any autoregressive models. While linear regression models are at risk of underfitting the underlying dynamic power relationship, FFNNs are at risk of both underfitting and overfitting the power relationship. With a finite amount of training data, FFNNs of sufficient size can fit each data point perfectly, i.e. memorize the data, while not actually learning the underlying relationship. In that case, the $PC_i$ data is overfitted and the estimation errors on $P_{core}$ for untrained $PC_i$ data will be significant. Therefore, careful consideration has to be taken regarding the chosen hyperparameters of the FFNN which are distinguished between *algorithm* hyperparameters (learning related) and *model* hyperparameters (architecture-related). For the algorithm hyperparameters, we train a multitude of networks for dynamic power estimation and compare both the resulting accuracy as well as training time and find that conjugate gradient backpropagation with Polak-Ribière updates provide the best training speed/accuracy trade-off. As stop conditions for training the FFNN, we use the following:

- stop after 1000 training epochs OR,
- an MSE below 1% on the training data OR,
- a minimum performance gradient of $1 \cdot 10^{-5}$ OR,
- 5 subsequent failed validation tests where additional training leads to higher estimation errors on the validation data.

The question of how to determine the optimal FFNN model hyperparameters is still an ongoing topic of research, therefore, we follow best practices for hyperparametrization. As a first step, the model hyperparameters have to be confined. For the activation function of the hidden neurons, we choose *tanh* after sweeping over a set of different activation functions and comparing estimation accuracy.

*Single-objective Hyperparameter Solution Space* The goal of our single-objective optimization methodology is to find neural architectures minimizing the power estimation error. The run-time overhead is then simply a direct consequence of the chosen solution with the lowest estimation error.

Our single-objective methodology exhaustively searches through a constrained solution space of FFNN architectures. Therefore, the solution space FFNN architectures should be both, representative and well-constrained such that necessary training time falls within the given compute limitation for generating the FFNN models.

For the number of hidden layers and hidden neurons per layer, we align ourselves with the related work for coarse-grained power models for servers/datacenters and confine the hidden layer and hidden neuron hyperparameters as shown in Fig. 2. We explore one two-layered *shallow* network with 1–30 neurons per hidden layer. Note, that we explore all possible combinations of the number of hidden neurons per layer, i.e. 900 differently parameterized two-layered FFNNs. We further investigate

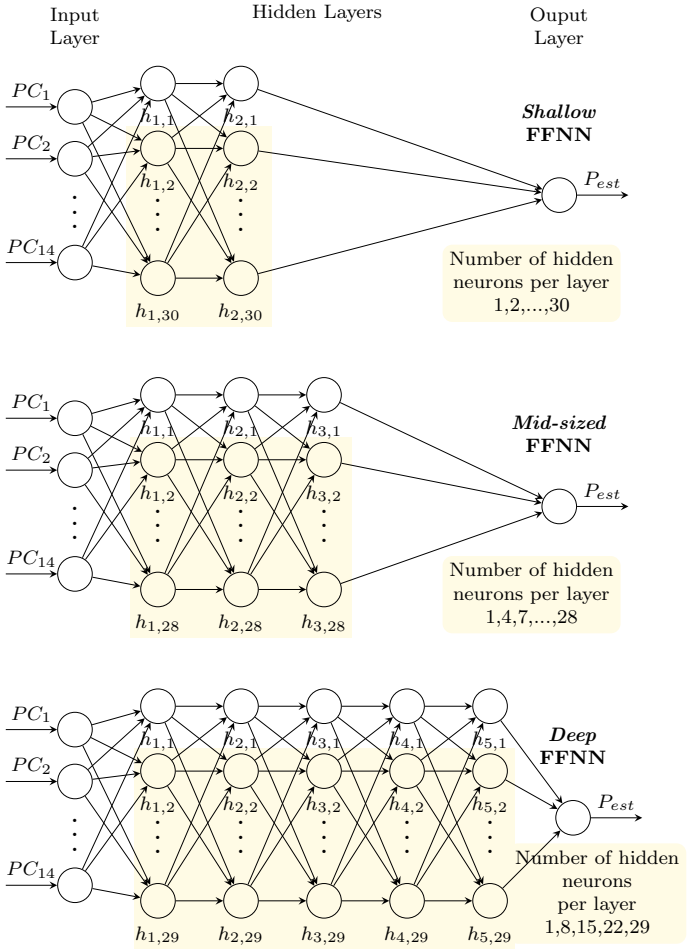



**Fig. 2** Overview of the FFNN architectures investigated by the single-objective methodology; the yellow shaded rectangles indicate the ability to parameterize the number of hidden neurons per layer, i.e. from at least one hidden neuron per layer up to the given maximum number

one *mid-sized* network with 3 hidden layers where the number of neurons per layer can be any number of 1, 4, 7, 10, 13, 16, 19, 22, 25, 28 and one *deep* network with 5 hidden layers where the number of neurons per layer can be 1, 8, 15, 22, 29.

The number of neurons per layer is constrained for the mid-sized and deep networks to keep the amount of training time on a reasonable level. Although adding layers and increasing the number of neurons per layer increases the risk of overfitting, it also decreases the risk of underfitting due to an undersized FFNN.

Overall, the three different layer sizes and possible neurons-per-layer of the FFNN neural architectures constitute $30^2 + 10^2 + 5^5 = 5025$ different neural architectures. This is the overall solution space for the single-objective optimization. In the following, we present the extension of our previous single-objective optimization methodology.

*Multi-objective Hyperparameter Solution Space* The goal of our multi-objective optimization methodology is to find neural architectures minimizing both the power estimation error as well as the run-time overhead. To simultaneously optimize both objectives, we need to explore a larger solution space than for the single-objective optimization which has to be searched heuristically rather than exhaustively.

We constrain the solution space to FFNN architectures with 2, 3, 4, or 5 layers with up to 30 neurons per layer without any constraints on the neuron amount. Therefore, the solution space is $30^2 + 30^3 + 30^4 + 30^5 = 25,137,900$ different neural architectures which is four magnitudes larger than for the single-objective optimization.

In the following, we show both our single-objective as well as multi-objective methodologies to find well-performing FFNN architectures for run-time power estimation.

### 3.2 Single-Objective Hyperparameter Optimization

Our methodology for the hyperparametrization of the model parameters and for the final training and computation of expected estimation accuracy are shown in Fig. 3.

We use tenfold cross validations for both, model hyperparametrization and to train a final FFNN of each size (shallow, mid-sized, and deep), i.e. for actual deployment.

First, the traced $PC_i$ and $P_{pack}$ data is partitioned into a training/validation data set (75%) and a holdout data set (25%) such that the holdout data sets contain benchmarks from different benchmark suites covering diverse power behaviors. The holdout data set is neither used for hyperparametrization nor to train the three final FFNNs and can thus be used to determine the actual performance of these FFNNs on data they have not seen yet. For the hyperparametrization, we iterate over all possible hidden neuron per layer combinations for each FFNN size (*shallow*, *mid-sized*, *deep*) and execute the first cross validation loop. In this loop, the training data set is further partitioned into tenfolds and each fold is used once for validation with the remaining folds being used for training the FFNN. We repeat this step for each fold ten times to produce statistically significant results and to be able to remove outliers, i.e. diverging FFNNs. Thus, 100 FFNNs are generated for each possible

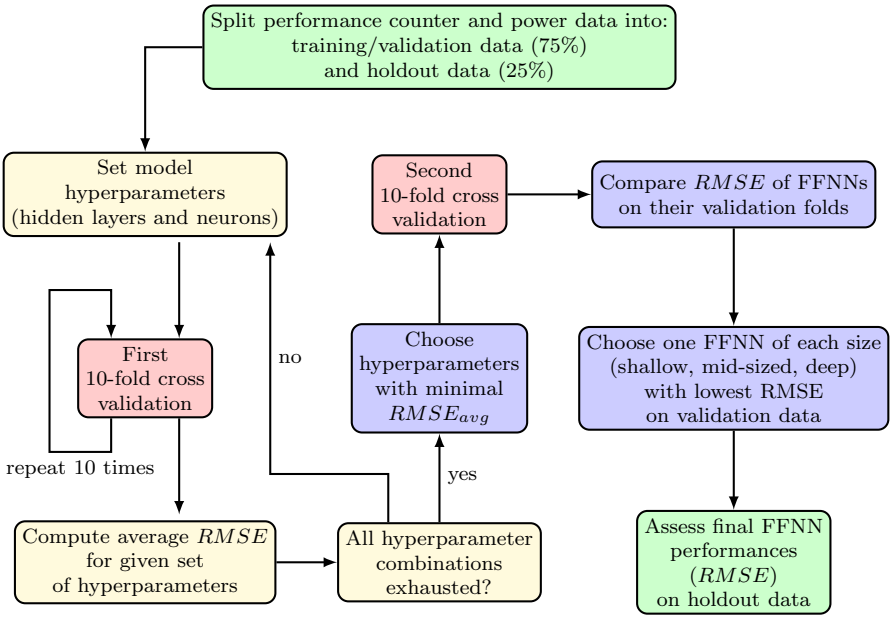



**Fig. 3** Flowchart of model hyperparametrization using a first tenfold cross validation, final FFNN generation using a second tenfold cross validation and FFNN estimation accuracy assessment on holdout data

model hyperparameter combination. After a full hyperparametrization run, the estimation error on the validation data is averaged for each hyperparameter over all folds. We then choose the hidden neuron parametrizations for each FFNN size with the lowest average *RMSE* under the assumption that these parametrizations provide the best general fit for the given performance/power data. The benefit of the repeated tenfold cross validations lies in the robustness of the average *RMSE* for the different hyperparameters and thus in choosing with high confidence good hyperparameters for generating the final FFNNs.

The hidden neuron parametrizations are then used in the second tenfold cross validation step where we generate an FFNN for each training/validation fold combination, i.e. 10 FFNNs for each FFNN size (*shallow*, *mid-sized* and *deep*) and choose those FFNNs for testing which performed best on their corresponding validation data. We use the second cross validation to minimize the risk of selecting an overfitting FFNN from the first cross validation where 100 different FFNNs were generated for each hyperparameter. The risk of the FFNN with the highest accuracy on their respective validation fold being overfitting is higher when 10 such FFNNs are available to choose from rather than just one. In the final step, we test the three chosen FFNNs on the holdout data to assess their potential dynamic power estimation performance in an actual deployment environment.

### 3.3 Multi-Objective Hyperparameter Optimization

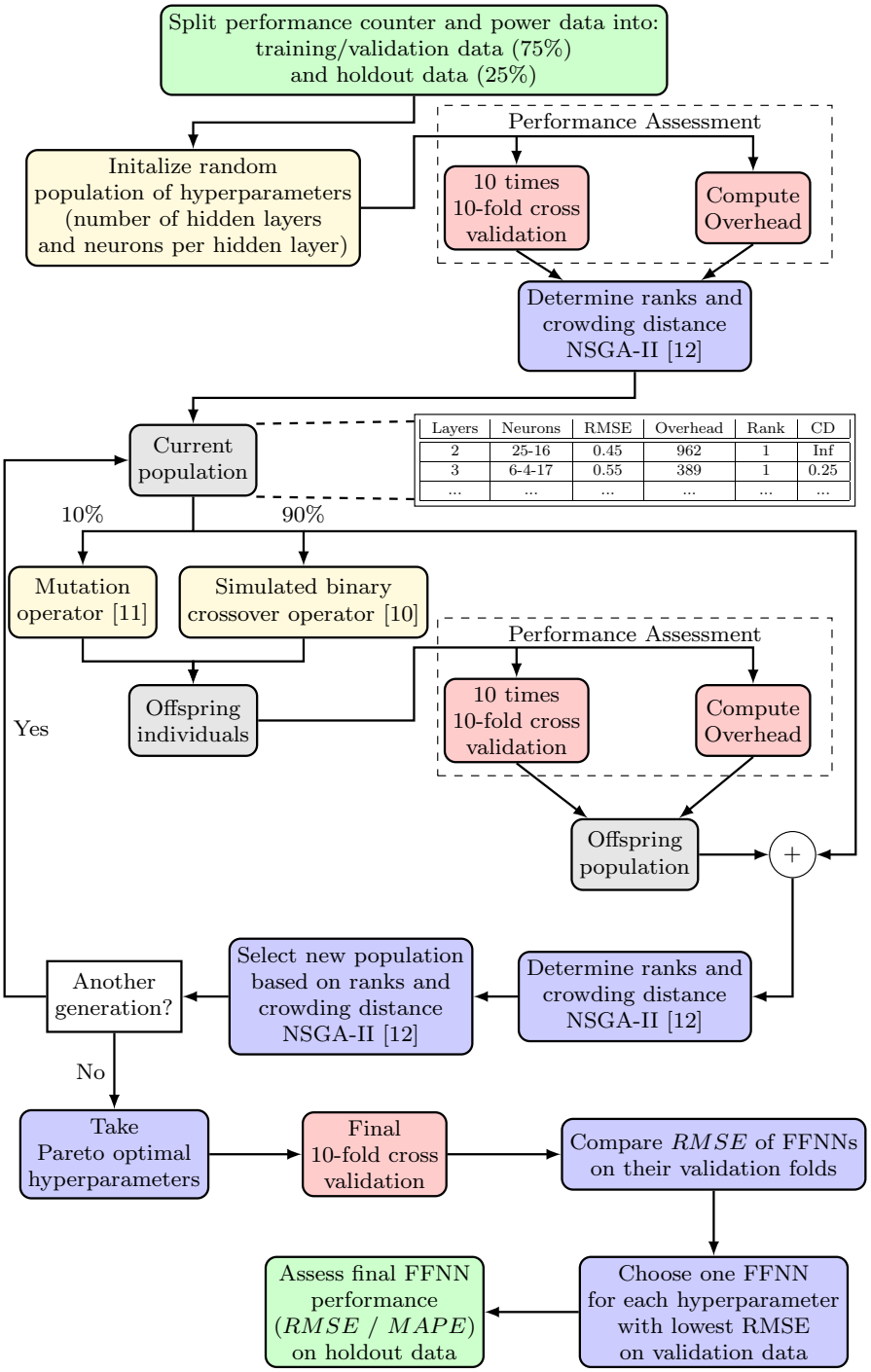
We adapt NSGA-II [12] with simulated binary crossover (SBX) [10] and polynomial mutation [11] for our multi-objective optimization of the neural hyperparameters for our power modeling FFNNs. SBX was chosen as the basis for our crossover operation as it enables gradual changes close to the chosen parent neural architectures. By this, the solution space is searched with higher probability in the vicinity and with lower probability in larger distances to existing solutions by the crossover operation. To ensure diversity of the population and avoiding being stuck in a local minima, we use polynomial mutation with a comparatively large percentage of the offspring populations generated by this mutation operator.

Our goal is to find neural architectures minimizing the modeling error while also minimizing the run-time inference overhead. In addition, we aim to make our methodology generic in regard to the neural architecture solution space, especially the number of layers. This has the advantage, that neural architectures with different number of layers can be searched in a single optimization run allowing for either faster convergence towards more optimal architectures within the same training time. The disadvantage of this approach is that in case of small population numbers, some layer values might be excluded too early from the population without the chance of getting back into the population.

We choose an explicit encoding of the neural architecture, containing the number of layers and the number of neurons per layer, as illustrated in Table 1. This table also contains the used performance metrics. The performance metrics are each individual's modeling error (RMSE) and its run-time overhead counted in MAC operations. The final dense FFNN layer after the last hidden layer is not explicitly encoded as its specification is fully derived from the last non-zero hidden layer. In the following, we use a population size of 50 to keep sufficiently large numbers of individual neural architectures in the population while also keeping the necessary compute time for each generational step in a reasonable range. We keep also an archive of all populations generated over the different generations to avoid computationally costly re-evaluation of the same neural architecture in the case that it is generated multiple times by crossover/mutation. An overview of our multi-objective methodology generating and optimizing the population is shown in Fig. 4.

**Table 1** Example of population encoding and with performance and NSGA-II metrics

| Architecture encoding | | Objective performance | | Relative performance NSGA-II | |
|---|---|---|---|---|---|
| Number of Layers | Neuron Distribution | RMSE | Overhead (MACs) | Rank | Crowding Distance |
| 3 | 6-4-17-0-0 | 0.55 | 389 | 1 | Inf |
| 2 | 25-16-0-0-0 | 0.45 | 962 | 1 | 0.25 |
| – | – | – | – | – | – |


Split performance counter and power data into: training/validation data (75%) and holdout data (25%)
Performance Assessment
Initalize random population of hyperparameters (number of hidden layers and neurons per hidden layer)
10 times 10-fold cross validation
Compute Overhead
Determine ranks and crowding distance NSGA-II [12]
Current population
Layers Neurons RMSE Overhead Rank CD
2 25-16 0.45 962 1 Inf
3 6-4-17 0.55 389 1 0.25
... ... ... ... ... ...
10%
90%
Mutation operator [11]
Simulated binary crossover operator [10]
Performance Assessment
Offspring individuals
10 times 10-fold cross validation
Compute Overhead
Yes
Offspring population
Another generation?
Select new population based on ranks and crowding distance NSGA-II [12]
Determine ranks and crowding distance NSGA-II [12]
No
Take Pareto optimal hyperparameters
Final 10-fold cross validation
Compare $RMSE$ of FFNNs on their validation folds
Assess final FFNN performance ($RMSE$ / $MAPE$) on holdout data
Choose one FFNN for each hyperparameter with lowest RMSE on validation data

◀**Fig. 4** Flowchart of multi-objective FFNN hyperparametrization using NSGA-II, tenfold cross validations and final FFNN estimation accuracy assessment on holdout data

First, the performance counter data is split the same way as previously into training/validation data as well as holdout data. Afterward, an initial population is generated with a random number of layers and neurons per layer for each individual constrained by the overall layer and neuron limits. The crossover operation is shown in Algorithm 1. The random population of neural architectures is then assessed with the same repeated tenfold cross validation as in the previous section to get consistent modeling errors, i.e. RMSE, for each neural architecture. In addition, the run-time overhead is calculated in multiply accumulate (MAC) operations for a single inference of the neural network architecture to generate a power estimate. Based on the error and overhead values, the individual ranks and crowding distance is computed using NSGA-II [12]. With this, the initial starting population is complete and functions as the current population.

In the next step, our genetic algorithm optimizes the population over a multitude of generations by applying a mutation and a crossover operation. Overall, we generate 50 new neural architectures, i.e. the offspring population, with 10% (5) neural architectures generated by the mutation operation and 90% (45) generated by the crossover operation. We decided on this distribution of the mutation/crossover likelihood to have sufficiently high randomness in the offspring population throughout the optimization while also having sufficiently high convergence towards improved neural architectures. In case a newly generated neural architecture has already been evaluated, i.e. is in the generational archive, we reapply the crossover or mutation operation to get a novel neural architecture while ensuring the above mutation/crossover distribution.

Our crossover operation for the neural architectures is shown in Algorithm 1. Note, that the two parent architectures used for a single crossover operation are chosen by binary tournament from the current population using the crowded distance operator as described in NSGA-II [12]. We set the SBX parameter regarding the probability distribution as $n = 20$ resulting in a wide probability distribution for the neuron values. With the number of layers being an explicit part of the solution space and thus optimization, the crossover operator has to differentiate two cases. In the first case, both parents have the same number of layers allowing us to simply use SBX once for each layer to generate two children architectures. In the second case, both parents have different number layers and we designate randomly with equal chance one of the parents as the *dominant* parent. The dominant parent's layer number is then set for both child architectures. If the dominant parent architecture has *fewer* layers than the other parent architecture, we perform SBX over the neuron values up to the dominant (smaller) layer number generating two children architectures. If the dominant parent architecture has *more* layers than the other parent architecture, we perform SBX over the neuron values up to the non-dominant (smaller) layer number and copy the dominant parent architecture's neuron values of the remaining layers for the two children architectures.

Finally, the real-valued neuron values of the children architectures are rounded to the closest integer values and in case the neuron limits (1 or 30) are underflown or overflown, the neuron values are set to the respective neuron limits.

**Algorithm 1:** Crossover operation

```
Input  : 2 parent architectures p_1 and p_2 chosen by binary
         tournament
Output: 2 child architectures c_1 and c_2
if both parents have same number of layers n_1 == n_2 then
    foreach layer i = 1 : n_1 do
        Generate real-valued neuron distributions for c_{1,i} and c_{2,i} using
         SBX [10]
    end
else
    Randomly choose dominant parent from p_1 and p_2 if dominant
     parent p_d has more layers n_d than other parent n_o then
        foreach layer i = 1 : n_o do
            Generate real-valued neuron distributions for c_{1,i} and c_{2,i}
             using SBX [10]
        end
        foreach layer i = n_o + 1 : n_d do
            c_{1,i} = p_{d,i} and c_{2,i} = p_{d,i}
        end
    else
        foreach layer i = 1 : n_o do
            Generate real-valued neuron distributions for c_{1,i} and c_{2,i}
             using SBX [10]
        end
    end
end
Round neuron values of to c_{1,i} and c_{2,i} closest integer values
if a neuron value exceeds min/max neuron limits then
    Set neuron values to limit values
```

For the mutation operation, random individual neural architectures are chosen from the current population and the number of neurons per layer mutated using polynomial mutation with the same probability parameter $n = 20$ as for the crossover operation [11]. Neuron values are increased and decreased with equal probability and the resulting neuron values rounded to the nearest integer value, with the value saturating in case it exceeds the neuron limit.

The performance of the offspring population in regard to modeling error and overhead is then assessed, and the individual ranks and crowding distance are determined. After both objectives have been evaluated, the current and the offspring population are combined and their relative performance is computed according to NSGA-II. In the final step of a generation, the best individuals—according to ranks and crowding distance—are selected for the next generation.

When the maximum number of generations is reached, the performance of the individual neural architectures of the final generation in the Pareto front is assessed on the holdout data. For this last step, the approach is similar to the single-objective methodology with FFNNs generated for the different validation folds and the ones with the lowest RMSE chosen for assessment on the hold out data with the only difference that there can be multiple neural architectures in the Pareto front and thus multiple final FFNNs.

## 4 Experimental Setup

We use HotSniper [21], which is based on the Sniper multicore simulator [5] and expands the Sniper simulator with periodic power simulations using McPAT [16]. The sniper simulator uses interval simulation to speedup simulation times while providing good simulation accuracy. It is widely used for the research of processor power/thermal management and power modeling, e.g. in [23, 27]. Compared to a cycle-level simulator, an interval simulator focuses on accurately simulating performance changes due to stalls of the execution flow, while modeling performance on a higher abstraction layer when the execution flow is continuous.

Our experimental framework simulates a 16-core processor with Intel Gainestown core microarchitecture, with the details given in Table 2.

We trace common performance counters similar to other state-of-the-art work, e.g. [2], with a sampling rate of 10 kHz. Some works use a larger number of performance counters—in the range of 50–100—to estimate power due to the possibility of the estimation error decreasing with an increasing number of performance counter inputs. However, processors commonly only support recording a small number of performance counters—in the range of 5–15—at the same time [4].

We use the power and energy information provided by McPAT in our evaluation which generates power information on package, core, and core component granularity with microsecond resolution. Finally, the benchmarks used for training the FFNNs and generating reference power models are also given in Table 2. The PARSEC 2.1 benchmarks vips, ferret as well as Splash-2 benchmarks volrend, water-nsquared and water-spatial had to be left out due to errors with the PIN tool used by Sniper for benchmark instrumentation. The usage of McPAT for obtaining power can be a limitation for our work. The simulated power values from McPAT can have inaccuracies which either favor/are neutral/disfavor our approach compared to the other state-of-the-art approaches. Our assumption is that the possible inaccuracies are on average neutral in regard to the comparison of modeling approaches.

**Table 2** Simulation frameworkv

| 16-cores (2×2 tiles) with NoC interconnect | | |
|---|---|---|
| Intel gainestown core architecture | | |
| Memory architecture | | |
| L1 caches | 32 KB (private) | |
| L2 caches | 256 KB (private) | |
| L3 caches | 8 MB (shared per tile) | |
| Performance information | | |
| Processor | Performance | |
| Unit | Counter | |
| Core | Instruction per cycle | IPC |
| | # Branch instructions | BPU |
| | # Floating point instructions | FP |
| | % C0 state residency of a core | C0 |
| L2 & L3 | # Load instructions | LxLI |
| Cache | # Store instructions | LxSI |
| | # Load misses | LxLM |
| | # Store misses | LxSM |
| | % Cycles lost due to misses | LxCLK |
| Workloads for model generation and evaluation | | |
| Suite | Benchmarks | |
| PARSEC 2.1 [3] | Blackscholes, bodytrack, canneal, dedup, facesim, fluidanimate, freqmine, raytrace, streamcluster, swaptions, x264 | |
| Splash-2 [33] | Barnes, cholesky, fft, fmm, lu, ocean, radiosity, radix, raytrace | |

## 5 Evaluation

### 5.1 Reference Power Models

We compare the estimation accuracy of the final FFNNs with the state-of-the-art linear approach published in [2] and a polynomial regression model as proposed by McCullough et al. [19]. For the linear model, we execute a set of microbenchmarks and the PARSEC/Splash-2 benchmarks on the system, trace $P_{pack}$ and $PC_i$ and generate a core-level linear regression model. Although [19] argues that a polynomial regression model was not able to accurately capture the non-linear power relationships—possibly due to overfitting—we still use it to test the hypotheses that our performance/power data could potentially be described well through polynomial regression and therefore obviating the need for more complex ANN methodologies. To generate the polynomial regression model, we use a similar methodology as described in Sect. 3 for FFNNs. First, we do repeated tenfold cross validations to determine the best polynomial order and then generate the final polynomial regression model through another tenfold cross validation

choosing the polynomial model with the highest accuracy on its respective validation fold. We explored maximum polynomial orders of 1–6 for the independent $PC_i$ inputs and found that a maximum polynomial order of 2 offered the best average estimation performance on the validation data.

### 5.2 Single-Objective Hidden Neuron Architecture Results

We first look at the best model hyperparameters of the three hidden layer sizes and the average root mean squared error (RMSE) on the validation data from the tenfold cross validation from our previous work [26]. For each of the three different FFNN sizes, the hidden neuron parametrizations with the lowest average RMSE are shown in Table 3. In addition, the relative error compared to the average core power is computed as $\frac{RMSE}{P_{avg}}$.

Shallow FFNNs with 2 hidden layers and 14 neurons in the first hidden layer and 29 neurons in the second hidden layer offer on average the best performance on the validation data. Both the mid-sized and deep FFNN offer worse performance than the shallow FFNN on the validation folds. Notably, all FFNNs steer towards larger average neuron counts within the available solution space. However, for the mid-sized and deep FFNNs where the larger neuron counts could be due to the coarse granularity of the solution space. Smaller FFNN architectures with specific neuron-distributions might be outside of the solution space but still offer similar or better accuracy for FFNNs with three or four layers.

### 5.3 Multi-Objective Hidden Neuron Architecture Results

As we use a multi-objective metaheuristic, we first investigate its convergence properties towards good FFNN architectures over time. In Fig. 5, the average RMSE and overhead values of the neural architectures are shown over 50 generations of genetic optimization. Also, the smallest RMSE and overhead values of the Pareto-optimal (rank 1) solutions in the population are also shown. We observe that the average overhead values are steadily decreasing while the average RMSE values fluctuate over the generations. After further investigating the population composition from generation to generation, we see that small numbers of neural architectures with above-average RMSE and very small overheads are introduced and kept in the population on the lower ranks. For example, neural architectures with only 1–3 neurons on a single layer drastically decrease overhead while simultaneously increasing the modeling error. However, this behavior is not in itself problematic as we manually observe an overall increase in population

**Table 3** FFNN architectures found by the single-objective optimization methodology with average RMSE on the randomized validation data

| FFNN | Number of Hidden layers | Neurons per Hidden layer | Average RMSE | Relative Error |
|---|---|---|---|---|
| Shallow | 2 | 14–29 | 0.31 W | 5% |
| Mid-sized | 3 | 25–28–25 | 0.36 W | 6% |
| Deep | 5 | 22–08–22–15–15 | 0.33 W | 6% |

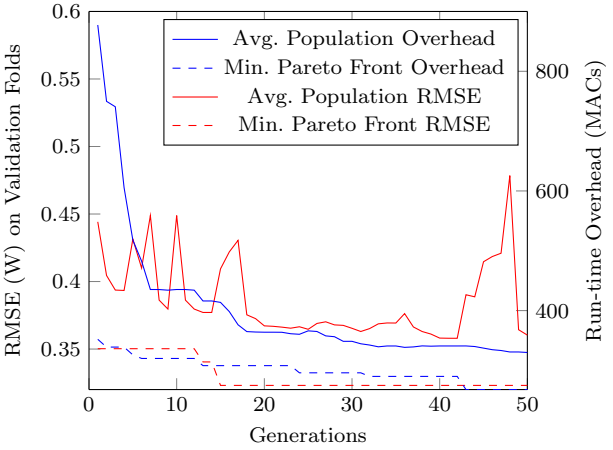



**Fig. 5** Average population performance over 50 generations as well as lowest overhead and RMSE values of the Pareto optimal solutions

diversity which can lead to more optimal solutions further down the generational timeline. As was to be expected for the solutions in the actual Pareto front of the population, the minimal values observed for either the RMSE or the overhead are both decreasing steadily. We conclude that our heuristic multi-objective optimization successfully optimizes the initial random neural architectures.

The Pareto optimal solutions after 50 generations with their corresponding RMSE values on the validation folds and their run-time overhead are given in Table 4. All four solutions have quite similar neuron architectures with small trade-offs between overhead and modeling error. We only found neuron architectures with different layer numbers and significantly different neuron distributions within the lower ranks of the final population. When comparing these architectures with the results of the single-objective optimization methodology in Table 3, we can see that the overhead values were successfully minimized, however the modeling errors are slightly larger for our heuristically optimized architectures.

Interestingly, our heuristic methodology has not found the same or similar neural architectures with lowest RMSE as the exhaustive search. However, this is not

**Table 4** FFNN architectures found by the multi-objective optimization methodology with average RMSE on the randomized validation data

| FFNN | Number of Hidden layers | Neurons per Hidden layer | Average RMSE | Relative Error | Run-time Overhead |
|---|---|---|---|---|---|
| Multi-1 | 3 | 5–4–8 | 0.32 W | 5% | 326 MACs |
| Multi-2 | 3 | 3–4–8 | 0.34 W | 6% | 290 MACs |
| Multi-3 | 3 | 3–2–8 | 0.37 W | 6% | 268 MACs |
| Multi-4 | 3 | 4–4–8 | 0.34 W | 6% | 308 MACs |

unexpected as the solution space for the heuristic methodology is four magnitudes larger than for the exhaustive, constrained single-objective methodology. In regard to compute/training time for the optimization, the heuristic optimization evaluated 2500 different neural architectures, while the single-objective optimization evaluated 5025 architectures.

### 5.4 Run-Time Inference Overhead for an On-Chip Micro-Controller Implementation

The computational and memory overhead of run-time inference of the FFNNs for producing a single power estimation is assessed by the necessary number of MAC operations and the memory needed to store the 32-bit neuron weights. An overview of both the compute overhead and memory overhead is given in Table 5 for all the different FFNN architectures determined through both single-objective and multi-objective optimization. The number of necessary MAC operations is derived from input computations and the computations in the hidden layers and the output layer.

Compared to the linear regression model, the *shallow* FFNN needs approximately 60 times more MAC operations and 40 times more memory while the *multi-x* FFNNs optimized for both accuracy and overhead need on average 21 times more MAC operations and 20 times more memory. Both, the computational and memory overhead are magnitudes higher for any FFNN implementation and we, therefore, discuss the feasibility and area overhead of a run-time inference implementation of the different FFNNs in the following.

At least on IBM multicore processors, micro-controllers are integrated for both power estimation—so-called power proxies—and for power management purposes [14]. In the following, we approximate the transistor overhead for power estimations using the shallow ANN assuming integrated micro-controllers for power estimation. As a reference micro-controller, the 32-bit ARM Cortex M0 is well established, can be conservatively operated at 50 MHz with an implementation using less than 100k

**Table 5** Necessary computations and memory for a single power estimation/model inference

| Model | Number of MAC operations | Memory in kBit |
|---|---|---|
| Single-objective optimization | | |
| Shallow FFNN | 827 | 20 |
| Mid-sized FFNN | 1971 | 56 |
| Deep FFNN | 1426 | 39 |
| Multi-objective optimization | | |
| Multi-1 FFNN | 326 | 10 |
| Multi-2 FFNN | 290 | 9 |
| Multi-3 FFNN | 268 | 8 |
| Multi-4 FFNN | 308 | 10 |
| Reference | | |
| Linear Model [2] | 14 | 0.5 |
| Polynomial Model | 28 | 1.0 |

transistors [1]. Besides, the required SRAM to store the weights for the *shallow* ANN would add an additional 120 k transistors. Each MAC operation and its associated load/store instructions takes 6 cycles on the M0 leading to approximately 5 k cycles for a single inference of the shallow FFNN, therefore, power estimations could be executed with a periodicity of $100\mu s$.

Depending on the requirements of the P&T management, one micro-controller would be needed per core if 10kHz power estimations are needed. Such a micro-controller implementation leads to an overhead of 250 k transistors under conservative assumptions and has to be compared to the power estimations being used for a complex out-of-order core having hundreds of millions of transistors. The area overhead of—at maximum 0.25%—would decrease the average relative power estimation error by 7.5%, translating to better P&T management and thus the possibility of higher compute performance and/or higher energy efficiency. Area overhead could be further decreased by custom logic for FFNN inference which would however remove the programmability of the power estimator through firmware changes. Also, lower estimation rates in the range of 1 kHz would allow one power estimating micro-controller to serve multiple cores thus also decreasing area overhead.

In conclusion, the *multi-x* FFNNs and the *shallow* FFNN can be implemented for run-time power modeling on today's multicore processors. The *multi-x* FFNNs have, due to their multi-objective optimization taking overhead into account, a clear advantage compared to the *shallow* FFNN with run-time overhead being 60% smaller.

### 5.5 Run-Time Inference Overhead for an Accelerator Implementation

We further investigate the implementation of the proposed FFNN architectures on hardware as an alternative to the micro-controller execution in Sect. 5.4. The simplicity of the FFNN architectures facilitates their synthesis as computation graphs on FPGA fabric. This follows a dataflow-style architecture, where all the weights remain on-chip while the activations flow through the network. The compute graph is pipelined, therefore successive inputs can be processed in different parts of the graph at any point in time. This leads to large improvements in latency and throughput, but requires more on-chip memory to hold the intermediate results between the layers, as well as the weights of the FFNN. The implemented designs follow the method proposed in [30], but offer a higher numerical precision of 8-bit for weights and activations.

The resulting inference rates and FPGA resource usage numbers are given in Table 6. The FPGA 6-input LUTs can be very conservatively translated to a transistor count of 3.8 million transistors per 10,000 LUTs, assuming worst case truth tables and no optimization of the logic functions. Of the different FFNN architectures, the *multi-x* architectures offer significantly higher inference rates and simultaneously lower compute logic and memory usage compared to the *shallow*, *mid-sized*, *deep* architectures.

When comparing the overheads of the possible on-chip micro-controller implementation in Sect. 5.4 with the FFNN accelerator approach, we observe that the accelerator approach for the *multi-x* architectures offers approximately a 100 times higher inference rate at a cost of approximately 20 times more transistors,

**Table 6** FPGA resource usage, latency and maximum inference rates for an accelerator implementation

| Model | LUTs | Memory in kBit | Latency in us | Inference Rate in kHz |
|---|---|---|---|---|
| Single-objective optimization | | | | |
| Shallow FFNN | 15,763 | 88 | 6.3 | 158 |
| Mid-sized FFNN | 21,937 | 111 | 17.8 | 56 |
| Deep FFNN | 32,810 | 156 | 12.3 | 81 |
| Multi-objective optimization | | | | |
| Multi-1 FFNN | 18,622 | 69 | 1.3 | 769 |
| Multi-2 FFNN | 17,272 | 49 | 0.9 | 1,064 |
| Multi-3 FFNN | 17,148 | 49 | 0.7 | 1,389 |
| Multi-4 FFNN | 17,151 | 48 | 1.1 | 893 |

under worst case transistor usage assumptions. Due to the very high inference rate of the accelerator a single accelerator could be used for a 100-core processor computing the power estimations for all cores at an effective per-core inference rate of 10 kHz. This would translate to 5 times lower area overhead compared to the per-core micro-controller implementation. Also, future very large core architectures might require higher power estimation rates than today's multi-core processors, making such an accelerator implementation on a per-core/compute tile/chiplet level even more advantageous.

### 5.6 Estimation Accuracy on Holdout Data

The hyperparameters gained from the single-objective optimization and shown in Table 3 are used to generate the first three final FFNNs (*shallow*, *mid-sized*, *deep*) using the second round of tenfold cross validation [26]. Four additional FFNNs (*multi-1/2/3/4*) are generated in the same way based on the Pareto optimal hyperparameters gained from the multi-objective optimization from Table 4. Note that for actual deployment in a multicore processor, one would only generate either the *shallow* FFNN or one of the four *multi-x* FFNNs as they all occupy the same Pareto front with the *shallow* FFNN having lowest error values and the other FFNNs having higher error values but lower overhead.

However, for providing an extensive performance overview and analysis we also show the performance of the *mid-sized* and *deep* FFNN. The estimation accuracy of all seven resulting FFNNs is then determined on the holdout data, i.e. data that has neither been used for the hyperparametrization nor for training the FFNNs. Table 7 shows the RMSE and percentage errors of the seven FFNNs, the model linear and the multivariate polynomial model as a comparison.

From the FFNNs, the *shallow* one has the best estimation performance with the remaining six FFNNs having worse error as was to be expected from the previous validation data results. Compared to the state-of-the-art linear model [2], we observe a decrease in the relative error of 7.5% *shallow* FFNN. Such an estimation improvement can be significant for both short-term power (density) management and long-term thermal management. For example, an overestimation of the power

**Table 7** Estimation accuracy of FFNNs, linear model, and polynomial model on the holdout data set

| Model | RMSE | Relative Error | MAPE |
|---|---|---|---|
| Single-objective optimization | | | |
| Shallow FFNN | 0.26 W | 4.5% | 5.4% |
| Mid-sized FFNN | 0.50 W | 8.4% | 8.0% |
| Deep FFNN | 0.40 W | 6.8% | 7.0% |
| Multi-objective optimization | | | |
| Multi-1 FFNN | 0.31 W | 5.3% | 5.9% |
| Multi-2 FFNN | 0.43 W | 7.2% | 7.7% |
| Multi-3 FFNN | 0.47 W | 7.9% | 8.2% |
| Multi-4 FFNN | 0.36 W | 6.1% | 6.0% |
| Reference | | | |
| Linear model [2] | 0.75 W | 12% | 12% |
| Polynomial model | 0.60 W | 10% | 11% |

consumption of 7.5% over a time range of ten milliseconds can lead to power-inefficient mapping and scheduling of tasks or an early end to frequency boosting by the power manager.

The best polynomial model had an RMSE of 0.01W on the validation data but an RMSE of 0.60W on the holdout data which is not significantly better than the performance of the linear model. Compared to the polynomial model, the relative estimation error of the *shallow* FFNN power estimator still decreases by 5.5%. The result of the best FFNNs having two or three layers as well as the best-suited polynomial order being 2, we interpret as the underlying non-linear performance counter/power relationship in itself being probably not overly complex. The comparatively bad polynomial model performance reconfirms the findings of McCullough et al. [19] that polynomial regression modeling very easily overfits the underlying performance/power data and is not well-suited to capture the non-linear relationships.

The *multi-x* FFNN models perform between 0.8% to 3.4% worse concerning the relative error compared to the *shallow* FFNN but all perform still significantly better than the linear and polynomial models.

In addition to the RMSE values and relative error values compared to average core power, we also provide the mean absolute percentage error (MAPE) values in the final comparison shown in Table 7. Compared to the relative error, MAPE penalizes underestimations of the core power consumption stronger than overestimations of core power which could be advantageous for conservative power management algorithms. However, we do not observe significant qualitative differences between the relative error values and the MAPE values and have included MAPE for sake of completeness.

On the final choice of which power models to use in a multicore processor, all presented power models come with a trade-off between modeling accuracy and runtime overhead. If accuracy is the most important objective for the processor designer, the *shallow* FFNN would be a natural choice. If a slight degradation of 0.8% in accuracy is acceptable, the *multi-1* FFNN would allow for 61% lower run-

time overhead compared to the *shallow* FFNN. If run-time overhead is the most important objective, one would probably still use the linear model.

## 6 Conclusion

In this paper, we extended our previous work on the use of FFNNs for fine-grained run-time power estimation on core-level with an estimation rate of 10 kHz. We proposed a multi-objective metaheuristic approach that aims to minimize both the modeling error as well as the run-time overhead. For this, we adapted NSGA-II for optimizing the neural architecture hyperparameters, i.e. number of layers and number of neurons per layer. Similar to our previous work, we used tenfold cross validations to avoid both an underfitting of the non-linear relations between performance counters and dynamic power as well as to avoid overfitting the training data while using holdout data to assess the accuracy of the final Pareto optimal FFNNs.

Our results show that the best FFNN from our previous work occupies the same Pareto front as the best FFNNs from our multi-objective optimization approach. Thus, our multi-objective optimization approach offers additional neural architectures with clear trade-offs between accuracy and run-time overhead for multicore power modeling. The highest accuracy FFNN from our previous work has in comparison to the newly generated FFNNs still a higher accuracy, however, the additional FFNNs from this work offer on average 60% lower overhead while still outperforming state-of-the-art linear and polynomial modeling approaches. Overall, the optimized FFNNs have between 4.1% and 7.5% lower relative error compared to a linear model and 2.1% and 5.5% lower relative error compared to a multivariate polynomial regression model. These improvements in accuracy can translate to improved multicore power and thermal management decisions, e.g. power-density aware task mappings and frequency boosting, by avoiding unnecessarily inaccurate power information input to these algorithms.

We show that a micro-controller-based implementation would allow the FFNN inference/power estimation for all of our optimized FFNN architectures to be executed at 10 kHz on each core with a maximum area overhead of 0.25% for large out-of-order cores. The run-time overhead optimized FFNN architectures leave more headroom for additional, e.g. power and thermal management, tasks on the proposed micro-controller. Finally, we provide an accelerator-based implementation which could further decrease area overhead by a factor of 5 for a many-core system or allow for significantly higher estimation rates of up to 1.38 MHz.

**Acknowledgements** This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Projektnummer 146371743-TRR 89 "Invasive Computing".

**Funding** Open Access funding enabled and organized by Projekt DEAL.

## References

1. ARM Limited: Cortex-M0 technical reference manual. Technical report (2009)
2. Bertran, R., Gonzelez, M., Martorell, X., Navarro, N., Ayguade, E.: A systematic methodology to generate decomposable and responsiv e power models for CMPs. IEEE Trans. Comput. (2013)
3. Bienia, C.: Benchmarking modern multiprocessors (2011)
4. Bircher, W.L., John, L.K.: Complete system power estimation using processor performance events. IEEE Trans. Comput. (2012)
5. Carlson, T.E., Heirman, W., Eyerman, S., Hur, I., Eeckhout, L.: An evaluation of high-level mechanistic core models. ACM TACO (2014)
6. Chadha, M., Ilsche, T., Bielert, M., Nagel, W.E.: A statistical approach to power estimation for x86 processors. In: Proceedings of the 2017 IEEE 31st international parallel and distributed processing symposium workshops, IPDPSW 2017 (2017)
7. Chen, X., Arbor, A., Dick, R.P., Mao, Z.M.: Performance and power modeling in a multi-programmed multi-core environment pp. 813–818 (2010)
8. Chu, X., Zhang, B., Xu, R.: Multi-objective reinforced evolution in mobile neural architecture search. In: Bartoli, A., Fusiello, A. (eds.) Computer Vision: ECCV 2020 Workshops, pp. 99–113. Springer, Cham (2020)
9. Cupertino, L.F., Da Costa, G., Pierson, J.M.: Towards a generic power estimator. Comput. Sci. Res. Develop. (2014)
10. Deb, K., Agrawal, R.B.: Simulated binary crossover for continuous search space. Technical report (1994)
11. Deb, K., Agrawal, S.: A niched-penalty approach for constraint handling in genetic algorithms. In: Artificial Neural Nets and Genetic Algorithms, pp. 235–243. Springer, Vienna (1999)
12. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-II. IEEE Trans. Evolut. Comput. **6**(2), 182–197 (2002). https://doi.org/10.1109/4235.996017
13. Huang, G.B., Chen, L., Siew, C.K.: Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Trans. Neural Netw. (2006)
14. Huang, W., Lefurgy, C., Kuk, W., Buyuktosunoglu, A., Floyd, M., Rajamani, K., Allen-Ware, M., Brock, B.: Accurate fine-grained processor power proxies. In: IEEE/ACM MICRO (2012)
15. Kim, Y., Mercati, P., More, A., Shriver, E., Rosing, T.: P4: Phase-based power/performance prediction of heterogeneous systems via neural networks. IEEE/ACM ICCAD (2017)
16. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: IEEE MICRO (2009)
17. Lin, W., Wu, G., Wang, X., Li, K.: An artificial neural network approach to power consumption model construction for servers in cloud data centers. IEEE Trans. Sustain. Comput. (2019)
18. Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., Banzhaf, W.: Nsga-net: neural architecture search using multi-objective genetic algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '19, pp. 419–427. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3321707.3321729
19. McCullough, J.C., Agarwal, Y., Chandrashekar, J., Kuppuswamy, S., Snoeren, A.C., Gupta, R.K., Diego, U.C.S., Labs, I.: Evaluating the effectiveness of model-based power characterization. Usenix Atc (2011)
20. Möbius, C., Dargie, W., Schill, A.: Power consumption estimation models for processors, virtual machines, and servers. IEEE TPDS (2014)
21. Pathania, A., Henkel, J.: HotSniper: Sniper-based toolchain for many-core thermal simulations in open systems. IEEE Embedd. Syst. Lett. (2019)

22. Rapp, M., Pathania, A., Mitra, T., Henkel, J.: Prediction-based task migration on S-NUCA many-cores. In: DATE (2019)
23. Rapp, M., Sagi, M., Pathania, A., Herkersdorf, A., Henkel, J.: Power- and cache-aware task mapping with dynamic power budgeting for many-cores. IEEE Trans. Comput. (2019)
24. Rapp, M., Sagi, M., Pathania, A., Herkersdorf, A., Henkel, J.: Power- and cache-aware task mapping with dynamic power budgeting for many-cores. IEEE Trans. Comput. **69**(1), 1–13 (2020). https://doi.org/10.1109/TC.2019.2935446
25. Rethinagiri, S.K., Palomar, O., Ben Atitallah, R., Niar, S., Unsal, O., Kestelman, A.C.: System-level power estimation tool for embedded processor based platforms. In: ACM RAPIDO (2014)
26. Sagi, M., Vu Doan, N.A., Fasfous, N., Wild, T., Herkersdorf, A.: Fine-grained power modeling of multicore processors using ffnns. In: Orailoglu, A., Jung, M., Reichenbach, M. (eds.) Embedded Computer Systems: Architectures, Modeling, and Simulation, pp. 186–199. Springer, Cham (2020)
27. Samei, Y., Dömer, R.: Automated estimation of power consumption for rapid system level design. In: IEEE IPCCC (2014)
28. Shahid, A., Fahad, M., Manumachu, R.R., Lastovetsky, A.: Improving the accuracy of energy predictive models for multicore cpus using additivity of performance monitoring counters. In: Parallel Computing Technologies (2019)
29. Su, B., Gu, J., Shen, L., Huang, W., Greathouse, J.L., Wang, Z.: Ppep: online performance, power, and energy prediction framework and dvfs space exploration. In: IEEE/ACM MICRO (2014)
30. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P., Jahre, M., Vissers, K.: Finn: A framework for fast, scalable binarized neural network inference. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, pp. 65–74. ACM (2017)
31. Vidnerová, P., Neruda, R.: Multi-objective evolution for deep neural network architecture search. In: Yang, H., Pasupa, K., Leung, A.C.S., Kwok, J.T., Chan, J.H., King, I. (eds.) Neural Information Processing, pp. 270–281. Springer, Cham (2020)
32. Walker, M.J., Diestelhorst, S., Hansson, A., Das, A.K., Yang, S., Al-Hashimi, B.M., Merrett, G.V.: Accurate and stable run-time power modeling for mobile and embedded CPUs. IEEE TCAD (2017)
33. Woof, S.C., Ohara, M., Torriet, E.: The Splash-2 programs: characterization and methodological considerations. In: ACM ISCA (1995)
34. Wu, W., Lin, W., He, L., Wu, G., Hsu, C.H.: A power consumption model for cloud servers based on elman neural network. IEEE Trans. Cloud Comput. (2019)



## Authors and Affiliations

**Mark Sagi[1] · Nguyen Anh Vu Doan[1] · Nael Fasfous[1] · Thomas Wild[1] · Andreas Herkersdorf[1]**

✉ Mark Sagi
mark.sagi@tum.de

Nguyen Anh Vu Doan
anhvu.doan@tum.de

Nael Fasfous
nael.fasfous@tum.de

Thomas Wild
thomas.wild@tum.de

Andreas Herkersdorf
herkersdorf@tum.de

[1] Technical University of Munich, Munich, Germany