Energy-Demand Estimation of Embedded Devices Using Deep Artificial Neural Networks

Timo Hönig, Benedict Herzog, and Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
{thoenig,benedict.herzog,wosch}@cs.fau.de

ABSTRACT

The need for high performance in embedded devices grows at a breathtaking pace. Embedded processors that satisfy the hunger for superlative processing power share a common issue: the increasing performance leads to growing energy demands during operation. As energy remains a limited resource to embedded devices, it is critical to optimise software components for low power. Low-power software needs energy models which, however, are increasingly difficult to create as to the complexity of today's devices.

In this paper we present a black-box approach to construct precise energy models for complex hardware devices. We apply machine-learning techniques in combination with fully automatic energy measurements and evaluate our approach with an ARM Cortex platform. We show that our system estimates the energy demand of program code with a mean percentage error of 1.8 % compared to the results of energy measurements.

CCS CONCEPTS

• **Hardware → Power estimation and optimization**; • **Computing methodologies** → *Machine learning*; *Neural networks*; • **Computer systems organization** → *Embedded systems*; *Embedded hardware*; *Embedded software*;

KEYWORDS

Energy Demand Analysis, Machine Learning, Embedded Systems

ACM Reference Format:
Timo Hönig, Benedict Herzog, and Wolfgang Schröder-Preikschat . 2019. Energy-Demand Estimation of Embedded Devices Using, Deep Artificial Neural Networks. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297338

1 INTRODUCTION

The demand for increasing processing power in embedded devices is at an on-going, breathtaking pace [3, 29]. New fields of application in professional work environments and private spheres ask for incredible processing power right at the data source (e.g., CMOS image sensors). New embedded processors designs which satisfy this hunger for superlative processing power share a common issue: the increasing computation performance comes with extreme energy demands during operation [9]. The urgently needed revolution in energy storage technologies, however, never happened [14]. This led to desperate efforts, such as cramming lithium-ion batteries into tiny enclosures with catastrophic results, for example, tens of thousands of dangerous explosions [27].

Instead of waiting for a revolution in energy storage technologies, researchers actively explore new ways of increasing the energy yield. To exploit available energy resources in the greatest extent possible, recent research has focused on improving the energy demand of system software and applications [3, 17]. In this field, energy models [21, 25] are essential for program-code analysis and low-power optimisations as energy models estimate the energy demand of program code for specific hardware platforms [25]. With the increasing complexity of processor designs [5], however, it becomes highly difficult to actually create energy models in general, that provide accurate energy demand estimates in particular. For example, recent (heterogeneous) multi-core processors that are common for embedded devices (i.e., IoT devices, smart phones) have up to eight general-purpose processor cores (e.g., ARM Cortex A73), they feature pipelines with no less than 15 stages, two-way branch prediction, and out-of-order execution.

Semiconductor manufacturers are very thin-lipped about the *specific* power and energy characteristics of their CMOS chips. It is common for semiconductor companies to provide examples how the hardware designs operate under ideal conditions for a few test cases, only. This lack of information leads to the need of applying reverse engineering techniques to understand the electrical properties of individual CMOS devices [26]. The engineering approach to reversely explore the power and energy characteristics entails expansive and manual measurements. And yet, due to the complexity of today's processors and hardware platforms, such methods of analysis yield incomplete results, only. Each individual hardware feature (e.g., instruction-level parallelism, multi-staged pipelines, out-of-order execution, multi-level caches, and so forth) contributes to the complexity of hardware devices and their individual energy-demand characteristics, and consequently increases the work required to establish robust energy models. In the light of the rising number of different platforms the effort becomes bigger and bigger and one has to ask the question: can we invest the effort to build a energy model for every new CMOS device at all?

In this paper, we present an approach to address the challenge of creating precise energy models even for complex hardware platforms. Our contribution applies machine-learning techniques to cope with the complexity of today's CMOS devices and analyses the hardware in a black-box approach during the construction of energy models. The contributions of this paper are threefold. First, we present a systems approach to tackle the challenge of constructing

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297338

precise energy models even for modern, complex CMOS devices. Second, we discuss the implementation of our approach which exploits machine–learning techniques. Our implementation runs automated energy measurements for the training of deep artificial neural networks which subsequently predict the energy demand of program code. Third, we evaluate our approach at the example of an ARM Cortex platform and compare energy demand estimations of our generated energy models with a state-of-the-art energy modelling technique which requires manual labour.

The remainder of this paper is structured as follows: Section 2 discusses the current state of the art of energy-demand analysis and presents knowledge on artificial neural networks. In Section 3 we outline the system architecture and present the implementation of our approach. In Section 4 we evaluate our current implementation and discuss the evaluation results. Related work and prior research is presented in Section 5, and Section 6 concludes the paper.

2 BACKGROUND

This section discusses the current status of energy-demand analysis of computing systems (i.e., energy models at different levels of abstraction) and presents background on (deep) artificial neural networks, specifically fully-connected feed-forward neural networks.

2.1 Energy-Demand Analysis

The energy-demand analysis of embedded and power-constraint computing systems is done at various levels of abstraction and depends on the individual use case. Thus, energy-demand analysis is either performed with little or no abstractions at hardware level through to higher abstractions at software level. In consequence to the varying degree of abstraction of the individual energy-demand analysis approaches, respective energy models are established at different levels of abstraction. Energy models at low levels of abstraction (i.e., hardware energy-models) provide precise energy-demand estimates, but at the same time they are pinned down to a certain hardware platform. In contrast to this, energy models at a higher level of abstraction (i.e., software energy-models) commonly trade off precision against generality and thus enable energy-demand analysis of software independent from a specific hardware platform.

At the hardware level, instruction-based energy models [21, 25] describe the energy demand for executing individual CPU instructions. At a higher level of abstraction, energy models with the granularity of basic blocks[1] are used to include inter-instruction effects [25] on the energy demand. For embedded devices, such energy models are suitable as they exactly characterise the energy demand of the underlying hardware platform [18, 22]. In particular, energy models with the granularity of basic blocks are suited to bridge the gap towards higher abstraction levels (i.e., function- or process level). Energy models for wireless links and networks [20] use packet-based energy models. State-based energy models and approaches which consider the energy demand of components after use, so-called tail energy [1], considerably improve the results as asynchronous system activities are tracked across different layers (i.e., software, hardware). The energy efficiency of software components often is described at function-level [8] or process-level [28]. Recent works that analyse the energy demand of embedded operating-systems [13] and mobile applications [2] consider the energy demand at system- and platform-level.

[1] A *basic block* is a sequence of CPU instructions that contains a single entry point and has no branches except at the exit.

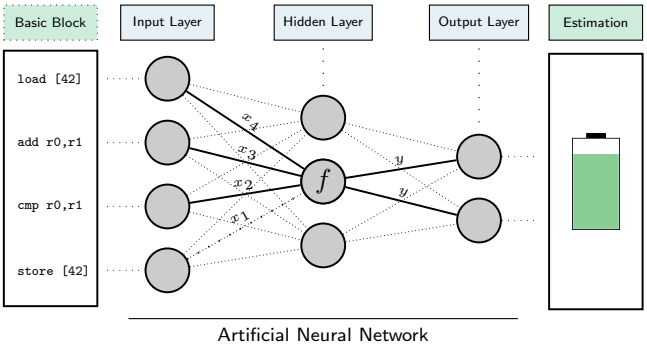

Figure 1: An artificial neural network consists of neurons (arranged in layers) and links between neurons. The output y of a neuron is calculated by applying the weighted sum of all input values x_i to an activation function f.

With the increasing number of functionalities that are implemented by embedded devices, the complexity of the individual hardware components (i.e., CPU, GPU, memory, wireless network) is growing steadily, too. The rising complexity at component-level and platform-level hence requires adequately complex energy-models which in turn influence the energy-demand analysis at all levels of abstraction. To reduce the engineering efforts for generating energy models at component-level and platform-level we use machine-learning techniques which exploit artificial neural networks to explore energy-demand characteristics in a black-box approach.

2.2 Artificial Neural Networks

Machine learning techniques have been successfully applied in various fields of research and application (e.g., text or image recognition, classification, and processing), especially over the past few years.

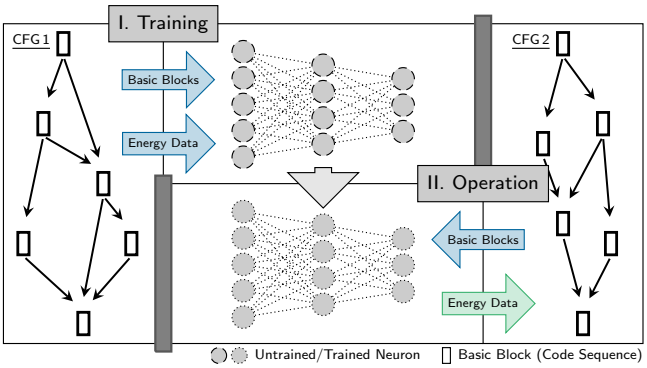

Figure 2: Our system uses basic blocks to train an artificial neural network to learn energy characteristics of an unknown hardware platform (I. Training). Subsequently, the trained network calculates the energy demand estimates for unknown inputs (II. Operation).

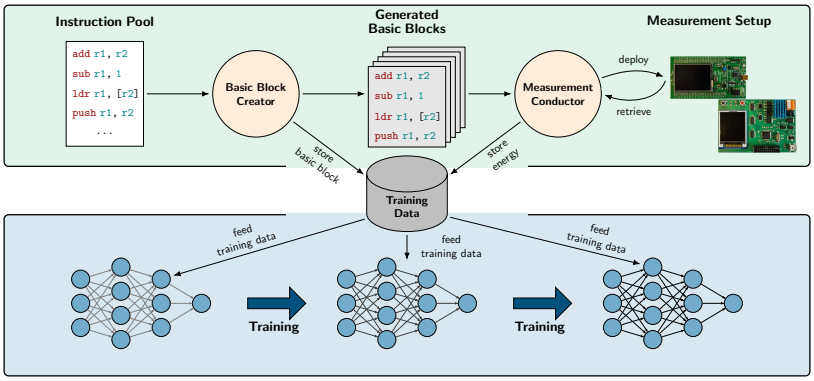

Figure 3: Illustration of the creation of an energy model based on basic blocks and an artificial neural network. First, basic blocks are derived from basic blocks of applications, generated, and automatically deployed on the device under test (DUT). Automatic energy measurements are used to determine the energy demand of each basic block. Both information, the structure of the basic block and its energy demand, are stored in a database. Subsequently, the database is used to train an artificial neural network to estimate the energy demand of basic blocks.

In particular, (deep) artificial neural networks currently experience a big renaissance [10, 12]. With increasing processing resources of today's computing systems such techniques can not only be used for classical machine learning problems [16], but to process complex tasks solving extraordinarily complex problems [19].

Artificial neural networks (ANN) can be considered as a simplified version of biological neural networks. They have great capabilities in learning patterns from data and creating robust models, even if the underlying data contain noise or outliers. Figure 1 shows an example for an ANN. The network consists of neurons and links between neurons. Every neuron has various input and output links from and to other neurons. Usually, neurons are arranged in layers, where every layer can have an arbitrary number of neurons and every neuron of a layer is connected with every neuron of the previous layer. Hence, such networks are called *fully-connected feed-forward neural networks*, as there are no cyclic dependencies and all neurons between two adjacent layers are connected.

The input is fed to a network through the *input layer* and passes through a network-specific number of so-called *hidden layers*. A single neuron calculates its output y by attaching a weight on each input x_i and adding up all weighted input values. The calculated sum is used as an input for an activation function f, which calculates the output y of the neuron. One typical activation function is a *rectifier function*, which is a positive ramp function and widely used in deep artificial neural networks. This output y of a neuron is either input for other neurons or it is one of the output values of the network, if the neuron it is part of the *output layer*.

A network is trained by comparing the output of the network with a label (i.e., the desired output), both specific for one input. Subsequently, the weights of the neurons are adjusted to reduce the distance (i.e., difference or error) between the label and the output of the network by a backward propagation of errors [15]. If the training data set is sufficiently large, the network learns the underlying principles for its model. Subsequent to its training, a network makes accurate predictions for unknown inputs, which have not been part of the training set.

The depth of a network describes the number of hidden layers, which is an important factor for the capability to learn complex features. Usually, the closer a hidden layer is to the output layer, the more abstract are learned features. A specific layer uses the inputs as provided from the previous layer (i.e., raw input or preceding hidden layer) to generate more abstract features. Starting from raw input with every layer the abstraction level rises up to the final output of the network. Networks with a large number of hidden layers, so-called deep artificial neural networks (DANN), provide strong results, especially in fields of image and speech processing [10, 12].

With a growing number of hidden layers the training of the network becomes increasingly complex and the required computation resources rise as the back-propagation algorithm needs to be executed for more layers. Therefore, the ideal shape of a network must consist of enough layers to respect the complexity of the problem to be solved and simultaneously be small enough to be trained using a reasonable amount of computation resources.

3 SYSTEM ARCHITECTURE

In this section we present the system architecture and the implementation of our prototype. We use deep artificial neural networks to model the energy demand of code sequences at basic-block level and we measure the energy demand on a development system to generate input data for training a neural network.

3.1 Overview

Figure 2 illustrates the general approach to utilise an artificial neural network for estimating the energy demand of applications which are composed of basic blocks. First, basic blocks are either generated or they are extracted from call function graphs (CFGs), for example, by applying tool support from the compiler (e.g., LLVM) and used to train the neural network (I. Training). After training, the network is used to estimate the energy demand for basic blocks, which, in sum, resemble the structure of CFGs, and that are different from the ones used during the training phase of the neural network and hence estimate the energy demand for unknown software (II. Operation).

The architecture of our systems relies on two main components. First, our approach relies on an energy measurement device which performs automated energy measurements of different code sequences (i.e., thousands of different basic blocks). Second, our approach uses an artificial neural network to learn energy characteristics which are specific to the device under test (DUT) while executing the sequences of code. Subsequent to its training, the network can estimate the energy demand of unknown sequences of code. Figure 3 shows an overview of our system architecture.

Initially, we generate a sufficient amount of data for the training phase of the artificial neural network (cf. Section 3.2). The training data consists of different basic blocks that are derived from basic blocks of applications and their respective energy demands. The energy demands are retrieved by automated energy measurements of the individual basic blocks that are performed on the DUT (cf. Section 3.4). Both information, that is, the basic block and the measured energy demand, are stored in a database.

Subsequently, the database is used to train our system with energy-demand data of the different code sequences (cf. Section 3.3). We choose to train our system at the level of basic blocks (code sequence with no branches) to see whether our trained network can model non-deterministic effects (e.g., cache misses) and inter-instruction effects. During operation and after its training, the neural network estimates the energy demand of basic blocks which have *not* been part of the training set. Hence, the network can be used to estimate the energy demand for a multitude of different basic blocks, which, in sum, resemble the structure of an application.

3.2 Training Data Generation

In order to obtain a sufficient amount of data, we implemented a tool to derive basic blocks from applications. The structure of typical basic blocks depend greatly on the utilised tool chain and its parameters (e.g., compiler, assembler, and optimisation level) and the structure of the software (e.g., focus on memory accesses or data processing). Figure 4 shows an example of the same basic block and compiler with different optimisation levels. The basic blocks differ in utilised instructions types, number of instructions, and complexity. Based on the analysis of basic blocks of software running on our DUT, our tool derives new basic blocks, which are comparable to basic blocks as extracted from applications.

The instruction pool, from which instructions are selected, contains all types of instructions, such as, data processing instructions (e.g., `add`), data comparison instructions (e.g., `cmp`), and memory instructions (e.g., `load`/`store`). Furthermore, the instruction pool contains different addressing modes (i.e., immediate values, register and, in case of load/store instructions, memory locations).

Every basic block consists of a various number of instructions of the instruction pool. The only constraint for basic block generation is the placement of push and pop instructions, which need to be placed in a well-formed order (i.e., the same number of both instructions and for every pop instruction a push instruction must be executed beforehand) for stack-integrity reasons. We assign a unique hash value to each basic block, which we test before including the basic block in the data set, to avoid identical basic blocks. The input for the hash function are the mnemonics of the assembler instructions representing the basic block.

Optimisation 00

```
main.entry:
push r7, lr
mov r7, sp
sub sp, 16
movs r0, 0
str r0, [sp,12]
str r0, [sp,4]
movs r1, 5
str r1, [sp,0]
str r0, [sp,8]
b.n label
```

Optimisation 03

```
main.entry:
sub sp, 4
movs r0, 5
str r0, [sp,0]
ldr r0, [sp,0]
cmp r0, 0
it lt
movlt r0, 0
addlt sp, 4
bxlt lr
```

Optimisation 0s

```
main.entry:
sub sp, 4
movs r0, 5
str r0, [sp,0]
ldr r0, [sp,0]
blt.n label
```

Figure 4: Entry basic block of the `main()` function compiled at different optimisation levels. The basic blocks differ in utilised instruction types, number of instructions, and complexity.

Each basic block is embedded in a test framework and deployed on our DUT. The test framework executes each individual basic block in a loop and simultaneously conducts an energy measurement, which measures the energy demand during execution. The energy measurement is started an stopped by a single instruction directly before and after the execution of the basic block under test, respectively, in order to maximise the precision of the measurements. These measurements represent the energy demand of the basic blocks on the DUT and the measurement data is basis for the neural network to draw conclusions about the energy demand of unknown basic blocks (i.e., basic blocks which are not included in the training set).

3.3 Energy Model

To estimate the energy demand of a basic block, we feed its instructions into the deep artificial neural network as an input value. Feed-forward artificial neural networks expect, by design, a fixed number of input values. In contrast, basic blocks have varying numbers of instructions per basic block. To meet the design of the neural network we developed a two-staged process. First, every basic block is normalised to a fixed size by adding `null` instructions (an instruction not included in the instruction set architecture of the processor, but indicating the neural network that this is not a real instruction). The fixed size is chosen sufficiently large to cover almost all basic blocks size usually created during the compilation of software. However, basic blocks exceeding this size can be split

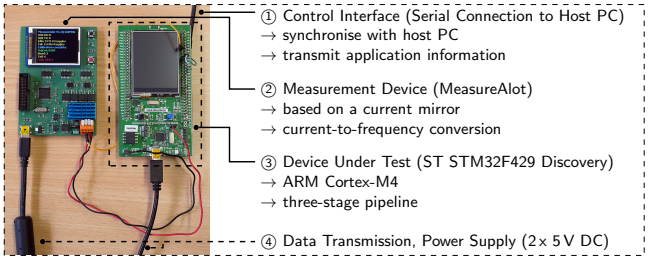

Figure 5: Our evaluation setup comprises a measurement device (left), a device under test (DUT, right), and a host PC (not shown). The neural network learns the energy characteristics of the DUT (dashed box) in a black-box approach.

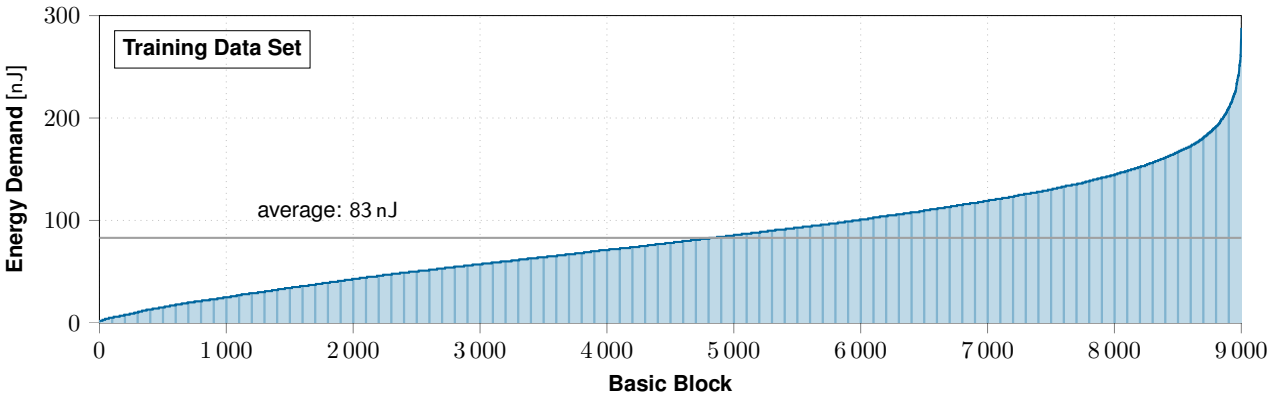

Figure 6: Energy demands of basic blocks included in the training data set ordered by energy demand. The minimum and maximum energy demand equals 2 nJ and 288 nJ, respectively. The average energy demand equals 83 nJ.

into two blocks. Subsequently, every instruction of the basic block is encoded using one bit per possible instruction type and an additional bit for the `null` instruction. For every instruction, the bit representing the type of the instruction is set to one and all other bits are set to zero, which is called a *one-hot* encoding. This allows the artificial neural network to distinguish different instruction types and especially to easily detect and ignore `null` instructions.

The network has a single output value, which represents the estimated energy demand of the basic block. The hidden layers of our network are fully-connected feed-forward layers and the number of neurons per layer varies with the position of the layer. In general, layers near the output layer have less neurons than layers near the input layer. This respects the rising abstraction level and leads to a single output value.

Given that our input and output features contain no hardware specific data, but only (a) the structure of the basic blocks (i.e., the instructions of each basic block) and (b) the individual energy demand for each basic block during execution on the DUT, our network can be considered as a black-box model. Thus, we put no engineering effort into modelling hardware specific details. The model automatically extracts and learns hardware specific details from the provided training data. This makes our approach very convenient to be used with arbitrary hardware platforms, by only substituting the training data set and retraining the network for the different platform (i.e., new DUT), without a need to change the general neural network architecture. For a DUT utilising a different instruction set architecture, only the basic block normalisation and the input layer of the neural network must respect the changed number of instruction types. The rest of the architecture remains unchanged.

For the evaluation, we split our data in three data sets (i.e., a training set, a validation set, and a test set). The training data set is used to train our model to estimate the energy demand of basic blocks. Therefore, the basic blocks are fed into the neural network as input values and the measured energy demands are provided as expected output values and the network adapts its weights in way that the inputs lead to the expected outputs. During the training phase, the validation data set is used to continuously evaluate the quality of our model, by estimating the energy demand of basic blocks from the validation data set, which are not part of the training set, hence unknown to our model, and compare the estimation with the actual energy demand. This allows an continuous evaluation of the training progress and to countervail overfitting and optimise training parameters, respectively. To avoid indirect overfitting to the validation data set, the test data set is finally used once to evaluate the quality of the final artificial neural network.

3.4 Implementation

To generate our training, validation, and test data sets, we performed energy measurements on an ARM platform, namely the STM32F429I-DISC1 Discovery Board from STMicroelectronics. The board powers a 32-bit ARM Cortex-M4 processor which has a three-stage pipeline and serves as DUT (cf. Figure 5) during our evaluation (cf. Section 4). The layout of the board allows to directly measure the energy demand of the processor, without interferences from other circuitry parts of the board.

To perform energy measurements of the processor, we use a high-precision measurement device, the MeasureAlot[2]. It runs energy measurements by exploiting a current mirror [11] and offers a trigger mechanism which implements a fine-grained signalling of the start and end of energy measurements. The DUT controls the measurement device (i.e., sending control signals to start and stop energy measurements) and a host PC retrieves measurement results, automatically. The setup allows us to perform a high number of energy measurements in an automated manner and without manual efforts by an engineer. Our implementation further interconnects the energy measurements with training of the neural network and evaluates the quality of the trained network by comparing the energy estimations with samples (i.e., energy measurements).

To implement the neural networks for our energy model we use the TensorFlow machine learning framework[3]. TensorFlow provides a high-level API to create, train, and run different types of artificial neural networks. The training of our deep feed-forward neural network is performed on a Linux desktop computer due to the need of sufficient computation power for the training phase. During

[2] www4.cs.fau.de/Research/MeasureAlot/
[3] www.tensorflow.org

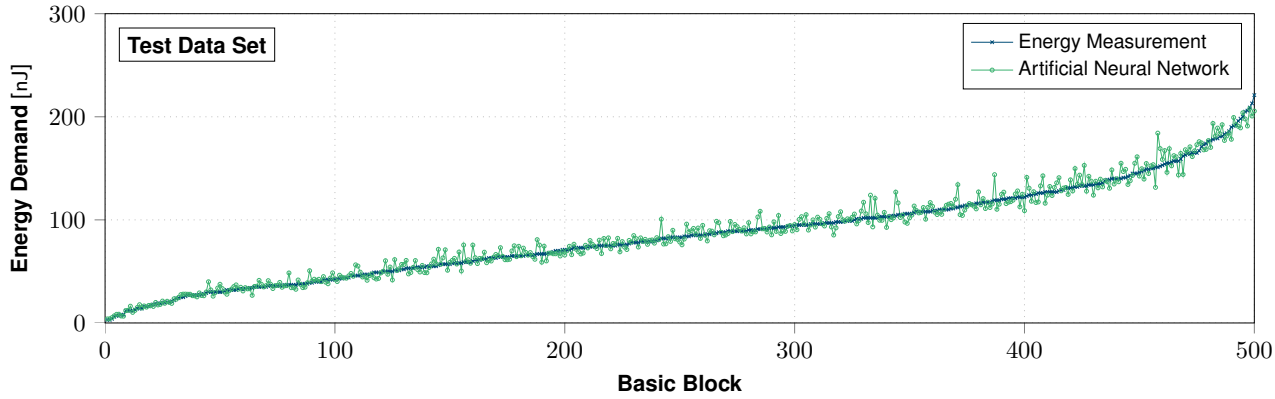

Figure 7: Energy demands for basic blocks included in the test data set ordered by the measured energy demand. Both, the measured energy demand (blue) and the energy demand as estimated by the neural network (green), are shown. The mean percentage error between energy measurements and energy estimations equals 1.8 %.

operation (i.e., estimating the energy demand of basic blocks) the network can be deployed either on the Linux desktop computer or even on the DUT.

For the training of our neural network we assign a unique identifier (UID) to each instruction type and feed the basic blocks into the neural network. To determine the best performing network we tried networks with different topologies, from two up to six hidden layers. The number of neurons per layer is between 512 and 2048 neurons, depending on the position of the layer. After training of the neural networks we compared the training and validation error of these networks and chose the neural network with five hidden layers and a total number of 5633 neurons for our evaluation.

4 EVALUATION

The evaluation of our implementation is performed on a setup consisting of an energy measurement device, the device under test, communication interfaces to the host PC, and power supply connections. The setup is shown in Figure 5.

We have generated and executed 10 000 basic blocks for the evaluation and consequently performed the same number of measurements on the device under test to determine the energy demand of each basic block. These 10 000 basic blocks are split into three data sets, that is, the training, validation, and test data set containing 9000, 500, and 500 basic blocks, respectively. Our instruction pool consists of over a hundred instruction types. The number of instructions per basic block varies from 3 to 328 instructions and covers all typical sizes of basic blocks.

Figure 6 shows the results of the energy measurements sorted by energy demand for the training data set. For all basic blocks part of our training set, we measure an average energy demand of 83 nJ, with a minimum and maximum energy demand of 2 nJ and 288 nJ, respectively. The basic block with lowest energy demand and the basic block with highest energy demand differ by a factor of 144. The varying energy demand of the basic blocks results from the individual energy demand of different instructions, different basic block sizes, diverging addressing modes, and variable inter-instruction energy costs. The main reason for the range of energy demands is the size of the basic blocks, but also the structure of a basic block influences the energy demand. For example, simple data processing instructions working on registers or intermediate values (e.g., `add`, `shift`) usually consume less energy than instructions containing memory accesses (e.g., `ldr`, `pop`) or more complex calculations (e.g., `mul`). An analysis of the energy demands of basic blocks with a fixed number of instructions (i.e., ten instructions) showed, for example, that the number of memory access influences the energy demand of the basic block. A basic block consisting of ten instructions with two memory accesses had an energy demand of 6 nJ, whereas a basic block consisting of ten instructions with seven memory accesses had an energy demand of 16 nJ. Additional energy demand is also induced by inter-instruction effects (e.g., cache and pipeline events) which depend on the specific order of instructions. This leads to strong differences in energy demand for basic blocks, even if the basic blocks contain the same number of instructions. In fact, the average energy demand per instruction in the training data set varies from 0.4 nJ to 1.6 nJ, which confirms our assumption that not only the basic block size, but also the instruction types and inter-instruction effects are important factors.

We further split our data set in three subsets. First, the *training data set* contains 9000 unique basic blocks and is used to train our neural network. Second, the *validation data set* contains additional 500 unique basic blocks. The validation data set serves as a verification for energy demand estimations of the trained neural network during training. Third, a *test data set* of another 500 unique basic blocks is used to run the final evaluation after the training phase. The basic blocks are distributed randomly among all three data sets to avoid any selection bias.

We examined several feed-forward neural networks of individual shape. The best performing neural network consists of five hidden layers with 2048 neurons for the first and second hidden layer, 1024 neurons for third hidden layer, and 512 neurons for the fourth hidden layer. The last layer consists of a single neuron emitting the energy demand estimation. Hence, the neural network consists of 5633 neurons in total. While networks with less than five hidden layers performed worse, networks with more than five hidden

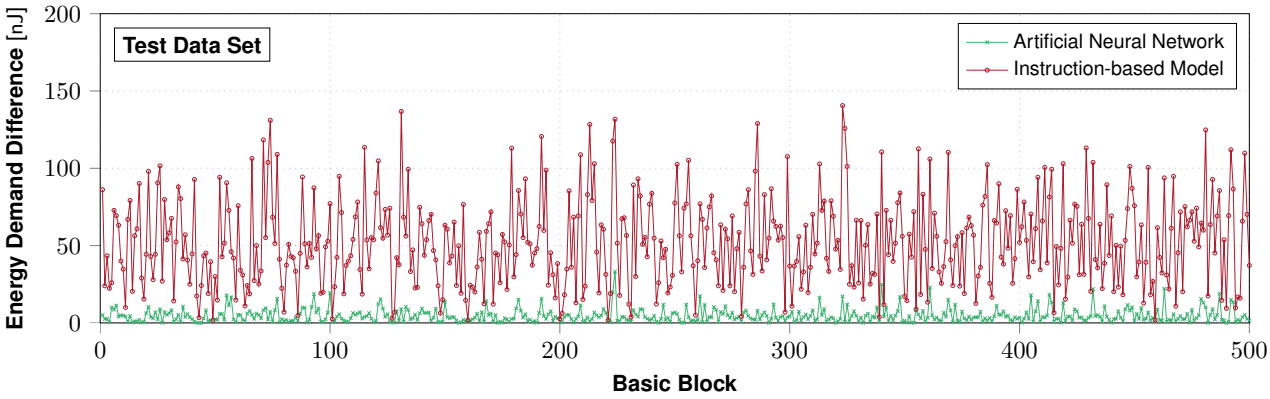

Figure 8: Absolute difference of the energy demand measurements and the estimations made by the artificial neural network (green) and the instruction-based energy model (red), respectively. The estimations of the instruction-based energy model show a significantly higher divergence from the energy measurements.

layers performed similarly well, but required significantly more time for training. Therefore, we chose the neural network with five hidden layers for the remainder of the evaluation. The training was executed on a Intel Core i5 and utilized the adam optimizer[4] with a learning rate of 0.001 and two momentum values of 0.5 and 0.75, respectively. All neurons utilize rectified linear units (ReLUs). For each training step a batch of 2500 basic blocks were fed into the neural network and in total 7400 steps of training were executed.

Out of all possible basic blocks (approximately 10^{199} combinations) our training data set covers a small number, only. However, the evaluation results show that the comparatively small training set is already large enough to properly train the network. After its training, the neural network precisely estimates the energy demand of basic blocks which were *not* included in the training data set. In the evaluation, we verify the energy demand estimations of the neural network by a test data set which contains 500 basic blocks. We validate the energy demand estimates of the trained neural network for each basic block of the test data by comparing the energy demand estimation of the neural network with the actual energy demand of the basic block (i.e., as conducted by measurements).

Figure 7 shows both, the energy demand as measured by the measurement device and the corresponding energy demand estimation of the artificial neural network for all basic blocks included in the test data set. The figure illustrates that the neural network precisely estimates the energy demand for all basic blocks consisting of all instructions types and sizes. The minimal and maximal number of instructions for the test data set was 4 and 244 instructions, respectively. The mean percentage error is 1.8 % compared to energy measurements. Our system yields similar amounts of under- and overestimations (232 and 268). Considering a set of basic blocks (e.g., by extracting basic blocks from a call function graph), the estimation errors compensate themselves quite well.

Furthermore, we compared our approach with a typical instruction-based energy model. Therefore, we determined the average energy demand per instruction using the same evaluation setup as for the energy demand measurements for the basic blocks. Specifically, we executed instructions of all instruction types and measured the energy demand during execution. Subsequently, we divided the energy demand by the number of executed instructions resulting in an average energy demand per instruction of 1.4 nJ. This instruction-based energy model shows significantly worse results compared to our approach utilising an artificial neural network. This is due to the fact, that an instruction-based energy model works on a different level of abstraction (instructions instead of instruction sequences). In fact, the mean percentage error was 64.4 % compared to the 1.8 % shown above for our approach. Figure 8 further illustrates the differences between the estimations made by the artificial neural network and the instruction-based energy model for basic blocks included in the test data set. On the y-axis the absolute difference between the energy demand measurement and the respective energy demand estimations is given. The difference of the instruction-based model between measurement and estimation is for almost all basic blocks significantly higher compared to the estimations made by the artificial neural network. These results show, that the instruction-based model can be used to estimate the order of magnitude of the energy demand of basic blocks, but lacks precision for fine-grained energy analyses. In contrast, our approach based on artificial neural networks is capable of fine-grained energy demand estimates for arbitrary basic blocks.

In sum, the evaluation results demonstrate that our neural networks are well-suited to establish energy models at the level of basic blocks, even for complex hardware platforms. Furthermore, the energy demand estimations are precise enough to be used as base-energy models for energy models at a higher level of abstraction (e.g., function level).

5 RELATED WORK

Machine-learning techniques have been subject of several research works related to the prediction and reduction of computer systems' energy demand. To the best of our knowledge, the work presented in this paper is the first to exploit machine learning techniques to automatically extract precise energy models of CMOS devices and hardware platforms. Early work by Chung et al. [6] explores

[4] https://arxiv.org/abs/1412.6980

the use of adaptive decision trees to predict idle periods at run-time and control strategies of the dynamic power management in order to efficiently power down unused devices (e.g., hard drives). Tesauro et al. present a reinforcement learning approach [23] which uses neural networks to control multiple criteria (i.e., power and performance) of web application servers by throttling CPUs. Further run-time optimisations have been proposed by Berral et al. [4]. The authors propose an energy-aware task-scheduling approach for data centre workloads and analyse the trade-off between power-savings and the fulfilment of service level agreements. At data-centre level, DeepMind [7] demonstrates a reduction of energy use by up to 40 %. The researchers working on DeepMind train deep artificial neural networks at different operating scenarios and parameters to optimise the efficiency of Google's data centres. The authors of [24] propose a prediction scheme which uses artificial neural networks to predict the energy demand of HPC kernels.

Basic blocks as basis for energy demand estimations have been utilised by prior work for energy models. Shnayder et al. [18] deconstruct basic blocks to estimate the number of executed CPU cycles and used simulations to determine execution counters for all basic blocks of an application. The total number of estimated CPU cycles is used to estimate the energy demand. Steinke et al. [22] transfer instructions on basic block level into an energy-efficient scratchpad memory and achieve an energy demand reduction compared to the execution in main memory or in caches. Section 2.1 discusses further research on energy models at abstraction levels other than basic blocks (e.g., instruction and process level).

6 CONCLUSION

In this paper we presented an approach to automatically construct precise energy-models for modern, complex CMOS devices. Our approach exploits deep artificial neural networks for creating energy models of complex hardware in a black box approach. We apply automated energy measurements of code sequences to train a neural network with five layers. Subsequent to its training in the evaluation, the neural network was able to predict the energy demand of unknown code sequences with a mean percentage error of only 1.8 % compared to energy measurements.

ACKNOWLEDGEMENTS

This work was partially supported by the German Research Council (DFG) under grant no. SCHR 603/13-1 ("PAX"), grant no. CRC/TRR 89 ("InvasIC", Project C1), and grant no. SCHR 603/15-1 ("LARN").

REFERENCES

[1] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. 2009. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM Internet Measurement Conference*. 280–293.

[2] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 588–598.

[3] Luiz André Barroso and Urs Hölzle. 2007. The Case for Energy-Proportional Computing. *IEEE Computer* 40, 12 (2007), 33–37.

[4] J. Berral, Ì. Goiri, R. Nou, F. Julià, J. Guitart, R. Gavaldà, and J. Torres. 2010. Towards Energy-aware Scheduling in Data Centers Using Machine Learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. 215–224.

[5] Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (May 2011), 67–77.

[6] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. 1999. Dynamic power management using adaptive learning tree. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design*. 274–279.

[7] DeepMind Technologies Ltd. 2016. DeepMind AI Reduces Google Data Centre Cooling Bill by 40%. Accessed 31.08.2018. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.

[8] Jason Flinn and Mahadev Satyanarayanan. 1999. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the 2nd Workshop on Mobile Computing Systems and Applications*. 2–10.

[9] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. 2016. Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture*. 64–76.

[10] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (Nov 2012), 82–97.

[11] Timo Hönig, Heiko Janker, Christopher Eibel, Oliver Mihelic, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. 2014. Proactive Energy-Aware Programming with PEEK. In *Proceedings of the 2014 USENIX Conference on Timely Results in Operating Systems*. 1–14.

[12] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. 2016. Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification. *ACM Transactions on Graphics* 35, 4 (2016), 110:1–110:11.

[13] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. 2012. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th European Conference on Computer Systems*. 29–42.

[14] Kostas Pentikousis. 2010. In search of energy-efficient mobile networking. *IEEE Communications Magazine* 48, 1 (2010), 95–103.

[15] David Rumelhart, Geoffrey Hinton, and Ronald Williams. 1986. Learning representations by back-propagating errors. *Nature* 323 (1986), 533–536.

[16] Arthur L Samuel. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of research and development* 3, 3 (1959), 210–229.

[17] Eric Saxe. 2010. Power-efficient Software. *Commun. ACM* 53, 2 (Feb. 2010), 44–48.

[18] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. 2004. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd ACM International Conference on Embedded Networked Sensor Systems*. 188–200.

[19] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[20] Suresh Singh, Mike Woo, and Cauligi S Raghavendra. 1998. Power-aware routing in mobile ad hoc networks. In *Proceedings of the 4th Annual ACM/IEEE International conference on Mobile computing and networking*. 181–190.

[21] Amit Sinha and Anantha P Chandrakasan. 2001. JouleTrack: A web based tool for software energy profiling. In *Proceedings of the 38th Annual Design Automation Conference*. 220–225.

[22] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, Mahesh Balakrishnan, and Peter Marwedel. 2002. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th IEEE Symposium on System Synthesis*. 213–218.

[23] Gerald Tesauro, Rajarshi Das, Hoi Chan, Jeffrey Kephart, David Levine, Freeman Rawson, and Charles Lefurgy. 2007. Managing Power Consumption and Performance of Computing Systems Using Reinforcement Learning. In *Advances in Neural Information Processing Systems 20*. 1497–1504.

[24] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely. 2012. Modeling Power and Energy Usage of HPC Kernels. In *Proceedings of the 2012 IEEE International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 990–998.

[25] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. 1994. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration* 2, 4 (1994), 437–445.

[26] Randy Torrance and Dick James. 2011. The State-of-the-art in Semiconductor Reverse Engineering. In *Proceedings of the 48th Design Automation Conference*. 333–338.

[27] Qingsong Wang, Ping Ping, Xuejuan Zhao, Guanquan Chu, Jinhua Sun, and Chunhua Chen. 2012. Thermal runaway caused fire and explosion of lithium ion battery. *Journal of power sources* 208 (2012), 210–224.

[28] Andreas Weissel and Frank Bellosa. 2002. Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 238–246.

[29] Geoffrey Yeap. 2013. Smart mobile SoCs driving the semiconductor industry: Technology trend, challenges and opportunities. In *Proceedings of the 2013 IEEE International Electron Devices Meeting*. 1–8.