Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Processor Frequency Sweet Spot Prediction based on Dynamic Code Analysis

Tobias Schiffmann

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Dr. Christian Simmendinger, Raphael Leiteritz, M.Sc. Gregor Daiß, M.Sc.
Commenced:	September 29, 2021
Completed:	March 29, 2022

Abstract

Supercomputing centers are tackling sustainability for years. Their systems consume a huge amount of electrical power and reducing the consumption only by a couple of percents can have a significant impact on running costs. Currently, the research for energy optimization techniques is in progress and there are promising techniques. However, there is a lack of regression based models which predict the energy-optimal power cap.

In this work two machine learning models are created to predict these values. These models are a random forest regression and a neural network. They are trained using a created dataset which contains values taken from processor counters. The models are optimized during this work and finally used in an experiment to set power caps during a benchmark execution.

The experiment findings show a first trend of energy savings and performance loss for different power capping methods. Executions that use power caps predicted by the regression models are able to save energy. These saving are about 8% for a benchmark while the impact on performance is a loss of about 2%.

Acknowledgement

I'd like to thank M.Sc. Thomas Gruber — one of LIKWID's developers. During the event set creation process I kept in contact with him. He helped me to select the events and their placements on the performance counters.

Thanks a lot to everybody who read this work and provided feedback. I am really grateful that you took the time to support me at writing.

And a great thanks to my supervisors and examiner who made this work possible.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ASHA Asynchronous Successive Halving Algorithm. 13

BDPO Bull Dynamic Power Optimizer. 23

CFD Computational Fluid Dynamics. 9, 35

CPMC Cache L3 Performance Monitoring Counters. 36

CPU Central Processing Unit. 17

CSV Comma-Separated Values. 38

DFC Data Fabric Counters. 36

DNN Deep Neural Network. 29

DVFS Dynamic Voltage and Frequency Scaling. 17

ECP Exascale Computing Project. 34

ELM Extreme Learning Machine. 28

FIXC Fixed Purpose Counters. 36

GEOPM Global Extensible Open Power Manager. 9

HLRS Höchstleistungsrechenzentrum Stuttgart. 34

HPC High Performance Computing. 17

HSMP Host System Management Port. 19

LIKWID Like I Know What I'm Doing. 24

MPI Message Passing Interface. 35

MSR model-specific register. 19

NPB NAS Parallel Benchmarks. 34

PMC Performance Monitoring Counters. 36

RAPL Running Average Power Limit. 19

REST Runtime Energy Saving Technology. 22

1 Introduction

Sustainability is growing in importance nowadays not only in society. Supercomputing centers also have been tackling this topic for decades as this paper shows [FC07]. Several efforts are conducted to motivate improvements in supercomputing in this area, for example the Green500 list. It is a ranking of the most energy efficient supercomputers. [SDSM21] These supercomputers are consuming a huge amount of electrical powers. Reducing the consumption only by a couple of percents can have a significant impact on running costs. The *Hawk* supercomputer, ranked 66 in the Green500 list for instance, consumes about 34 GWh in a year. To put this number in scale, a single household in Germany consumed about 3,113 kWh in 2018 according to [Bun20].

1.1 Motivation

Currently, the research for energy optimization techniques is in progress and there are existing promising techniques. A lot of them are using Dynamic Voltage and Frequency Scaling (DVFS) which adapts a Central Processing Unit (CPU)'s frequency to reduce or increase its performance and energy consumption. Some of these approaches are using machine learning to detect application phases in which DVFS is suitable. They mainly differentiate between memory and compute bound phases. In memory bound phases the application workload does not depend on the CPU and therefore its frequency can be reduced to save energy without significant impact on overall performance. During compute bound phases the processor's frequency should be high to have the necessary performance to conduct the workload. However, these solutions have the drawback of only two options and in nearly all cases the system administrator has to know which value to put the CPU's frequency to. There exists a lack of regression based models which predict the energy-optimal power cap.

1.2 Goals and Scope

The scope of this work is to create two regression based models that predict the energy-optimal processor frequency for a given workload. These models shall conduct this prediction during an application's run time. Both models are compared with each other. Additionally, the system's performance is evaluated if fixed system power limits are set.

The models' shall predict power caps for High Performance Computing (HPC) systems which tackle high amounts of workload. Therefore, the aim is to save energy while not reducing the compute power of a system. The power caps shall be place on the optimal power value which is based on instructions per Watt.

1.3 Structure of Work

In order to create these regression models the current state of literature is explained in the next chapter. Additionally, other work closely related to this work is presented. Chapter 3 explains the machine learning methods used. Furthermore, it introduces the tools and frameworks that are necessary for the implementation. The process of data collection and forming the data foundation is explained in Chapter 4. Afterwards in Chapter 5 first implementations and results of the machine learning models are presented and evaluated. Based on these findings the models are optimized in Chapter 6 and used in Chapter 7 to conduct a real system experiment. Different workloads are run for which the systems' power caps are predicted and adjusted on each node. Finally, a conclusion is drawn, findings discussed, and insights about future work is given.

2 Related Work

This chapter introduces other work and material concerning power metering and limiting. Similar work will be presented and differentiated to this thesis.

2.1 Power Metering

In current literature three popular approaches are known to measure energy consumption of applications. Firstly, physical measurement using **external power meters**. To do so additional hardware devices are put between the system and the external power supply. This method is the most accurate and is considered as ground truth in other papers. Secondly, processor vendors install **on-chip power sensors** on their CPUs which can be used to meter the power consumption. These sensors write their data in special hardware counters which can be read by software. Finally, **energy predictive models** simulate the energy consumption of an application. These models are based on theoretical energy models and require application specific parameters to predict the consumption of an entire application or only a time interval. [FSML19] [SFML21] [RRS+14]

As different processing unit vendors offer different on-chip power sensors, the approach to access them differs as well. The two main vendors, namely *Intel* and *AMD*, both offer different interfaces to access their power hardware counters.

Running Average Power Limit (RAPL) is the name of Intel's interface which provides mechanisms for various power related tasks. The interface allows system administrators to meter the current power of the CPU or to enforce power limits. It uses non-architectural model-specific registers (MSRs) which are registers used for debugging, performance monitoring or execution tracing. [Int21b]

Host System Management Port (HSMP) is the interface provided by AMD. HSMP offers access to system management functions like reading processor power or writing processor power limits. To do so, values can be set or read in so-called mailbox registers on the CPU by software running on the system. [Adv21]

Taking a look at current literature, it can be seen that some concerns are raised regarding the accuracy of on-chip power sensors. The authors of [RRS+14] claim that RAPL is close enough to physical measurements to use its power values for power metering. Other work, e.g. [FSML19], shows that for some applications it cannot capture the holistic picture of dynamic power consumption. The authors do not recommend the use of these sensors. However, to be more precise, they divided the applications in three classes regarding the accuracy of the on-chip power sensors to meter their power consumption. One class contains applications RAPL was able to measure precisely most of the time. For another class RAPL's values needed some calibration to be precise. The last class of

applications consisted of ones whose consumption pattern could not be followed by RAPL. The authors explain the existence of the last class by the increasing complexity in modern multicore CPUs concerning shared resources like cache. However, they claim the power values by RAPL to be deterministic and reproducible. They are therefore applicable to be used as parameters in energy predictive models.

In this thesis the use of on-chip power sensors is chosen as power metering option. Although they seem to have flaws regarding their accuracy, their results are deterministic and suitable as model parameters. Furthermore, one of their biggest advantages is that they are state-of-the-art and present in nearly all modern processors. On-chip power sensors do not have to be installed like external power meters would have to be for the test environments or in production later on.

2.2 Power Capping

Power capping describes the process of configuring the CPU to never exceed a defined power limit. As described in the previous Section 2.1, hardware vendors offer interfaces to set power limits by software. Modern microprocessors have different power management states that can be set or adjusted by these interfaces. These are throttle states (T-states), idle states (C-states), sleep states (S-states) and performance states (P-states). Latter are used by RAPL and HSMP as they define the amount of voltage the CPU is allowed to use during workload. [Int21b]

However, limiting the electrical power consumption of processing units comes with the price of reduced compute power. Application executions need more time to finish and the overall computation throughput is reduced. Therefore, reducing the processor power consumption may not always result in energy savings. Used energy for an application to execute can be described by a convex function with a global minimum.

$$P(s) = P_{base}(t) + P_{work}(t), \qquad t = \frac{workload}{s}$$

Where s is a processor's speed and t is the time the processor needs to execute the workload. Execution time t thereby equals the amount of workload divided by the processor speed. The minimum of $P(s)$ is the *critical speed* or *frequency sweet spot* in which the processor consumes the least amount of energy during execution.

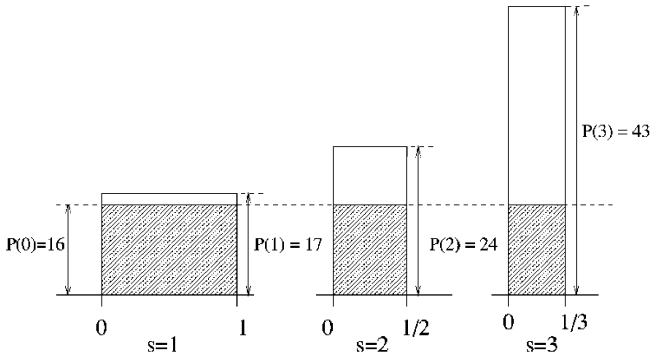


Figure 2.1: Job energy consumption at different processor speeds [ISG07]

Figure 2.1 illustrates the energy consumption function at different processor speeds. The total areas of these bars are the amount of consumed energy. Base energy is marked with gray and is constantly consumed at the same amount $P(0)$. Differences can be seen in the bars' areas as an increased speed means a shorter execution time as well as a higher peak value of power consumption. Due to the convex nature of $P(s)$ the bar of $P(2)$ shows the smallest area and can be seen as the critical speed here. [ISG07]

Another important aspect of power capping are distributed parallel applications. They run on several nodes simultaneously and slowing down single nodes has to be done carefully. The slowdown may propagate to other nodes due to communication and synchronization used in distributed programs. Thereby, the overall performance of the system decreases, and unnecessary waiting nodes are wasting energy. [TLP+13]

Dynamic Voltage and Frequency Scaling (DVFS)

DVFS is special form of power capping. It is an effective method already implemented in modern microprocessors. Here, the power caps are set during run time by DVFS controllers. Several scientific literature about this topic does exist and resulting performance loss is handled differently depending on the requirements of the compute system. [ISG07] [KCCC08] [GBI21]

In [WCC14] the authors present a DVFS system designed for cloud datacenters. Here, the main goals besides energy consumption reduction are better resource utilization as well as the compliance with service level agreements. Cloud datacenters can contain large numbers of heterogeneous servers with different hardware. Therefore, each node has to be evaluated differently.

Reliability in task execution plays an important role in [CKMC21]. According to the authors, DVFS reduces a system's reliability as transient fault rates are higher at low frequency levels. An approach is introduced to combine DVFS with techniques to ensure correct execution of tasks within their deadlines. Task-level, processor-level, and system-level DVFS are considered.

DVFS is also used for chip temperature management. Most work regarding this topic takes action in case the temperature exceeds a certain threshold. The DVFS controller then reduces the frequency to cool down the processor. The approach of [PM21] aims to control processor activity and cooling mechanisms with little impact on performance. Cooling phases during workload shall only be executed if compulsory during runtime. Their approach aims to trigger cooling mechanisms prior to workload execution to keep the impact on performance small.

As the work of this thesis shows, DVFS is also a field related to machine learning. Another work which aims to combine these topics is [GBI21]. The authors introduce a regression model based on workload characteristics to obtain the frequency sweet spot. They furthermore aim to maintain a global power budget and enforce power constraints. Linear regression is chosen by the authors as it is computational inexpensive and therefore results in little overhead during run time. Performance monitoring counters gather instructions per cycle, power consumption, and operational frequency. The models use these values as input parameters to define a voltage-frequency level. In contrast to their paper, this work aims to gain further insights by collecting more performance monitoring counters. Furthermore, different machine learning models are compared and evaluated.

2.3 Further Material

This section presents frameworks, products and tools closely related to this thesis. Some of them have goals that are close to the goal of this work. However, the approaches are always different. None of the works shown here predicts the power cap sweet spot based on regression. Some frameworks shown in this section may offer potential to improve the findings of this work. The references are introduced in order of their publication dates.

The authors of [LTF+14] present a tool to detect application phases based on performance counters – called **Runtime Energy Saving Technology (REST)**. As described in 2.2 application phases can be memory or compute bound and their work limits power differently according to their phase detection. They use second level cache misses, number of CPU cycles and number of cycles spent waiting for the superqueue to empty itself. A superqueue is a buffer for requests to the main memory. These requests are conducted due to cache level two misses. Superqueues are present in each processor core, according to [Int12].
REST uses a linear function to select one out of three processor frequencies. The lowest frequency is for memory bound phases whereas the highest value is defined for compute bound phases. In case the phase is neither memory nor compute bound the frequency to select depends on the overall goal of the system. Hence, the method takes the lowest value if energy consumption shall be as low as possible, or a different value in case REST shall work on low energy with high performance. A great disadvantage of REST is that the exact values for these three frequencies have to be predefined manually. It is therefore mainly a tool to conduct phase detection for an application.

The work presented in [TLP+13] is a tool to provide the sequence of frequencies of the lowest energy consumption for a given application. It is called **UtoPeak** and profiles program executions on all available CPU frequencies by gathering the number of executed instructions and measuring consumed energy. Each phase of an application has to be profiled on every frequency level to gain full insights. As lower CPU frequency lengthens the execution time, UtoPeak splits the application in phases based on instructions and not based on time. This makes sure each phase contains the exact amount of workload. These phase profiles are the foundation to create a sequence of frequencies with the lowest energy consumption. Afterwards, the sequence is then validated with an actual run in which the frequencies are set accordingly. UtoPeak's precision in that work reaches an average of 96% and achieves good results compared to other DVFS based tools. However, UtoPeak was only used on single node parallel applications in [TLP+13]. Furthermore, it is important to mention that UtoPeak is only a tool to evaluate the potential of DVFS for an application. It is not able to adjust the power consumption during run time, as it has to run the application several times in advance on various frequencies.

FoREST-mn is presented in [HPGJ14] which is able to optimize the energy of parallel jobs running on more than one node. This internode DVFS controller takes advantage of the low processor usage of some program phases as well as communication to save more energy. The resulting slowdown on programs is controlled and can have a user-defined threshold. Frequency decisions are taken at run time and program phases are discovered without prior knowledge. However, FoREST-mn can only save energy of iterative programs. These workloads are common in HPC and execute tasks repetitively. This fact allows the controller to run different frequencies for each iteration and determine which one is the power optimal one. In case the program behavior changes and no further

iterations are executed, FoREST-mn has to determine the optimal frequency again. Each process is individually controlled to select the suitable frequency for each task and node. Therefore, processes that create slack time due to message emitting can be sped up to reduce the overall execution time.

Another important work to mention is **Global Extensible Open Power Manager (GEOPM)** which is an open source framework to explore and optimize power usage of heterogeneous compute platforms. It works as a tree-hierarchical runtime job-level power manager. The framework grants further insights in a job's application executions regrading energy usage and can optimize distributed applications by improving their energy efficiency. To do so it detects MPI and OpenMP phases and aims to reduce effects of work imbalance, jitter, and hardware variation. Furthermore, with GEOPM system administrators are able to interact with platform-agnostic interfaces to adjust hardware settings manually. The key difference between other runtime power managers and GEOPM is that the latter is designed for system scales ranging from rack-level to extreme-scale. It's tree-hierarchical structure allows fan-out communication while keeping the additional workload small. A leaf controller is launched on each compute node during a job's execution time. Some of these controllers execute responsibilities of higher tree levels – the aggregator controllers. And finally, one of the aggregator controllers is assigned to be the root controller of the tree. The overall thread structure of GEOPM can be seen in figure 2.2. Lastly, this power manager offers the system job scheduler mechanisms to optimize energy efficiency while not exceeding a scheduler-specified job power budget. [GEO21] [ESC+17]

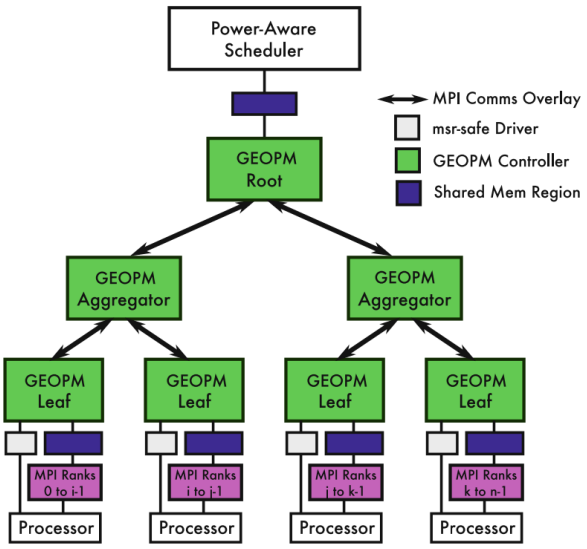


Figure 2.2: GEOPM Tree Structure [ESC+17]

As it comes clear GEOPM mainly focuses on job power optimization. System-wide power is not taken care of. Furthermore, this power manager's abilities are limited to MPI and OpenMP programs, which are the main used runtime frameworks in scientific computing.

The **Bull Dynamic Power Optimizer (BDPO)** is a very similar work to this paper and presented in [Fra17] and [FM17]. The authors of the latter reach application optimization without source code adjustments or pre-execution application profiling. No major performance minimization is witnessed while saving energy by DVFS. BDPO tries to recognize compute behavior and detect workload phases based on performance counters. The performance counters used in this work can be found in appendix A. The authors' selection is taken as baseline selection for this work and is extended further to gain greater insights. The phases BDPO detects are *CPU-bound*, *memory-bound* and *balanced behavior*. As mentioned in other work, the processor frequency can be decreased in memory-bound phases to a certain degree without performance loss. In contrast, during computing phases an increase of frequency directly increases the performance of a compute-system. At the time of writing this work, BDPO is not able to define the optimal frequency on its own. Values for these three phases have to be predefined manually.

As mentioned previously, the goals of BDPO are very similar to the goals of this paper. Both optimize applications without changes in source code and pre-execution analysis. However, the work of this paper aims to be independent of application phases. The levels of optimal power shall not only be defined by compute- or memory-bound phases. Each time interval of an application shall be executed on the most optimal frequency level with a greater number of options.

Another source closely related to this work is [WKE+19]. The authors are using performance counters collected by Like I Know What I'm Doing (LIKWID) to perform machine learning based predictions — similar to this work. However, the work's models predict how much the performance of parallel applications is decreased by power caps. The authors evaluate different machine learning algorithms ranging from clustering methods, over different types of tree based predictions, to neural networks. As their work concentrates on parallel applications, they do not evaluate the performance decrease of the whole application and concentrate only on parallel execution phases. Serial phases are not of their interest.

Finally, when talking about power management frameworks **Variorum** has to be mentioned. It is a vendor-neutral library which exposes interfaces for power monitoring and controlling. These interfaces can be used by other software. Current state of Variorum is not yet production-ready and the number of supported systems is limited. However, it offers great potential, as the calls for power capping are independent of the hardware. They would call the corresponding method of the Variorum interface which increases the flexibility of power management tools. [BBE+21]

In summary, the goals of this work differ to the current state of literature in the following points. This work aims to conduct a regression based prediction of power cap sweet spots, instead of detecting phases with pre-defined power caps. Power cap predictions shall be possible also for unknown workload of any type. Therefore, machine learning models are trained to learn the relationship between CPU behavior and sweet spot. Different types of regression based models are chosen for this task.

3 Prerequisites

In this chapter machine learning concepts and models are clarified which are used in this work. Random forests and neural networks are explained and methods to adjust these models for regression problems are shown. Furthermore, tools and frameworks used in this work are introduced.

3.1 Regression

Regression is a machine learning method which learns relationships between inputs and a continuous numerical output. Similar to other methods the model tries to represent a function f by learning from training data and can thereby predict outputs for novel inputs. Function f can be represented for regression models by $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. The function takes an n-dimensional input and creates an m-dimensional output of continuous values.

Regression is considered supervised learning as the data has to provide target variables to the model. A commonly used category of regression is *parametric regression*. It assumes the function f is well represented by a parameterized model. The model's parameters have to be learned to fit the training data best. [GBC16] [SS15]
Common examples for regression models are Locally Weighted Regression [AS95], Gaussian Mixture Regression [HGCB08], and Support Vector Regression [Vap99].

The regression methods used in this thesis are using a random forest and an artificial neural network. They are explained further in this section.

3.2 Random Forest

The initial idea of a random forest is a classifier consisting of a collection of classifiers Θ_k. It is called forest as the classifiers in the collection are decision trees. They are in depth presented in [Bre01]. A random forest's output for input x is the most popular output of all decision tree classifiers $h(x, \Theta_k), k = 1, ...$ in the forest. The method is called random as the features for individual classifiers are chosen to a certain degree randomly.

Decision trees consist of root and child nodes. These nodes are assigned with a data set's features. When predicting a single data point, one takes a look at the point's values for these features. Depending on these values, one walks down a path in the tree until reaching a leaf node. The leaf contains the class label the data point is given then. An example decision tree and a prediction for a data point can be seen in figure 3.1.

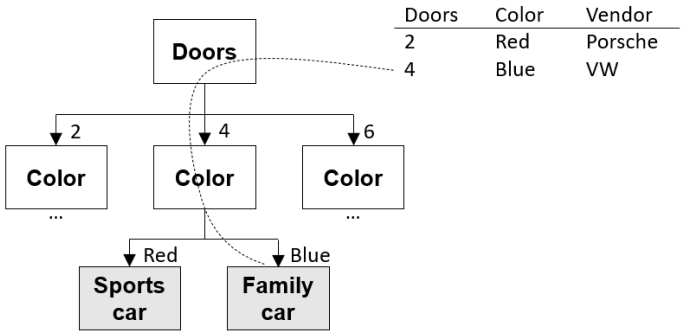


Figure 3.1: Example Decision Tree

The authors of [Bre01] claim that the random forest method has various benefits. They are relatively robust to outliers due to the majority vote. Furthermore, the algorithm is simple, can easily be parallelized, and it can give further insights into the data. Internal estimates of error, decision strength, correlation and variable importance can be analyzed. To do so, out-of-bag estimation can be used which extracts some data sets from the training data. These data sets are not used during the training process and predicted in a later step. The outcomes can be evaluated to retrieve the previously mentioned metrics. More details can be found in [MS10].
Finally, random forests do not overfit when adding more trees. However, doing so creates an upper bound for generalization error. This means that the prediction error on unknown data is limited. This bound is influenced by the accuracy of individual classifiers and internal dependence between the classifiers.

Random forests can be adjusted to conduct regression by using trees that output numerical values instead of class labels. The predictor takes the average over the output values of individual classifiers. Random feature selection increases the accuracy for regression.

3.3 Artificial Neural Networks

Artificial neural networks are machine learning models inspired by biological brains. However, they are not designed as realistic models of brain functions. Fully trained, they represent mathematical functions that are too complex to be created in advance, e.g. due to a high dimension input. These networks learn an approximation of this function from data provided during their training process. This learned function can later be executed on new unknown data. [GBC16]

Artificial neural networks contain several layers of neurons that are connected with each other. These neurons can be activated individually depending on their inputs and forward their activation to the next layer of neurons. A detailed illustration of such a neuron can be seen in Figure 3.2. The output values of following neurons $u_1 ... u_D$ are multiplied by so-called weights $w_{j1} ... w_{jD}$ and afterwards a bias $b_{j1} ... b_{jD}$ may be added. These results are summed up to z_j and given to the activation function ψ_j. This function's outcome is also the neuron's outcome o_j which is provided to the following layer of neurons. [Sch20] [SS15]

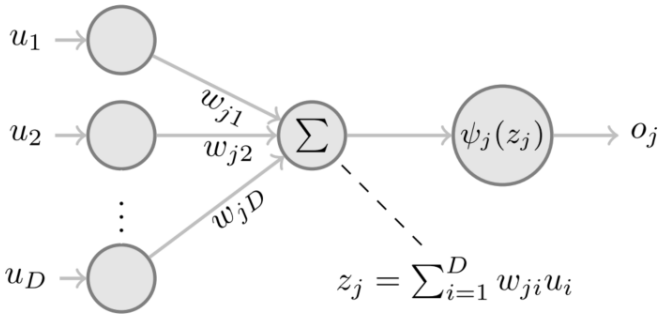


Figure 3.2: Detailed View of Neuron [SS15]

An example of an activation function is the **rectifier linear unit** or **ReLU**:

$$\psi(x) = max(x, 0)$$

It is the recommended function for neural networks. As illustrated in Figure 3.3 the function consists of two linear pieces. According to the authors of [GBC16] the model therefore preserves benefits of linear models. These are for instance, good generalization and easy optimization. [GBB11]

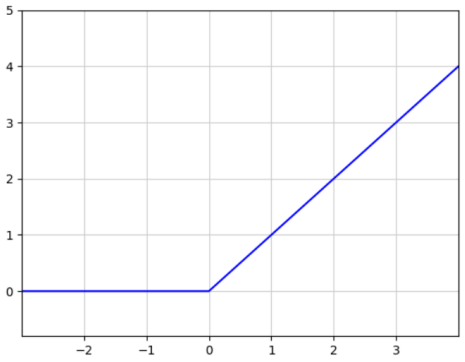


Figure 3.3: ReLU Function

Training

During the training process of a neural network training data is processed by the network. The output is evaluated by a cost function which provides insights of how big the difference between the network's output and the true label is. To minimize this error, weights of the neurons have to be adapted by using the gradient descent method. One can say the network learns from the data. The aim of training is to minimize the values of the cost function for all data points in the training data set. However, as the network's goal is to process unknown data sets, its performance is also evaluated using a separate data set – the test set.

When trying to keep the error for the training data set small, it might be possible that the model is no longer able to generalize. Meaning it cannot process new data points reliably which results in **overfitting**. The opposite is to train the model not sufficiently. In this case even the model's outcome for the training data has a high error and the model is **underfitted**. One has to find a good balance between a model's performance on the training data and the test data. [GBC16]

Architecture

As introduced earlier neurons are usually arranged in layers. The architecture of a complete artificial neural network starts with the *input layer*, as shown in Figure 3.4. On the other end there is an *output layer* which provides the results of the neural network. Although in the example figure only

one output neuron is seen, it can consist of more than just one neuron. Between these two layers, there can be any arbitrary number of layers. They are called *hidden layers* as no direct outside communication with their neurons is intended. [Nie15]

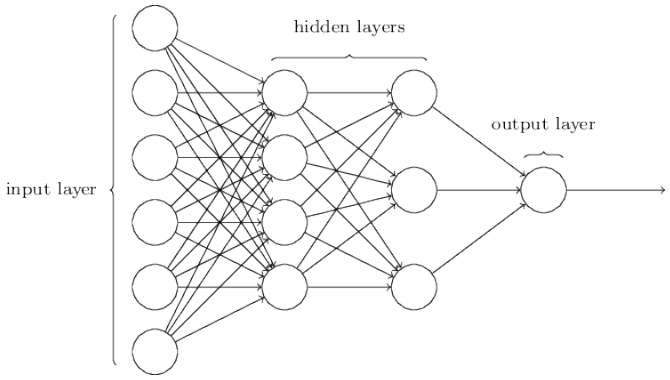


Figure 3.4: Architecture of Artificial Neural Network [Nie15]

All examples so far and all upcoming neural networks in this work send the output of neurons to the neurons of the next layer. They forward their outcomes through the network, making it a so-called *feedforward neural network*. However, it is possible to redirect outcomes of neurons to previous layers to create recurrent neural networks. These are not used in this work as it increases the complexity significantly.

3.3.1 Neural Network Regression

In [SS15] artificial neural networks are mentioned as a possible algorithm for regression. However, several versions of them are explained. Some of them are listed in the following subsections.

Extreme Learning Machine (ELM)

The simplest version is the *Extreme Learning Machine (ELM)*. It is a method to train single-hidden layer feedforward network. Meaning, it does only have one hidden layer besides the input and output layers. Furthermore, it is important to mention that Extreme Learning Machines (ELMs) are a batch regression algorithm. Meaning, training examples are provided to the model all at once and the model parameters are fixed afterwards. Doing so reduces the training time significantly. The authors of [HZS06] state that an ELM has the potential to train a neural network thousands of times faster than backpropagation. Its mathematical model looks like this:

$$\sum_{i=1}^{N} \beta_i g(w_i * x + b_i) = o$$

Here, N is the number of hidden nodes and $g(x)$ the activation function. w_i is the weight vector connecting input nodes and the hidden layer. And all b_i are the biases in this network. Whereas, β_i is the weight vector connecting the hidden layer and the output layer. Finally, x are the input samples and o the output values.

The ELM algorithm only consists of three steps. First of all, random values are assigned to the weights in w and biases in b. Afterwards the hidden layer output matrix H is calculated using the training samples. And lastly the output weights in β are computed to fit the training set. As this method is not using gradients during training, it is resilient to local minima in error functions and can be used for non-differentiable activation functions. [HZS06]

Backpropagation

The opposite of an ELM is the backpropagation method. Here, training is done incrementally by updating the weights of the neural network using one training example at a time. Data sets create an output and a value for the cost function. In backpropagation the resulting error flows backwards through the network. This information can be used at each layer to compute the gradients which are used to adjust the network's weights and to minimize the cost function for this data point. A gradient ∇f is the partial derivative of a function f and shows how much and in which direction parameters have to be adjusted to reach extreme points of function f. Weights can be adjusted using gradient descent and a learning rate ϵ:

$$w_{new} = w_{old} - \epsilon * \nabla f$$

This computation is done at each layer and the weights can be computed in parallel as vectors. The chain rule is used to propagate the error backwards through the complete network. Thereby, the gradient of the cost function at each layer can be computed. [GBC16] [RHW86]

3.3.2 Deep Neural Network (DNN)

In contrast to ELMs, backpropagation can be used in multi-layer feedforward neural networks. The authors of [SS15] state that using ELM only the final hidden layer's weights can be trained. This is due to the fact that these are the only weights linear with respect to the network's output. A multi-layer feedforward network is also called Deep Neural Network (DNN). Each layer can learn its own representation of the data and may discover structures in large data sets. This deep architecture of the network allows it to learn and represent more complex mathematical functions than single layered networks could. [SS15] [Ben09] [LBH15]

Constructing a Deep Neural Network (DNN) involves many design choices. One has to choose a cost function, an optimizer, the form of output, and the design for the hidden layers. The form of output in this case is a numerical value as the DNN is used for regression in this work. For the hidden layers several further options exist: the amount of layers, how the layers are connected, their activation function, and each layer's width. This part is an active research area and there is no guiding theory yet. The authors of [GBC16] state that the design process consists of trial and error. Different designs can be tested against a validation data set and the most promising can be used. [GBC16]

3.4 Tools and Frameworks

3.4.1 Like I Know What I'm Doing (LIKWID)

LIKWID offers a variety of tools for performance oriented programmers. It is developed at the computing center in Erlangen and funded by the Federal Ministry of Education and Research of Germany. LIKWID provides engineers insights and interfaces to CPUs produced by AMD and Intel. For instance, using `likwid-topology` one can probe hardware thread and cache topology in multi-socket nodes or `likwid-pin` enforces threads to execute on defined cores. [THW10]

likwid-perfctr

However, LIKWID's only tool used in this work is `likwid-perfctr`. It grants insights into a CPU's performance counters. Performance counters count hardware events which are taking place on a processor during code execution. This functionality is implemented directly in the hardware and therefore results in no overhead. These counters allow engineers to know what is exactly happening on a processor at the moment or during their program's execution. `likwid-perfctr` contains various modes to allow different types of measurements. For example, the *wrapper mode* measures only hardware events that are related to an application's execution. The mode used in this work is the *stethoscope mode*. It allows to profile the complete performance counter for a defined time interval. [THW10] [RTHW14]

The number and types of performance counters as well as for the hardware events differs from CPU to CPU as they are closely related to hardware. A list of the available hardware events for the AMD Zen2 architecture which is used in this work can be found in the appendix A. An example for how `likwid-perfctr` is used can be seen in section 4.5 in listing 4.1.

3.4.2 scikit-learn

scikit-learn is a widely used Python module for machine learning. As an open source project it aims to provide various machine learning algorithms on an easy-to-use basis. These algorithms can be supervised or unsupervised and face tasks of classification, regression, and clustering.
Furthermore, scikit-learn offers a variety of tools that might be used in model evaluation and data processing — for example feature extraction or cross validation. [PVG+11]

3.4.3 PyTorch

PyTorch is an open source library for tensor computation and deep learning. A tensor is a multidimensional array which is utilized in neural network computations. They can be transferred to the graphical processing unit by PyTorch to utilize its highly parallel and compute intensive architecture. PyTorch offers a simple interface to work with neural networks in Python. A neural network can be created in a few lines of code. [IPK21]

Different types of network layers and activation functions can be added to the network. All types of layers and activation functions offered by PyTorch can be found in [Tor19a]. Furthermore, different optimization functions for the neural network are provided by PyTorch which are used during the training process. These can be found in [Tor19b].

3.4.4 Tune

Tune is an industry standard tool for distributed hyperparameter tuning maintained by the Ray team. Hyperparameters are model variables that are defined before training a machine learning model. They mainly consist of model architecture decisions. For instance, the number of tree classifiers in a random forest is a hyperparameter.
Tune is simple to be integrated with PyTorch to be used on neural network models. It offers a simple interface between training functions and hyperparameter search algorithms. The concept of Tune can be seen in Figure 3.5. One defines a training function for the neural network which will be wrapped by Tune's trainable interface (Trainable API). This function will be executed by `tune.run()` with various options for the models hyperparameters. The model is trained several times or even simultaneously with different hyperparameter configurations. `tune.run()` furthermore allows us to log training results, set training checkpoints, or utilize early stopping. In the last step — `tune.Analysis` — all models are evaluated and the best hyperparameter setting is chosen. [Ray22] [LLN+18]

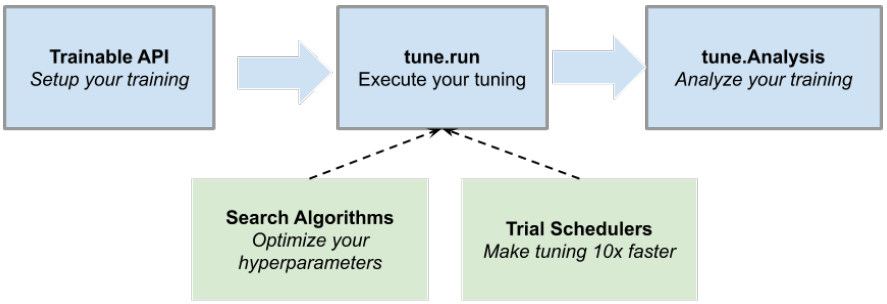


Figure 3.5: Tune's Key Concept [Ray22]

Furthermore, Tune offers different searching algorithms. Ranging from basic ones like grid or random search to more advanced algorithms. It converts the provided search space of hyperparameters values to a format the search algorithm expects.
To increase the training's process efficiency, Tune provides *Trial Schedulers*. They allow stopping or tweak hyperparameter trainings. For example, in case a hyperparameter configuration is performing badly compared to another one, the Trial Scheduler can terminate that training to save compute resources and decrease the overall training time. [Ray22]

4 Data Collection

The goal of this work is to create machine learning models that are able to predict power caps based on workload behavior. As a first step it is important to create a dataset that can be used to train the machine learning models. It has to capture a workload's behavior and contain a power cap target variable.
This chapter presents how and what data is collected to characterize a workload. Additionally, it shows how the energy-optimal power caps are defined. The data collection process is explained and which type of data is collected. Furthermore, the benchmarks to collect the data from are introduced.

4.1 Design

To characterize a workload, performance monitoring counters are collected as they keep track of which tasks are conducted by the CPU. These counters are collected for a time interval t_1 which is called **characterization phase**. As the aim is to predict the optimal power cap for the follow-up execution interval t_2 instructions and power in Watt are measured during that phase. This follow-up interval of length t_2 is referred to as **performance interval**. These measurements have to be gathered for all power caps and can be used to calculate the instructions per Watt ratio. The power cap providing the best ratio will be the resulting label.

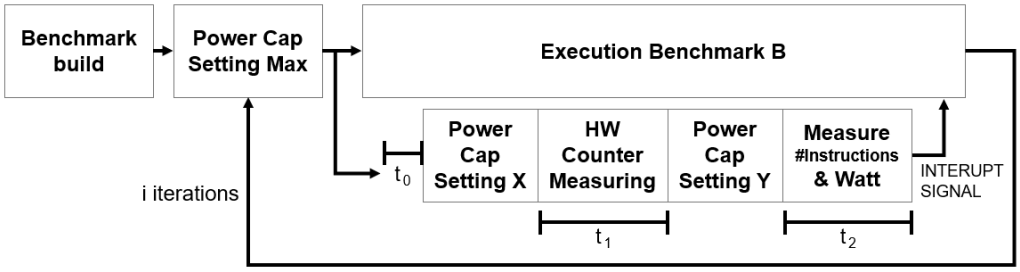


Figure 4.1: Data Collection Process

Figure 4.1 shows the data collection process of a benchmark. Firstly, the benchmark has to be build and the execution environment is prepared. The power caps are set to the maximum value. This ensures that the benchmark is at the same point of execution after time interval t_0 as it has to be run several times. Afterwards, the benchmark can start its execution. At the same time another process is started which waits for the time interval t_0 to pass. This process then caps the power of the system to the value X of that iteration and measures the performance counters for time interval t_1. Then, the process caps the system again to a different power cap Y and measures instructions and Watt for

time interval t_2. When this is done an interrupt signal is sent to the benchmark execution as the rest of the execution is no longer needed. This process is executed for each benchmark B, power level X and power level Y.

4.2 Benchmarks

In this work different types of workloads are chosen for the data collection. They are explained in this section. The Exascale Computing Project (ECP) Proxy Applications and NAS Parallel Benchmarks (NPB) suite are benchmark collections that are available on their websites. An OpenFOAM application, a molecular dynamics simulation, and a NS3Dneo application are provided by the Höchstleistungsrechenzentrum Stuttgart (HLRS). All of them are real world applications.

4.2.1 Exascale Computing Project (ECP) Proxy Applications

This benchmark suite offers several proxy applications which are small, simplified codes. They model computational characteristics of large applications without the complexity of large code bases. The ECP proxy apps specifically aim to represent the most important features of exascale applications. Ember and SWFFT proxy apps are used in this paper. [Exa22]

Ember represents multi-node communication patterns that are relevant to HPC workloads. These patterns have a great impact on scalability and parallel performance. In this paper the halo3d and the halo3d-26 pattern are chosen. The benchmarks are therefore named *ember* and *ember26*. Both are nearest neighbor-like. However, halo3d is defined as structured, in contrast to halo3d-26 which is unstructured nearest neighbor-like.

SWFFT is a distributed fast Fourier transformation. The proxy app firstly distributes data between ranks and afterwards computes the Fourier transformation algorithm.

4.2.2 NAS Parallel Benchmarks (NPB)

NAS Parallel Benchmarks (NPB) is a collection of programs to evaluate the performance of parallel supercomputers. It is created by the NASA Advanced Supercomputing (NAS) Division. The collection contains kernels and pseudo-applications which can all be adjusted in workload size. [NAS22]

The **embarrassingly parallel (EP)** kernel measures performance without interprocessor communication. Limits of floating point performance are estimated.

Short and long distance communication is tested by **multigrid (MG)**. The kernel utilizes highly structured data communication.

An approximation of the smallest eigenvalue of a large, sparse, symmetric positive definite matrix is computed by the **conjugate gradient (CG)** kernel. Long distance communication of an unstructured grid is tested. As well as matrix-vector multiplication.

Fast Fourier transformation (FT) is a kernel that tests long distance communication performance.

Integer computation is tested in a large **integer sort (IS)** kernel.

During the execution of the **lower-upper Gauss-Seidel solver (LU)** a large number of very small messages are sent. A matrix is decomposed into a lower and an upper triangular matrix.

Scalar Penta-diagonal solver (SP) as well as **block tridiagonal solver (BT)** are pseudocodes which solve three sets of uncoupled systems of equations. The difference between both codes are the structure of these systems of equations. Coarse grained communication is tested.

4.2.3 OpenFOAM

OpenFOAM is an open source object-oriented library for Computational Fluid Dynamics (CFD) written in C++. Its goal is to create physical models to simulate motion and forces of fluids – namely liquids and gases. However, it might also include simulation of thermodynamic models. CFD applications are created to simulate their behavior. An example application would be the air flow around a car as shown in Figure 4.2. Instead of creating a new physical model for every change in a vehicles design and test them in wind tunnels, one can adjust the virtual model and run the simulation. This method is less expensive and offers more flexibility for designers. [Jas09]

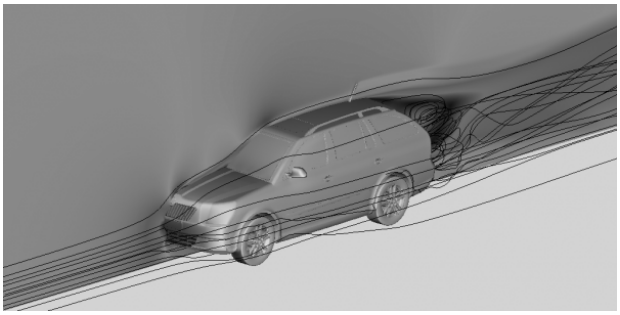


Figure 4.2: Car Air Flow CFD Example [Gre16]

The workload of an OpenFOAM application is separated into three stages – *Pre-processing*, *Solving*, and *Post-processing*. Pre- and Post-processing take care of data handling and make sure that the actual solver can execute in different environments. Running the solver in parallel, the associated fields of fluids are divided and allocated to different processors. This method is called domain decomposition and uses the Message Passing Interface (MPI) for communication. In the parallel execution cases pre-processing is responsible for decomposing and distributing the workload. Whereas during post-processing the solvers' solutions are reconstructed into one field. As the solvers' workload contains numerical analysis a lot of matrix multiplications are executed iteratively. To finish the iterative work of a solver either a defined number of iterations is exceeded or the current solution's value is below a tolerance threshold. [Gre21]

4.2.4 Molecular Dynamics Simulation

This type of application tries to capture the behavior of biomolecules in atomic detail and in fine temporal resolution. Molecular dynamics simulations predict the movement of every atom in a molecular system over time. Several steps are calculated repeatedly to predict the current state of motions of these atoms. The atoms' positions, forces, and velocities are updated constantly based on physical computations. Furthermore, bonding and non-bonding molecular interactions are also taken into account. The workload of these systems is an iterative calculation for multiple particles in parallel. [HD18] [All+04]

Molecular dynamics simulations are for example used in drug discovery. Molecules are modified in simulations to test how the molecule binding process behaves. Thereby, scientists can adjust these molecules until they behave as intended.

4.2.5 NS3Dneo

NS3Dneo is another CFD application similar to OpenFOAM. However, this one is a direct numerical simulation for aeroacoustic simulations. Its goal is to simulate turbulences resulting from airplane turbine jet streams. These findings are used to check noises produced by these turbines and reduce them. An example is illustrated in Figure 4.3. A flow field including two streams with different velocities is simulated. These streams result in disturbances creating roll-ups. The Figure shows these roll-ups between X values of 100 and 150. They are the dominant source of noise of a jet turbine and have to be avoided to reduce the level of sound. NS3Dneo uses domain decomposition to assign the workload to different processes which solve their tasks using multithreading. Simulations of flows require costly iterative procedures to find correct pressure distributions. The workflow therefore contains multiple iterations. [WPS+19] [Bab09]

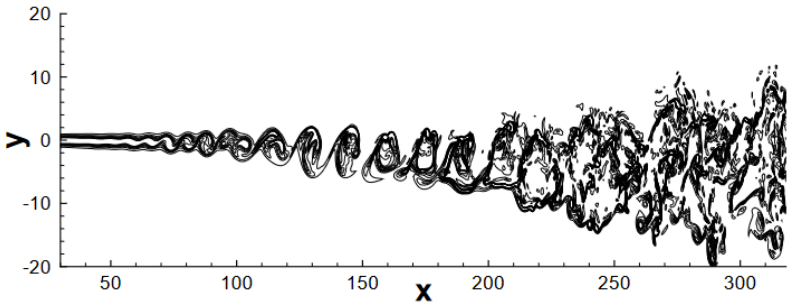


Figure 4.3: Example Problem for NS3Dneo [Bab09]

4.3 Performance Counters

The authors of [FM17] decided to detect application behavior based on three performance groups. These are *Instructions & Cycles*, *Package Energy Consumption* and *Memory Usage* including level one and two cache usage. This mix of event sets which can also be seen in Appendix A looks promising and includes all essential dimensions to characterize an application. This set of events is chosen as a great foundation to be adjusted. It is extended to face this work's problem and to fit in this work's environment.

LIKWID offers several groups of performance counters to reflect an application's behavior in different dimensions. For the Zen2 architecture a list of the corresponding events in each group can be found in Appendix A. To grasp as much information as possible about an application the event mix for the data collection is desired to spread across several of these performance groups. The Zen2 processor offers 2 Fixed Purpose Counters (FIXC), 6 Performance Monitoring Counters (PMC), 4 Data Fabric Counters (DFC) and 2 Cache L3 Performance Monitoring Counters (CPMC) allowing to measure 14 events at a time. However, events are bound to a certain type of performance counter. E.g. the *RETIRED_INSTRUCTIONS* event is only possible to be measured with a PMC.

A solution to this limited counter problem is to measure different event sets after another. This method allows further insights into other dimensions of the application. However, it does not grasp the full potential of an event as there might be a different behavior of an application during the time this event is not measured.

Counter	Event Set 1	Event Set 2
PMC0	CPU_CLOCK_UNHALTED	ICACHE_FETCHES
PMC1	RETIRED_INSTRUCTIONS	ICACHE_L2_REFILLS
PMC2	RETIRED_BRANCH_INSTR	RETIRED_SSE_AVX_FLOPS_ALL
PMC3	RETIRED_MISP_BRANCH_INSTR	MERGE
PMC4	DATA_CACHE_ACCESS	LS_DISPATCH_LOADS
PMC5	DATA_CACHE_REFILLS_ALL	LS_DISPATCH_STORES
DFC0	DRAM_CHANNEL_0	DRAM_CHANNEL_0
DFC1	DRAM_CHANNEL_1	DRAM_CHANNEL_1
CMPC0	L3_ACCESS	L3_ACCESS
CPMC1	L3_MISS	L3_MISS

Table 4.1: Event Sets Used in Data Collection

As a trade off the number of event sets is chosen to be small and thus reducing the time certain events are not measured. The event sets can be seen in Table 4.1. FIXC are not utilized as they are only capable of measuring events concerning the CPU clock. However, it is not recommended measuring these using the FIXC. PMC should be used instead.

Therefore, PMC counters are used to measure the CPU clock (CPU_CLOCKS_UNHALTED) in addition to executed instructions (RETIRED_INSTRUCTIONS). Additionally, they are used to take a look at instructions that are executed before the CPU knows which execution path it will take. These are either predicted correctly (RETIRED_BRANCH_INTR) or it comes clear that these instructions are executed falsely (RETIRED_MISP_BRANCH_INSTR) and have to be reverted according to [Int21a]. Cache events for the PMC counter type are read, write and prefetch access to the data cache (DATA_CACHE_ACCESS), allocations to the data and instruction cache (DATA_CACHE_REFILLS_ALL / ICACHE_L2_REFILLS) and fetching access to the instructions cache (ICACHE_FETCHES). Executed floating point vector instructions are measured using two PMC counters. According to Thomas Gruber, a LIKWID developer, the corresponding events (RETIRED_SSE_AVX_FLOPS_ALL & MERGE) have to be placed on PMC2 and PMC3 for Zen2 architecture. This is due to the fact that AMD transferred these counters from an older processor generation in which increments of performance counters are smaller than 16. However, in this Zen2 it might be necessary to increment by 16 although only a signal path of four bits is available, e.g. when using single precision fused-multiply-add with AVX512. The MERGE event is therefore used as an extension of the RETIRED_SSE_AVX_FLOPS_ALL event to combine the signal paths of both counters and enable greater increments. The last events measured by PMC count the dispatched load and store operations (LS_DISPATCH_LOADS / LS_DISPATCH_STORES).

Finally, the last counters are DFC and CPMC. Two DFC measure events counting read and write commands to the memory (DRAM_CHANNEL_0 / DRAM_CHANNEL_1). The last registers are CPMC which measure accesses and misses to L3 cache. The events of both counter types are gathered in both event sets. [Adv19] [Adv21]

4.4 Environments

The environment the data is gathered from is described in table 4.2. The system contains several nodes of which three or four were used during benchmark execution. All nodes have the same hardware configuration and two sockets with an AMD EPYC 7702 processor installed. These have 64 cores and cache sizes of 512 kilobytes for L2 and 16384 kilobytes for L3 cache. The total memory of a single node is about 52 gigabytes.

CPU	Sockets	Cache (L2 / L3)	Total Memory
AMD EPYC 7702 (64 Cores)	2	512 KB / 16384 KB	52 GB

Table 4.2: Test Environment

4.5 Implementation

Scripts are created to execute the data collection process automatically and several times. The values for the power cap are 100, 120, 140, 160, 180 and 200 Watt for the processors. Therefore, six power caps exist which each have to be executed six times to get the optimal cap for the next execution interval. According to this, each benchmark has to be run 36 times.

The size of t_1 is ten seconds as each performance counter event set is measured for five seconds. The `likwid-perfctr` command is executed with each event set, as for example in Listing 4.1.

```
EVENT_1="CPU_CLOCKS_UNHALTED:PMC0,RETIRED_INSTRUCTIONS:PMC1,"\
        "RETIRED_BRANCH_INSTR:PMC2,RETIRED_MISP_BRANCH_INSTR:PMC3,"\
        "DATA_CACHE_ACCESSES:PMC4,DATA_CACHE_REFILLS_ALL:PMC5,L3_ACCESS:CPMC0,"\
        "L3_MISS:CPMC1,DRAM_CHANNEL_0:DFC0,DRAM_CHANNEL_1:DFC1"
./likwid-perfctr -f -c 0-255 -g "$EVENT_1" -O -S 5s
```

Listing 4.1: likwid-perfctr Example

The `-f` flag tells likwid-perfctr to force the writing of registers. `-c` defines the processor's hardware threads to measure. Each processor core has two of them and the system does have two sockets with 64 cores each. Therefore, 256 hardware threads per node have to be measured. Which performance monitoring counters are measured is defined by the `-g` flag and the `EVENT_1` variable which is defined in the top line. It is a list of events which are followed by a colon and the corresponding counter to measure this event in. `-O` tells `likwid-perfctr` to output the values in Comma-Separated Values (CSV) format. Finally, `-S` tells LIKWID to measure in stethoscope mode a defined time. In the listing 4.1 it is defined as five seconds.

For the next execution interval of t_2 a size of 15 seconds is chosen. During this time only instructions are measured by likwid-perfctr and the used power is measured by HSMP. As the latter is only able to gather the power in points of time, four power values are taken. This happens at the beginning and three times in gaps of five seconds.

Different numbers of nodes are used for the benchmarks. For NPB and ECP four nodes are used. Whereas the OpenFOAM benchmark is executed on three nodes only. The measurements for each node are written into a file named by a node's name followed by `-pm.out`. These files are formatted,

merged, and appended to a file by `format_data.sh`. The result is a single CSV file for several benchmark executions – `formatted_data_n.csv`. This process is illustrated by the first part of Figure 4.4.

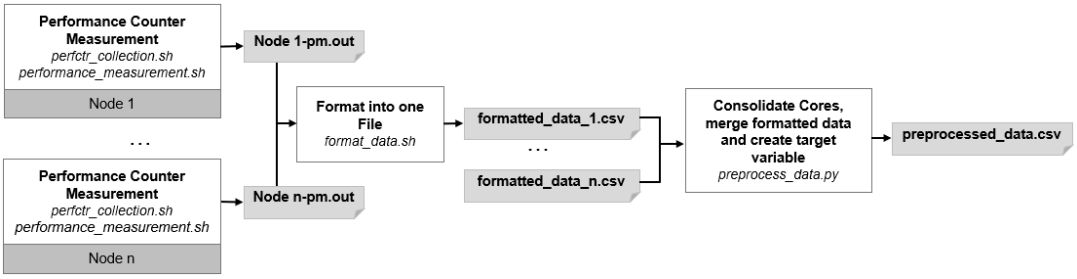


Figure 4.4: Data Preprocessing Implementation

The next step in the data preprocessing is the execution of the `preprocess_data.py`. Data is grouped by benchmark and power cap X. To get the value of energy, the mean of the power values is taken and multiplied by the time interval which is 15 seconds. All cores' instruction value are summed and divided by the number of cores to also get the mean value. The instruction mean is then divided by the average power used to get the instruction per Watt ratio. As this is done for all power caps in an execution interval, one can select the power cap giving the max value for this ratio. This power cap becomes the target value for this data point.
The process is repeated for each group in each `hostname-pm.out` file. Thereby a single file is created containing data ready to be processed by the machine learning models – `preprocessed_data.csv`.

A first dataset is created containing 96,192 data points as shown in Table 4.3. 22,464 of these points contain data for runs of ECP Proxy Apps. Meaning, each application has about 7,500 points. For the NPB benchmark suite 63,360 data points are created which makes approximately 8,000 points per benchmark. Finally, the OpenFOAM application has about 10,000 data points.

ECP Proxy Applications	22,464 (7,488 each)
NAS Parallel Benchmarks	63,360 (7,920 each)
OpenFOAM	10,368

Table 4.3: Number of Data Points Collected

A best practice in machine learning is to separate a dataset into training, test and validation datasets. Training data is used during the learning process and the models are adjusted based on this data. To evaluate the learning process testing data is processed, and their prediction error can be checked. These datasets are created out of points generated by the ECP and NPB benchmarks. The training set contains three fourth of the data and the testing set contains the rest. Before splitting the dataset, it is shuffled. The data points of the OpenFoam benchmark are used as a validation dataset. This dataset is put aside to check later on how the models perform on new data which is unknown to them.

5 Machine Learning Models

The collected data is used to train the machine learning models in this chapter. It is stated how the random forest regression and neural network are created. Furthermore, first results for both models are shown and evaluated.

5.1 Random Forest Regression

5.1.1 Design

Random forest regression is implemented in Python using the `scikit-learn` package which is explained in further detail in Subsection 3.4.2. Parameters for this model are the number of trees, maximum depth of the trees and other tree splitting related options. To find the most suitable ones for the training and testing dataset parameter search is conducted. GridSearch is used which is also part of the `scikit-learn` library and defined in [sci22]. It creates a parameter grid with every combination of values for these parameters that are provided to GridSearch. The parameters are then optimized using **k-fold cross-validation**.

Cross-validation aims to prevent overfitting by resampling data. It splits the learning dataset in training and test dataset. In k-fold cross validation the overall dataset is split into k equal sized disjoint datasets. k - 1 subsets are taken to form the first training dataset which is used to train the model. The model is then evaluated by the remaining subset – the test subset. This training and evaluation process is repeated until each subset has been the test subset. The average of these evaluation performances is then the cross-validation performance. [Ber19]

Number of Trees	200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000
Maximum Depth	10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Split Features	`'auto'`, `'sqrt'`
Cross Validation	3
Scoring	`'neg_mean_absolute_error'`

Table 5.1: Parameter Grid and Parameters for GridSearch

The provided parameter grid and parameters for GridSearch itself are listed in Table 5.1. For the number of trees a value range from 200 to 2000 is chosen. It might be important to mention that using more trees in random forest regression does not lead to overfitting. Meaning the only downside when choosing more trees is a reduced performance due to more computational workload. A list ranging from 10 to 100 is provided to the maximum depth of the trees in the forest. Two options are

provided to the split feature parameter `max_features`. It defines how many features are considered when a tree has to be split. `'auto'` means that all features are considered whereas using `'sqrt'` only as many features are considered as equal to the square root of the total amount of features.

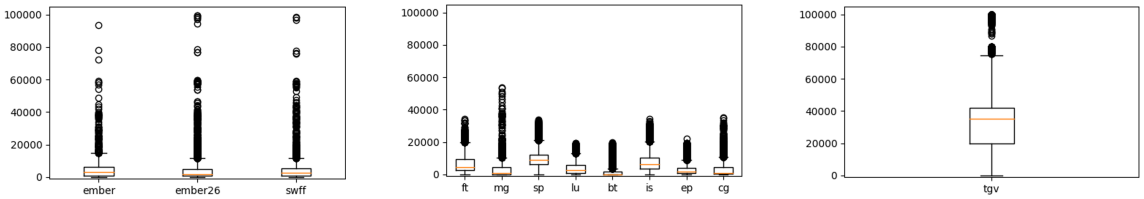
The last two options in the table are parameters provided to GridSearch. Cross validation tells the algorithm what kind of k-fold cross validation should be conducted. Here, a three-fold cross validation is used. Scoring defines how the validation should be conducted. `'neg_mean_absolute_error'` means to minimize the *Mean Absolute Error* [CM04].

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} |y_i - \hat{y}_i|$$

It is the sum of absolute difference between predicted, and actual value divided by the number of data points. This metric is chosen here and in further evaluation of the model as it describes the error distance well. Furthermore, it can be illustrated greatly as it contains the exact value without further processing needed.

5.1.2 Results and Evaluation

The resulting forest consists of **2,000 trees** with a **maximum depth of 20**. First results on the training dataset look promising as random forest regression achieved a mean absolute error of about 4,330 for ECP Proxy Apps and about 4,742 for the NPB suite. As the processor has a power range starting at 100,000 milliwatts up to 200,000 milliwatts an error of 5,000 is only 5 percent of the possible value range. It can be considered a rather good value. However, for the OpenFoam application the mean absolute error is equal to 34,765. Figure 5.1 shows the error values for each benchmark.



(a) Absolute Errors ECP **(b)** Absolute Error NPB **(c)** Absolute Error OpenFoam

Figure 5.1: Results for Random Forest Regression

One can see in Subfigure 5.1a that the results for the ECP Proxy Applications are spread across the whole range. However, the three fourths of all values are below an error value of 20,000 as the box plots show. This is illustrated by the upper bound of the box which also is the third quantile.
Although the NPB benchmark suite performs slightly worse in terms of mean absolute error, the results are less spread as shown in Subfigure 5.1b. No prediction exceeds 60,000 and all boxes are below 20,000.
The results for the OpenFOAM application can be seen in Subfigure 5.1c. It can be seen that the box is placed in a higher value range and that the error values are spread across the whole range. The OpenFOAM application is not part of the training nor the test dataset. It is therefore completely unknown to the random forest regression. These bad results might suggest that the model is overfitting to the training data which is investigated in Section 6.

5.2 Neural Network

5.2.1 Design

For the neural network an architecture with four layers is chosen which is illustrated in Figure 5.2. The input layer consists of 121 neurons which normalizes the inputs. Meaning, the values are adjusted to be in range between zero and one. The next two layers are hidden layers and equal in size. They consist of 256 neurons and are fully connected with their following layer. The output layer does only consist of one neuron as only one numerical output value is expected.

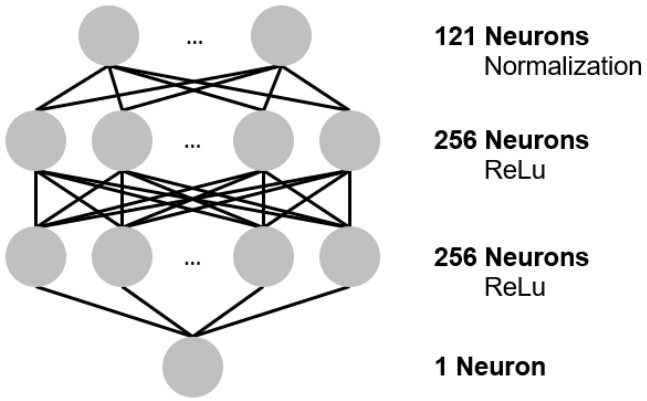


Figure 5.2: Architecture of Created Neural Network

The network is implemented also in Python using PyTorch which is presented in Subsection 3.4.3. It offers various tools to create neural networks in Python. Both hidden layers are using the *ReLU function* as activation function as this one is simple and gives the model the ability to generalize well, as stated in Subsection 3.3. As regression is conducted no activation function is necessary for the output neuron. The difference between actual target values and the values predicted by the network are evaluated by a loss function. The network uses the *Mean Squared Error Loss* [CM04] which is often used for regression. It is the average squared difference between the predicted and actual value for the dataset.

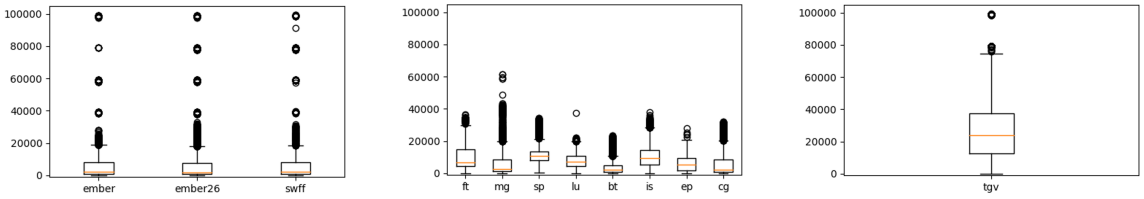
$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} (y_i - \hat{y}_i)^2$$

Adam is chosen as optimizer. It is a gradient-based optimization for neural networks' weights. The authors of [KB14] state that it is computationally efficient and easy to use as its hyperparameters require little tuning.

The network is trained with a batch size of 64 and 200 epochs. The batch size describes how many data points form a subset which are processed for one learning step. Each update to the weights is computed based on the average cost function results of this subset. [GBC16]
An epoch describes one complete usage of the dataset. The number of epochs therefore states how often the complete dataset is used in the training process. Each data point of a set is used during one epoch, either in a batch or processed as a single data point.

5.2.2 Results and Evaluation

Results for this model can be seen in Figure 5.3. Errors values of the model's prediction are illustrated in a boxplot for each benchmark.



(a) Absolute Errors ECP **(b)** Absolute Error NPB **(c)** Absolute Error OpenFoam

Figure 5.3: Results for Neural Network Regression

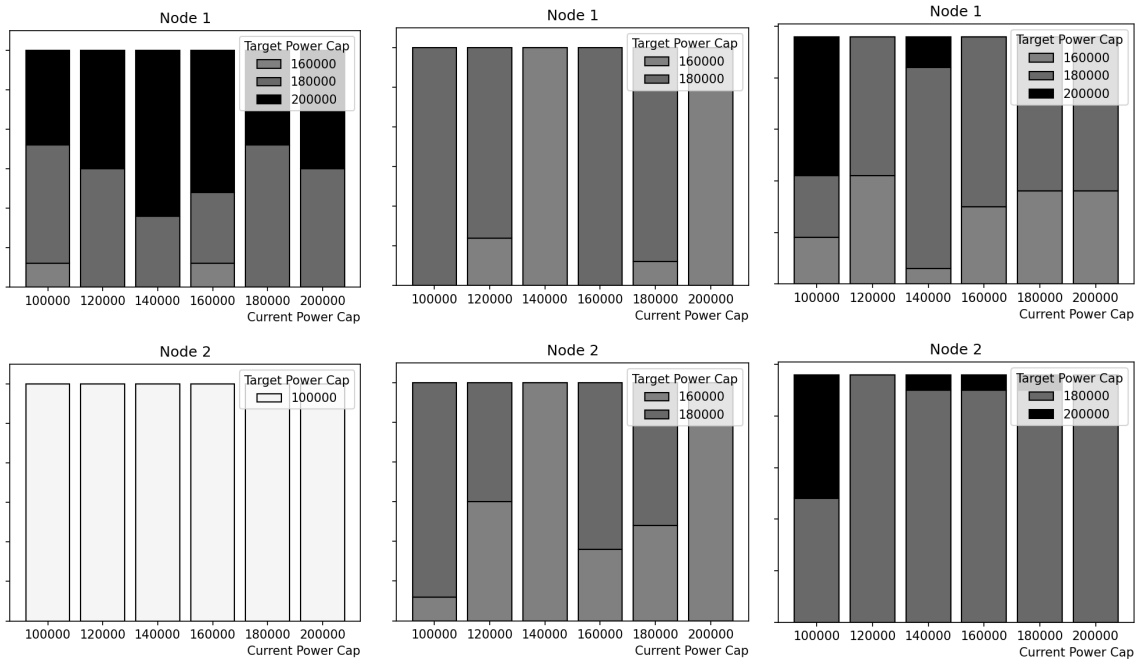
Predicted values are overall quite close to the target variable for ECP and NPB applications. As Subfigures 5.3a and 5.3b show the boxes are all below error values of 20,000. The main difference between both box plots is that for the ECP Proxy Apps the errors are spread across the whole value range. In contrast, the NPB applications never exceed error values of 65,000 milliwatts. Nevertheless, the mean absolute error for ECP Proxy Apps is 6,043 and therefore lower than the mean absolute error for the NPB suite which is 7,696.
For the OpenFOAM application the neural network behaves similar to the random forest regression. This application was also unknown by the network and the mean absolute error is about 29,925 milliwatts. Behavior like this might suggest that the model is overfitting to the training dataset.

6 Optimization

Based on the results of the last chapter the models are optimized in this chapter. The gathered data is analyzed first to gain insights and relations between data points. Afterwards, the models' architecture and parameters are adjusted to perform better on the validation dataset.

6.1 Data Analysis

Before diving into model adjustments, it is clever to take at look at the data and check if it is reasonable. Therefore, the benchmarks' behavior with different power caps during characterization phase is evaluated and if the target variable is equal for the same execution phase. Figure 6.1 shows the target variables of three benchmarks for all power caps on two different systems. It can be seen that overall the target variables are quite close to each other. Except for a few outliers the data points form a single or two lines. In the cases of two lines, e.g. in Subfigure 6.1b, it is supposed that the exact optimal power limit is between those two values.



(a) Target Variable of Ember **(b)** Target Variable of BT **(c)** Target Variable of OFOAM

Figure 6.1: Target Variables for Different Benchmarks on Different Nodes

However, it can be seen in Subfigure 6.1a that the ECP Proxy Apps are not using all nodes the measurement is running on. This is due to a mistake in the execution scripts as not enough processes are spawned to fully utilize the systems. However, these data points are not useless. They are characterizing nodes in idle state which can be capped on lowest power. The models can therefore also learn how to react to this kind of behavior. Nevertheless, this mistake is fixed, and additional data is collected for the ECP Proxy Applications fully utilizing their resources.

Clustering

Clustering is used to gain further insights about the dataset. Similar workload characteristics should result in the same cluster. Therefore, the majority of a benchmark's data points are expected to be in the same clusters. **DBSCAN** is chosen as clustering algorithm.

DBSCAN is a density-based clustering algorithm and creates clusters based on a threshold number of neighbors *minPts* and a radius ϵ. Both are parameters that can be adjusted when using this algorithm. Data points that have more than *minPts* neighbors within their radius of ϵ are called *core points*. All these neighbors within this radius of a core point are assigned to the same cluster. In case, these neighbors are also considered core points their neighbors are assigned to this cluster as well. [SSE+17]

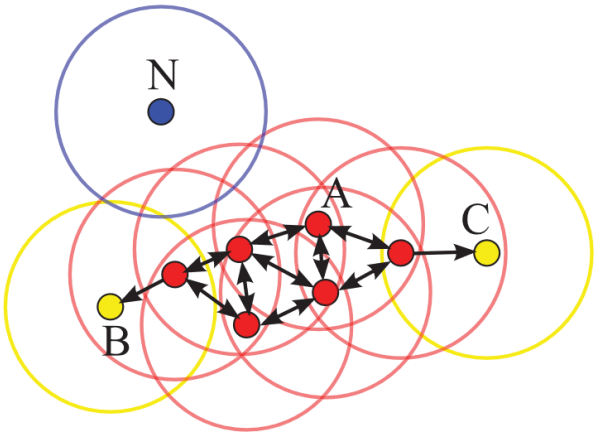


Figure 6.2: Illustration of DBSCAN [SSE+17]

Figure 6.2 illustrates the principle of DBSCAN. The radius ϵ for each point is illustrated by a circle and the minimum number of neighbors to be a core point is equal to four. *N* is a noise point and is the only point that is not assigned to the cluster. *B* and *C* are both border points as they do not exceed the *minPts* threshold of neighbors. All other points including *A* are core points.

Before conducting DBSCAN [1] clustering with the help of scikit-learn a scaler has to be run on the dataset. It standardizes the set by transforming the data to Gaussian distributions with zero mean and a unit variance. This is done because many machine learning estimators in scikit-learn achieve bad results if the features are not normally distributed data. Standardization therefore is

[1] scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html, Accessed: 9th Feb, 2022

often a requirement using scikit-learn estimators. A *StandardScaler* [2] is used to perform this task. DBSCAN formed 124 clusters for the 96,192 data points using an epsilon value of *1.5* and a *minPts* of five.

To see how each benchmark is assigned to a cluster a Sankey diagram is created and shown in Figure 6.3. On the left of the chart the benchmarks are listed whereas the clusters are listed on the right. The output flows show which clusters a benchmark is assigned to. Additionally, the thickness of the flow indicates how often a benchmark is assigned to the cluster. To simplify the understanding and increase readability, clusters which contain only one benchmark are merged with others that contain only the same benchmark. Performance counters are used as input values to the clustering algorithm. Therefore, it can be expected that workloads which are behaving similar should be assigned to the same cluster.

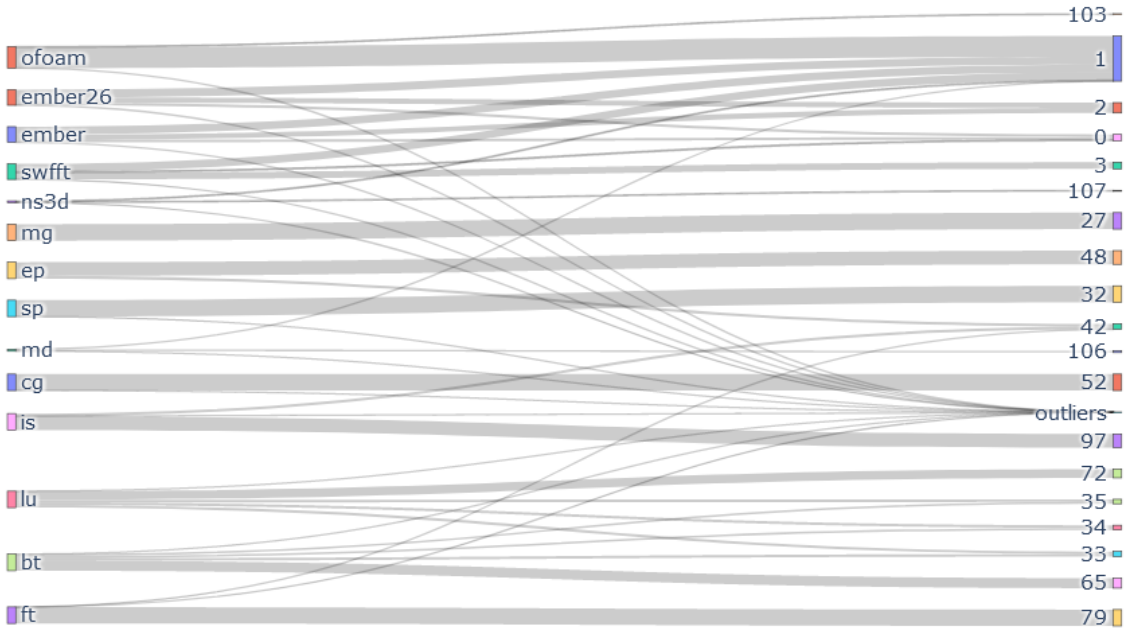


Figure 6.3: Clustering of Benchmark Interval Characteristics

The Sankey diagram shows that *ember26*, *ember* and *swfft* are often assigned to cluster 1. This cluster is containing idle phases in which no real workload is done on the nodes and the optimal power level is at the lowest. It can also be seen that Figure 6.3 has only a few intersections of flows except for flows to the outlier cluster. However, it can be seen that its bar is not very thick and therefore only a few data points are defined as noise. Overall one can see that the benchmarks can be distinguished in the dataset quite well.

Another point interesting to mention is that, changing the value of the radius ϵ to *2.0* results in different clusters. The OpenFOAM benchmark is assigned mainly to cluster 1. An illustration of this assignment can be found in Appendix B by another Sankey diagram. This means that the characteristics of the OpenFOAM benchmark overall are closer to idle state than to other benchmarks. OpenFOAM is the only benchmark which is not part of the training and test dataset as it is contained

[2] scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html, Accessed: 9th Feb, 2022

in the validation dataset. It is therefore new to the models. The models often seems to consider it an idle phase and predict low power caps while the actual power cap is higher. This might be the reason why the OpenFOAM results for the models are so bad.

Data Mixture

Next on the models' behavior is checked in case a few data points of the OpenFOAM application are provided during training to it. 300 of them are added to the training and test datasets and train the models with it. For the random forest regression a maximum depth of 16 and 4096 trees are used. The neural network stays unchanged.
The results for both models look much better for the validation dataset whereas the testing error only slightly increases. Mean absolute error for OpenFOAM is decreased from 33,350 to 9,733 for random forest regression. For the neural network the mean absolute error decreased from 29,925 to 17,220.

It can be concluded that the models did not overfit the testing data. Instead, the models did just not know how to handle these datasets. However, the results of the regression using the neural network are not yet satisfying and further investigation is planned.

Data Set Expansion

Additional applications are provided by the HLRS. They are integrated in the data collection system, and the workloads are included in the dataset. Namely, a *molecular dynamics simulation* and a *NS3Dneo* application are added. Both are explained in Subsections 4.2.4 and 4.2.5.

Another point of interest is to adjust time intervals in the data collection process. The question is if the lengths of these intervals have an impact on the system. The characterization interval t_1 is extended from *10* seconds to *60*, and *120* seconds. For the performance interval t_2 it is decided to adjust it from *15* seconds to *120*, and *300* seconds. The resulting interval setups and number of data points collected can be seen in Table 6.1 for each benchmark.

t_1	t_2	BT	CG	EP	FT	IS	LU	MG
10	*15*	7,920	7,920	7,920	7,920	7,920	7,920	7,920
60	*120*	864	720	864	1,008	864	864	1,008
120	*300*	1,008	864	864	720	1,008	720	864
t_1	t_2	SP	Ember	Ember26	SWFFT	TGV	NS3Dneo	MD
10	*15*	7,920	7,488	7,488	7,488	10,368	864	864
60	*120*	1,008	1,440	1,440	1,440	1,512	864	864
120	*300*	864	1,344	1,296	1,296	1,080	864	1,296

Table 6.1: Number of Data Points per Time Interval Configuration

Data Set Normalization

Next on, it is checked if it is possible to normalize the time intervals in the data and merge the benchmark data. If this is the case, one machine learning model for all interval lengths can be used as they would not depend on this value.

To do so, the metrics collected during the characterization phase are divided by the length of this interval in seconds t_1. In case the data could be normalized, using the clustering algorithm of earlier sections the same benchmark would be assigned to the same cluster. Figure 6.4 shows the clustering results of the FT benchmark which is part of the NPB suite and the Ember benchmark from ECP proxy applications. Furthermore, the figure shows how each normalized interval of the Molecular Dynamics and OpenFOAM applications are clustered.

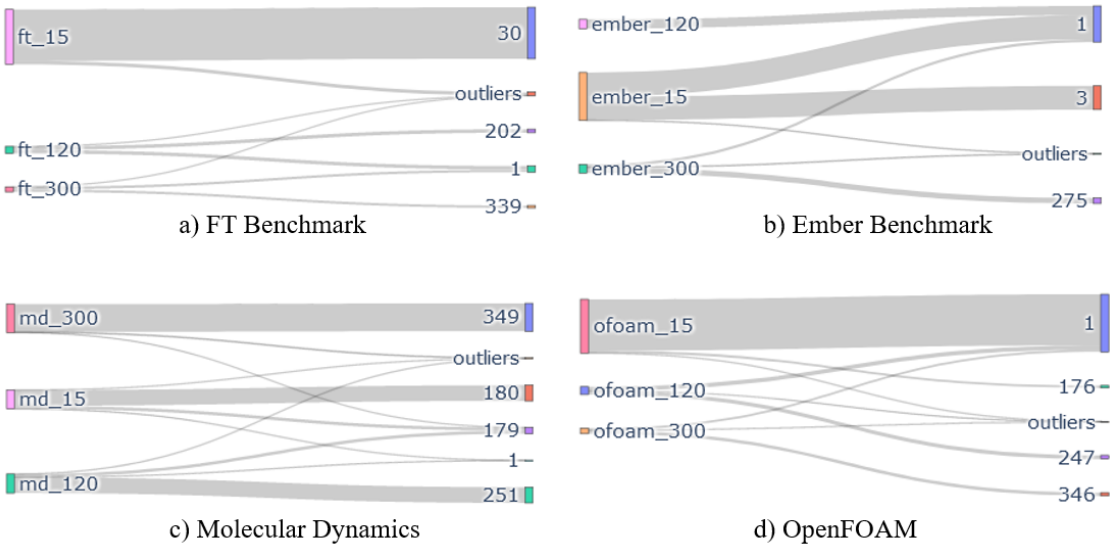


Figure 6.4: Clusters of Normalized Benchmark Data

One can see that the workloads' normalized intervals are strongly split into different clusters. Some clusters contain multiply interval lengths, e.g. cluster 179 in Subfigure 6.4c. However, these are the minority and consist of fewer data points which is illustrated by the thickness of the flows. Clusters that contain only one interval length consist of more data points. For instance, cluster 180 in Subfigure 6.4c for the Molecular Dynamics application has more data points than cluster 179. Therefore, it is concluded that it is not possible for us to normalize and merge the data based on the interval lengths.

6.2 Regularization

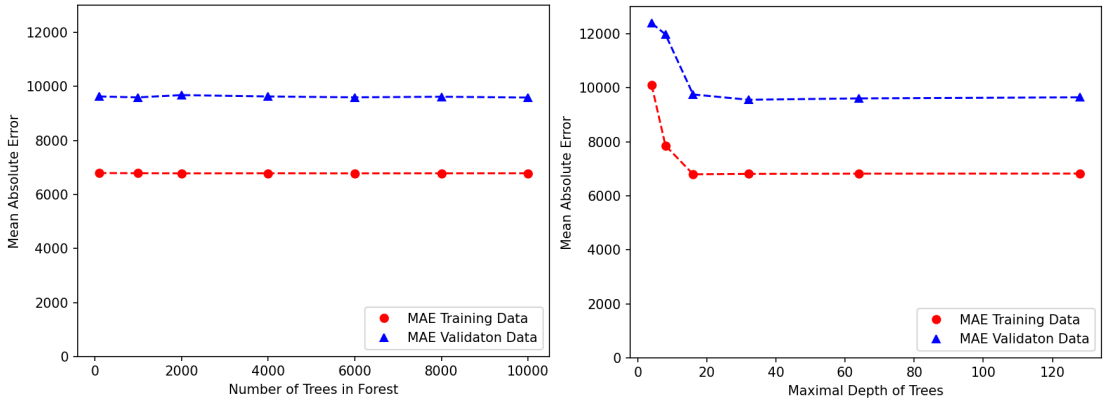
When it comes to reduce overfitting the terms **generalization** and **regularization** are often used. Generalization describes the ability of a model to perform well on unknown data. Its measurement is the generalization error which is the error of predictions on data not used during the training process. As the same benchmarks are used in the training and the test set so far, the validation set is the only dataset in this use case that is completely unknown to the model. Therefore, the

generalization error is conducted on this data.
Regularization describes any modification to the learning algorithm to reduce the generalization error. However, it aims not to influence the error on the training dataset. [GBC16] [JGR19]

6.2.1 Random Forest Regression

The results for unknown datasets are not yet good enough. To check if the model architecture is responsible for overfitting of the model, a deeper look at the parameters of the random forest regression is taken. The dataset with the characterization interval t_1 equal to 10 seconds and the performance interval t_2 equal to 15 seconds are utilized. A GridSearch was already conducted on the parameters of the random forest regression, however it did not consider the error on the validation dataset.

As previously stated in Section 3.2, the authors of [Bre01] prove that random forests do not overfit by adding more trees to the model. In some cases adding more models to the overall model can even decrease the generalization error. This concept is used in **boosting** which combines several models to average their output. Thereby all models will usually not do all the same errors on test data. However, it comes with the price of high computation and memory costs. [GBC16]
To prove this concept within the scenario random forests with *100*, *1,000*, *2,000*, *4,000*, *6,000*, *8,000*, and *10,000* trees are created. No changes in the model performance can be seen. All models had a mean absolute error of approximately 6,780 for the test set and approximately 9,600 for the validation set. The results are illustrated in Subfigure 6.5a. However, the run times of the forests did increase significantly. A forest with small numbers of trees is therefore used. As the validation error of the forest with a number of **1,000** trees is slightly smaller than for the 100 and 2,000, it seems better to use this parameter value for the random forest regression.



(a) Number of Trees in Forest **(b)** Maximum Depth of Trees

Figure 6.5: MAE for Different Hyperparameters of Random Forest

The next parameter to take a look at is the maximum depth for the trees. Several forests with different parameter values are executed for `max_depth`. Namely, the list of $[4, 8, 16, 32, 64, 128]$. The corresponding mean absolute errors can be seen in Subfigure 6.5b. It can be seen that the minimum is somewhere between a maximum depth of 16 and 32 for the mean absolute error for both datasets. Using a maximum of 16 results in an error of approximately 6,795 for the training

and 9,752 for the validation dataset. For a maximum depth of 32 these values are 6,811 and 9,557. The differences between these values are rather small. Hence, it is decided to use a maximum depth of **16** as this reduces the computational overhead out of both options.

For the `max_features` parameter two trees with maximum depth of 16 and 1,000 trees are run using both possible parameters — `'sqrt'` and `'auto'`. It defines how many features are considered when a tree has to be split. The difference between the performance of both is slightly as the forest with 'sqrt' parameter has a mean absolute error of 6,791 for the test set and 10,541 for the validation set. The other forest performed slightly better with numbers of 6,783 for the test and 9,625 for the validation dataset. Performances of both forests are quite similar. However, as the forest using 'auto' for `max_features` performs slightly better at generalization, this parameter is utilized from now on.

6.2.2 Neural Network

Target Variable Normalization

During investigation of the neural network model it is discovered that the model has quite small outputs in the beginning of the training process. The maximum reached values were in ranges of hundreds. However, the desired value range is 100,000 to 200,000. It is possible to either use initial weights of that scale or modify the architecture to transform output values to this range. For the first option one decides to assign defined values to the neural network's weights. The model then uses these weights as starting point for the training process. However, in the scenario this comes with the price of increased execution cost as values in ranges of hundred thousand are used. Furthermore, values of this size increase the space which is needed in memory. Therefore, the architecture of the neural network is adjusted.

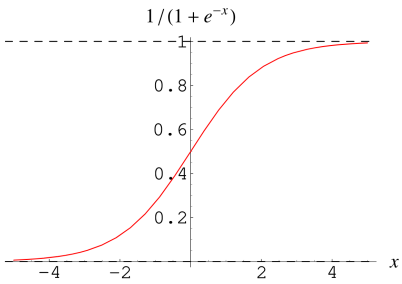


Figure 6.6: Sigmoid Function [Wei22]

First, as it is desired to have values in a certain range, a decision is made to utilize the Sigmoid function to set limits for the output. This function takes input values of infinitely negative or positive size and projects them on values between zero and one. [Wei22]

$$S(x) = \frac{1}{1 + e^{-1}}$$

Figure 6.6 illustrates the shape of the Sigmoid function. This function is applied to the output of the neural network. To set the output limits to the desired values the output of the Sigmoid function is multiplied with 100,000 and add a value of 100,000 on top. The new function looks therefore as the following:

$$f(x) = \left(\frac{1}{1 + e^{-1}} * 100\,000\right) + 100\,000 \quad , f(x) : [-\infty; +\infty] \mapsto [100\,000; 200\,000]$$

Another option to face this issues is to normalize the dataset. Meaning, the input variables are transformed to values between zero and one. The network will learn to predict values in this data range. After training, when using the model for predicting power cap sweet spots the outputs are transformed back to values in the actual value range.

An experiment is run to see which constellation performs best using the test and training dataset for a performance interval t_2 equal to 15 seconds. To do so after each epoch the mean absolute error for the test dataset is taken. This experiment is run ten times and take the average of these runs for each model. Figure 6.7 illustrates the results.

The black dashed curve shows how the old neural network performs. It's training error reduces significantly during first 50 epochs and reduces in a smaller degree afterwards.
The model design which introduces the output transformation layer and the sigmoid function performs the worst on the training dataset. It is illustrated by the green dotted curve. Training error does not reduce for this model.
The second approach of normalizing the dataset can be seen in the Figure by the red solid and blue dashed-dotted curves. Difference between both models is that the latter does include a sigmoid function in last layer. Both do not improve significantly, however reduce the training error in a small but steady way. However, the blue dashed-dotted curve does overall perform best compared to all other models.

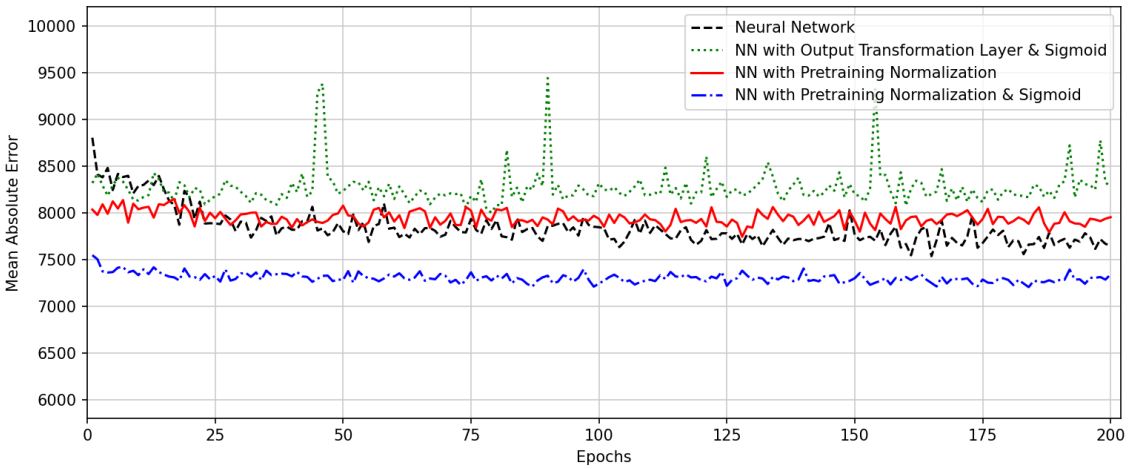


Figure 6.7: Mean Absolute Error of Different Data Transformation Methods

Nevertheless, one can see that after 25 epochs the training error declines rather slowly for all curves. Only the black dashed curve further reduces its training error further. However, never reaches error values the blue dashed-dotted curve achieves. Therefore, and as **early stopping** is a great concept of avoiding overfitting for a neural model, it is decided not to stay at 200 epochs. Instead, the number of epochs is reduced to 25 which furthermore reduces the workload for the model training process as side effect. From now on, the model represented by the blue curve is used. It performs variable normalization and uses a sigmoid function.

Hyperparameter Tuning

A neural network has a lot of hyperparameters that impact the performance of the model. Hyperparameters are variables that are defined before training the model. For example, the number of layers in a neural network is a hyperparameter. Furthermore, the numbers of neurons in each layer are additional hyperparameters. Finding the best settings for the network's hyperparameters a hyperparameter tuning is conducted. Tune is chosen which is explained in Subsection 3.4.4. It is a

framework for hyperparameter tuning and integrates well with PyTorch which is used to implement the neural network. It is possible to train several neural networks with different hyperparameters at the same time and compare their performance. Table 6.2 lists the hyperparameters of scope and which possible values are provided to them. The number of hidden layers, each of their sizes, the learning rate of the optimization function, and the batch size used in the training process are checked.

Hyperparameter	Value Range
Number of Hidden Layers	2, 4, 6, 8
Layer Sizes	32, 64, 128, 256, 512
Learning Rate	0.1, 0.01, 0.001, 0.0001
Batch Size	2, 4, 16, 64, 128

Table 6.2: Hyperparameter Configuration

To execute the hyperparameter tuning a training function has to be implemented and a configuration of these hyperparameters has to be created. The creation of the configuration can be seen in Listing 6.1. For number of hidden layers, learning rate, and batch size `tune.choice()` is used which picks one value from the provided list. `tune.sample_from()` makes it possible to create own functions which create a hyperparameter configuration. It is used for the size of each layer to define a list of values that match the number of hidden layers. This way each hidden layer can have a different width. `layer_size` is therefore a list that contains as many integers as the number of hidden layers which is represented by `spec.config.nr_of_hidden_layers` in Listing 6.1. Training function and hyperparameter configuration are provided to `tune` as parameters to the `tune.run()` function which then executes the training function several times with different sets of the configuration. A random search is conducted which samples from the configuration and define it to do so 1,000 times.

```
config = {
    "nr_of_hidden_layers": tune.choice([2, 4, 6, 8]),
    "layer_size": tune.sample_from(lambda spec: random.choices(
                                    [32, 64, 128, 256, 512],
                                    k=spec.config.nr_of_hidden_layers)),
    "lr": tune.choice([1e-1, 1e-2, 1e-3, 1e-4]),
    "batch_size": tune.choice([2, 4, 16, 64, 128])
}
```

Listing 6.1: Ray Tune Hyperparameter Configuration

Tune allows us to use trial schedulers which fasten the hyperparameter tuning process. They terminate bad trials and may also alter hyperparameters of already running trials. A recommended trial scheduler is **Asynchronous Successive Halving Algorithm (ASHA)**. It is introduced in [LJR+20] and the authors describe it as parallel, aggressively early-stopping way for large scale hyperparameter tuning. The concept of successive halving is quite simple as it first evaluates all configurations and keeps only a portion of the best performing ones. In case of halving, it would be the best performing half of configurations. However, one can set the fraction to continue working with to a desired value. This step is then repeated several times. ASHA is a parallel and asynchronous implementation of this successive halving algorithm. Hence, compute resources can be utilized completely and the overall tuning time is minimized.
The ASHA scheduler is defined as seen in Listing 6.2. The configuration evaluation is based on

training iterations as this value is passed in `time_attr` to the scheduler. Each trial is evaluated a maximum number of training iterations equal to the number of the epochs which is 25. The ASHA scheduler tries to minimize the loss and reduces the configurations by factor four. Meaning only a fourth of all configurations are evaluated further. These are the best performing ones. Furthermore, a `grace_period` is defined which is equal to a fifth of the epochs. This means that each configuration has to have 5 iterations before being stopped.

```
scheduler = ASHAScheduler(
    time_attr='training_iteration',
    metric="loss",
    mode="min",
    max_t=EPOCHS,
    grace_period=EPOCHS/5,
    reduction_factor=4
)
```

Listing 6.2: ASHA Scheduler Instantiation

Table 6.3 shows the ten best hyperparameter configurations based on training error. Additionally, the validation error is calculated for the models to decide which configuration seems most suitable for the work. One can see that a lot of models have a learning rate of 0.0001 and a batch size of 2. The small batch size means that more learning steps are being conducted which is the reason for the frequent occurrence of this batch size.
Although the models perform great on the testing data, they have rather bad results on the validation dataset. For instance, the models number nine and ten have an average testing errors of approximately 7,100 whereas their validation error is higher than 40,000. Only three of these ten models have a validation error of less than 30,000. However, it is possible to not consider these models overfitted because of the small batch size. There are two models in the top list that have a batch size of 16 and nearly all models use the smallest learning rate provided. Although there are more steps performed with small batch sizes, the small learning rate makes these steps small.

No.	Layers	Learning Rate	Batch Size	Test Error	Validation Error
1	[128, 64, 256, 256, 128, 64, 256, 64]	0.0001	2	6,984	27,656
2	[32, 512, 128, 512]	0.0001	2	7,051	34,263
3	[256, 32, 128, 512, 64, 32]	0.0001	2	7,060	30,927
4	[512, 512, 64, 512, 32, 64, 64, 512]	0.0001	16	7,071	29,589
5	[512, 512, 512, 256, 512, 128]	0.0001	2	7,082	33,530
6	[256, 512, 128, 256, 256, 64, 512, 128]	0.0001	2	7,083	35,041
7	[256, 256, 512, 32, 128, 256]	0.001	16	7,107	29,852
8	[512, 512, 64, 32, 128, 256, 512, 128]	0.0001	2	7,111	31,369
9	[256, 64, 256, 32]	0.0001	2	7,121	43,963
10	[512, 256, 64, 32]	0.0001	2	7,129	67,726

Table 6.3: Best Hyperparameter Configurations

Model number one is taken as it is the best model for the test dataset and also the best for the validation dataset. It has 8 layers, a learning rate of 0.0001, and a batch size of 2. For the number of neurons in each layer the values from Figure 6.3 are taken which are 128, 64, 256, 256, 128, 64, 256, and 64.

7 Experiment

In this chapter the machine learning models introduced earlier are tested in production. The trained models are running simultaneously on a system under workload. They are going to adapt the power cap of the compute system during that time. The dynamical power capped application execution are compared with fixed power caps and executions with no power cap.

7.1 Design

In the experiment all benchmarks introduced so far are executed several times. One time without power cap, once with a power cap at 160W per socket and multiple times with a dynamic power cap predicted by the models. For each interval length previously defined an execution with the random forest regression and another using the neural network are run. The experiment process is illustrated in Figure 7.1. The only steps that differ between each run are marked in gray.

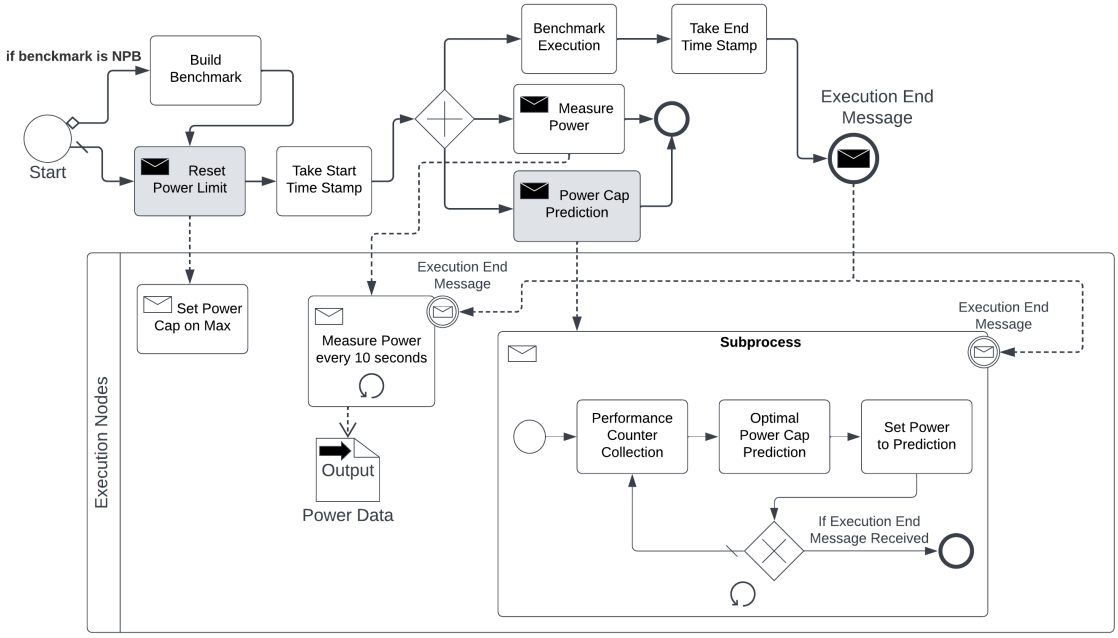


Figure 7.1: Experiment Design

In case a benchmark included in the NPB suite is run the first step is to building the benchmark. Otherwise, this step is omitted, and the process starts with the next task which to set the power limit. In this step which is marked in gray two options are possible. Firstly, power caps are removed in case the execution is conducted without or dynamic power cap. Or secondly, the power cap is set

to a value of 160 Watt for each socket. This is done in case the execution uses a fixed power cap. To do so a message is sent to each node involved in the benchmark execution which then sets the power caps accordingly. Afterwards a time stamp is taken to mark the beginning of the execution interval.

The next steps in the experiment process are executed in parallel. These steps are the actual benchmark execution, power measuring, and the power cap prediction. Power is measured by a separate process on each node. It uses HSMP to get the current power on each socket. This measurement is retrieved every ten seconds and can thereby calculate the overall energy consumption of the execution. The power cap prediction is only conducted when executing dynamic power capping. This task starts a process on each node which is represented in Figure 7.1 by the subprocess.

As mentioned earlier, a random forest and a neural network are created for the time intervals defined earlier. In this subprocess performance counters are collected during the characterization phase t_1 that is 10, 60, or 120 seconds long. The model then predict the optimal power cap for the performance phase t_2 that is 15, 120, or 300 seconds. Similar to the data collection process power caps of 100, 120, 140, 160, 180 and 200 Watts are used. The power is capped to the value which is closest to the cap the model predicted. Afterwards, wait for the end of the performance phase to repeat this process by collecting performance counters.

Finally, when the benchmark finishes its execution another time stamp is taken to calculate the benchmark's execution time. Furthermore, an interrupt signals is sent to the power measuring and power cap prediction processes. These process just terminate as they are no longer needed.

Machine Learning Models

The models used, and their hyperparameters can be seen in Table 7.1. For random forest regression 1,000 trees with a maximum depth of 16 are used. Whenever a tree has to be split all features are considered.
The neural network is trained by an *Adam* optimizer with a learning rate of 0.0001 and a batch size of 2. Its architecture contains eight layers of different sizes and ReLU activation functions. Expect the last layer which uses a Sigmoid activation function. The networks inputs are normalized to values between zero and one and its outputs are retransformed to the value range between 100,000 and 200,000.

	Random Forest	**Neural Network**
Hyperparameters	No. trees: *1,000*	No. Layers: *8*
	Max Depth: *16*	Learning Rate: *0.0001*
	Split Features: *'auto'*	Batch Size: *2*
		Layer Sizes:
		[*128, 64, 256, 256, 128, 64, 256, 64*]
		Optimizer: *Adam*

Table 7.1: Models used in Experiment

The resulting models' performance on the testing and training datasets can be seen in Table 7.2. The models for the performance interval t_2 equal to 15 are performing great on the testing data set. However, they perform rather badly on the validation dataset compared to the models for other interval lengths. For the other interval lengths the models have a mean absolute error between 10,000 and 20,000.

	Random Forest	**Neural Network**
t_2 = 15		
Testing Error	4,892	7,464
Validation Error	34,762	26,662
t_2 = 120		
Testing Error	11,533	12,181
Validation Error	13,901	15,805
t_2 = 300		
Testing Error	14,562	16,241
Validation Error	16,497	19,189

Table 7.2: Validation and Testing Error

Environment

The experiment is executed on the same environment as the data is gathered from, which is described in Section 4.4. Four nodes are used for all application workloads except for the OpenFOAM benchmark for which three nodes are used. The hardware configuration of each node can be seen in Table 7.3. They have two sockets equipped with AMD EPYC 7702 processors. These processors have 64 cores and cache sizes of 512 kilobytes for L2 and about 15 megabytes for L3 cache. The total memory of a single node is about 52 gigabytes.

CPU	**Sockets**	**Cache (L2 / L3)**	**Total Memory**
AMD EPYC 7702 (64 Cores)	2	512 KB / 16384 KB	52 GB

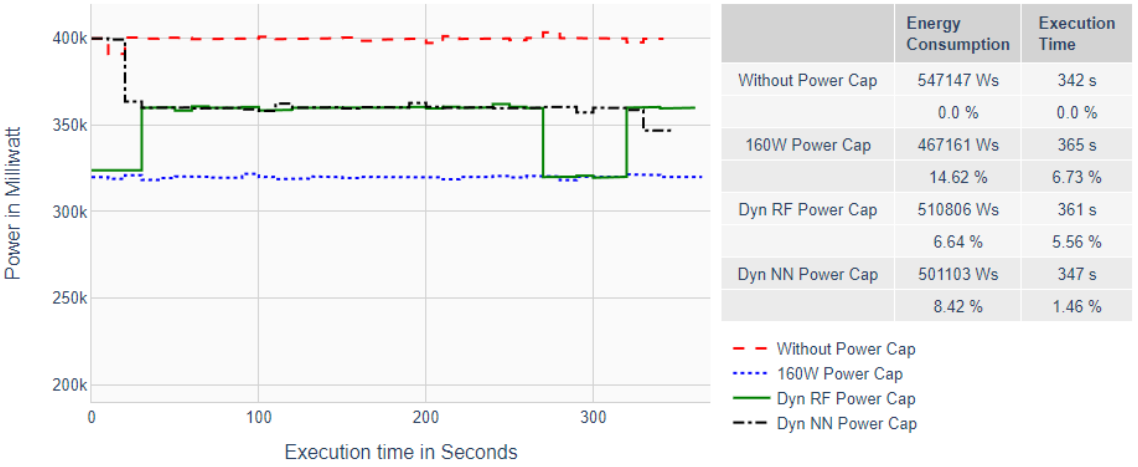
Table 7.3: Experiment Environment

7.2 Results

The experiment is conducted for the FT, BT, LU, SP, Ember, and OpenFOAM benchmarks. Each of them are evaluated for the first performance interval t_2 which is 15 seconds. It has to be mentioned that it has only been possible to run the experiment once. Therefore, the numbers and graphs of these findings are not statistically reliable and have to be observed with care. However, they show a first trend and are a foundation for further steps.

Figure 7.2 shows the power measurements of the FT benchmark on a single node. The other nodes involved in the execution behave the same and are therefore omitted. The red dashed line shows the measurements without power cap and the blue dotted line the power with a fixed power cap.

The power measurements with dynamic power caps that are predicted by the random forest model, are illustrated by the green solid line. Finally, the measurements with power caps predicted by the neural network are shown by the black dashed-dotted line.
The other benchmarks' graphs can be found in the Appendix C.



	Energy Consumption	Execution Time
Without Power Cap	547147 Ws	342 s
	0.0 %	0.0 %
160W Power Cap	467161 Ws	365 s
	14.62 %	6.73 %
Dyn RF Power Cap	510806 Ws	361 s
	6.64 %	5.56 %
Dyn NN Power Cap	501103 Ws	347 s
	8.42 %	1.46 %

Figure 7.2: Power Graphs of FT Benchmark

The overall power consumption of a benchmark can be calculated by the area under each curve. To do so, the average value of the power measurements is taken and multiplied by the total execution time of the benchmark. Finally, these values are summed for each of the nodes involved in the execution. Results of these calculations and the corresponding execution time are seen in Table 7.4 for all benchmarks. Furthermore, the energy savings and performance loss based on execution time can be seen in percentage values.

	FT		**BT**		**LU**	
Without Power Cap	547,147 Ws	342 s	616,061 Ws	385 s	627,496 Ws	392 s
Fixed 160W Power Cap	+14.62%	+6.73%	+8.0%	+15.06%	+11.89%	+10.2%
Dynamic RF Power Cap	+6.64%	+5.56%	+3.57%	+9.35%	+8.43%	+13.27%
Dynamic NN Power Cap	+8.42%	+1.46%	+4.76%	+5.19%	+4.04%	+11.99%
	SP		**Ember**		**OpenFOAM**	
Without Power Cap	627,496 Ws	526 s	8,522,623 Ws	5,327 s	6,984,866 Ws	7,511 s
Fixed 160W Power Cap	+18.93%	+0.57%	+20.1%	-0.11%	+9.74%	+0.6%
Dynamic RF Power Cap	+5.09%	+6.84%	+10.11%	+0.08%	-0.31%	+4.46%
Dynamic NN Power Cap	+7.7%	+1.33%	+10.32%	+0.02%	+2.29%	+3.91%

Table 7.4: Power Consumption & Execution Time of Benchmarks with Interval Time of 15s

One can see that the fixed power cap of 160W overall saves most power. For the SP and Ember benchmarks power savings of 20 percent are reached. Its execution time is most of the time longer compared to runs without power cap. This is the case as the computational performance is degraded with decreased power. However, for the Ember and OpenFOAM benchmarks the fixed cap is close to the execution performance without power cap or even faster. Again, it has to be emphasized that it has only been possible to run this experiment once and have to say that this might be the case due to statistical noise.

The dynamic power caps perform best for the FT and BT benchmark. Especially, the power caps that are predicted by the neural network model look promising. Although they do not reach the power savings of executions with a fixed power cap, the performance loss is significant smaller compared to fixed power caps.

Nevertheless, one can see that the dynamically capped executions often take longer than the fixed power cap ones, although the latter have lower caps. This is due to the fact that the dynamic power capping mechanism introduces additional workload. Computational effort is necessary to predict the optimal power cap. This overhead is expected to be reduced when running this experiment with greater time interval lengths.

8 Conclusion

In this last chapter this work is summarized, and a conclusion is drawn. Afterwards, possible future steps are listed.

8.1 Summary

During this work an introduction to the current state of literature is presented. Work that aims to achieve similar goals as this work is evaluated and compared. Afterwards, a dataset is created based on performance counters. It aims to capture the characteristics of a workload and a corresponding power cap sweet spot. Meanwhile, two machine learning models are designed which shall predict these power caps. These models are a random forest regression and a neural network. They are trained based on the created dataset, optimized, and adjusted to achieve generalization on unknown data. Finally, these models are used in an experiment to set power caps during a benchmark execution.

8.2 Conclusion

The aim of this work is to create machine learning models which can predict energy-efficient power caps. These models shall save energy with only few impacts on a workload's performance. Prediction results of the machine learning models look promising. The models perform well on the testing dataset with error values around ten percent of the value data range.
Executions in the experiment with predicted dynamic power caps perform well in some cases. For example the power caps predicted by the neural network save 8% of energy while reducing the performance by less than 2%. It has to be mentioned that it has not been possible to execute the experiment several times. Hence, the numbers and graphs in this case are not statistically correct. However, they show a trend and dynamic power caps predicted by machine learning models have the potential to save power with few impact on performance.

Although the executions with a fixed power cap perform overall better in the experiment, it is not recommend using fixed power caps for all workload types. In the experiment only MPI based workloads are executed which use several messages as means of communication between processes. In these scenarios a reduced fixed power cap can save energy without a great impact on performance. However, when conducting shared memory or hybrid workload an increase in power does also result in an increasing performance. A fixed power cap can therefore reduce a system's performance significantly in these cases.

8.3 Future Work

In the future it is possible to adapt this work to not use only on-chip power sensors. External power meters have higher accuracy and are considered ground of truth concerning power. Additionally, it is possible to extend the work to power cap predictions of additional hardware components, for instance graphical processing units.

Furthermore, running the experiment several times to justify the findings is considered. It is of interest to see how the system behaves in case of extended interval lengths. The workload introduced by prediction computation might also be investigated.

Finally, further machine learning algorithms might be evaluated. They can be compared concerning accuracy, as well as computational overhead. A deeper look into the performance counter selection can be done. It is possible to check which of these counters has the most impact on the power cap prediction. Or it might be interesting to see if there are any metrics that might even be irrelevant for this use case.

Bibliography

[Adv19] Advanced Micro Devices Inc. *Available performance monitors for the AMD® Zen2 microarchitecture*. 2019. URL: https://github.com/RRZE-HPC/likwid/blob/master/doc/archs/zen2.md (cit. on p. 37).

[Adv21] Advanced Micro Devices Inc. "Preliminary Processor Programming Reference (PPR) for AMD Family 19h Model 01h, Revision B1 Processors Volume 2 of 2". In: (2021). URL: developer.amd.com/resources/epyc-resources/epyc-specifications (cit. on pp. 19, 37).

[All+04] M. P. Allen et al. "Introduction to molecular dynamics simulation". In: *Computational soft matter: from synthetic polymers to proteins* 23.1 (2004), pp. 1–28 (cit. on p. 35).

[AS95] C. G. Atkeson, S. Schaal. "Memory-based neural networks for robot learning". In: *Neurocomputing* 9.3 (1995), pp. 243–269. ISSN: 09252312. DOI: 10.1016/0925-2312(95)00033-6 (cit. on p. 25).

[Bab09] A. Babucke. "Direct Numerical Simulation of Noise-Generation Mechanisms in the Mixing Layer of a Jet". PhD thesis. May 2009. DOI: 10.13140/RG.2.2.11798.19523 (cit. on p. 36).

[BBE+21] P. Bailey, S. Brink, D. Ellsworth, A. Marathe, L. Morita, T. Patki, B. Rountree, K. Shoga, S. Walker. *Variorum Documentation: Release 0.4.1*. 2021. URL: variorum.readthedocs.io (cit. on p. 24).

[Ben09] Y. Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009 (cit. on p. 29).

[Ber19] D. Berrar. "Cross-Validation". In: *Encyclopedia of Bioinformatics and Computational Biology*. Elsevier, 2019, pp. 542–545. ISBN: 9780128114322. DOI: 10.1016/B978-0-12-809633-8.20349-X (cit. on p. 41).

[Bre01] L. Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 08856125. DOI: 10.1023/A:1010933404324 (cit. on pp. 25, 26, 50).

[Bun20] S. Bundesamt. *GREEN500*. 2020. URL: https://www.destatis.de/EN/Themes/Society-Environment/Environment/Material-Energy-Flows/Tables/electricity-consumption-households.html (cit. on p. 17).

[CKMC21] M. Cui, A. Kritikakou, L. Mo, E. Casseau. "Fault-Tolerant Mapping of Real-Time Parallel Applications under multiple DVFS schemes". In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 18.05.2021 - 21.05.2021, pp. 387–399. ISBN: 978-1-6654-0386-3. DOI: 10.1109/RTAS52030.2021.00038 (cit. on p. 21).

[CM04] V. Cherkassky, Y. Ma. "Comparison of loss functions for linear regression". In: *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*. Vol. 1. IEEE. 2004, pp. 395–400 (cit. on pp. 42, 43).

[ESC+17] J. Eastep, S. Sylvester, C. Cantalupo, B. Geltz, F. Ardanaz, A. Al-Rawi, K. Livingston, F. Keceli, M. Maiterth, S. Jana. "Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions". In: *High Performance Computing*. Ed. by J. M. Kunkel, R. Yokota, P. Balaji, D. Keyes. Vol. 10266. Lecture Notes in Computer Science. Springer International Publishing, 2017, pp. 394–412. ISBN: 978-3-319-58666-3. DOI: 10.1007/978-3-319-58667-0-21 (cit. on p. 23).

[Exa22] Exascale Computing Project. *Exascale Proxy Applications*. 2022. URL: https://proxyapps.exascaleproject.org (cit. on p. 34).

[FC07] W.-c. Feng, K. Cameron. "The green500 list: Encouraging sustainable supercomputing". In: *Computer* 40.12 (2007), pp. 50–55 (cit. on p. 17).

[FM17] Fabio Ferrero, Matteo Sonza Reorda. "Analysis and dynamic optimization of energy consumption on HPC applications based on real-time metrics". PhD thesis. POLITECNICO DI TORINO, 2017. URL: https://webthesis.biblio.polito.it/6423/1/tesi.pdf (cit. on pp. 23, 36, 69).

[Fra17] France Boillod-Cerneux. *Towards energy consumption application profiling with BULL energy software*. Montpellier, France, October 4-5, 2017. URL: https://public.weconext.eu/eocoe/2017-10-04/video_id_003/index.html (cit. on p. 23).

[FSML19] M. Fahad, A. Shahid, R. R. Manumachu, A. Lastovetsky. "A Comparative Study of Methods for Measurement of Energy of Computing". In: *Energies* 12.11 (2019), p. 2204. DOI: 10.3390/en12112204 (cit. on p. 19).

[GBB11] X. Glorot, A. Bordes, Y. Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 315–323 (cit. on p. 27).

[GBC16] I. Goodfellow, Y. Bengio, A. Courville. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016. ISBN: 9780262035613 (cit. on pp. 25–27, 29, 43, 50).

[GBI21] M. Gupta, L. Bhargava, S. Indu. "Dynamic workload-aware DVFS for multicore systems using machine learning". In: *Computing* 103.8 (2021), pp. 1747–1769. ISSN: 0010-485X. DOI: 10.1007/s00607-020-00845-2 (cit. on p. 21).

[GEO21] GEOPM Working Group. *GEOPM Service Documentation*. 2021. URL: https://geopm.github.io (cit. on p. 23).

[Gre16] C. Greenshields. *Computational Fluid Dynamics*. 2016. URL: https://cfd.direct/openfoam/computational-fluid-dynamics (cit. on p. 35).

[Gre21] C. Greenshields. *OpenFOAM User Guide Version v2112*. 2021. URL: https://www.openfoam.com/documentation/overview (cit. on p. 35).

[HD18] S. A. Hollingsworth, R. O. Dror. "Molecular dynamics simulation for all". In: *Neuron* 99.6 (2018), pp. 1129–1143 (cit. on p. 35).

[HGCB08] M. Hersch, F. Guenter, S. Calinon, A. Billard. "Dynamical system modulation for robot learning via kinesthetic demonstrations". In: *IEEE Transactions on Robotics* 24.6 (2008), pp. 1463–1467 (cit. on p. 25).

[HPGJ14] J.-P. Halimi, B. Pradelle, A. Guermouche, W. Jalby. “FoREST-mn: Runtime DVFS beyond communication slack”. In: *International Green Computing Conference*. IEEE, 11/3/2014 - 11/5/2014, pp. 1–6. ISBN: 978-1-4799-6177-1. DOI: 10.1109/IGCC.2014.7039158 (cit. on p. 22).

[HZS06] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew. “Extreme learning machine: theory and applications”. In: *Neurocomputing* 70.1-3 (2006), pp. 489–501 (cit. on pp. 28, 29).

[Int12] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2012. URL: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf (cit. on p. 22).

[Int21a] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2021. URL: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/custom-analysis/custom-analysis-options/hardware-event-list/instructions-retired-event.html (cit. on p. 37).

[Int21b] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (cit. on pp. 19, 20).

[IPK21] S. Imambi, K. B. Prakash, G. Kanagachidambaresan. “PyTorch”. In: *Programming with TensorFlow*. Springer, 2021, pp. 87–104 (cit. on p. 30).

[ISG07] S. Irani, S. Shukla, R. Gupta. “Algorithms for power savings”. In: *ACM Transactions on Algorithms* 3.4 (2007), p. 41. ISSN: 1549-6325. DOI: 10.1145/1290672.1290678 (cit. on pp. 20, 21).

[Jas09] H. Jasak. “OpenFOAM: open source CFD in research and industry”. In: *International Journal of Naval Architecture and Ocean Engineering* 1.2 (2009), pp. 89–94 (cit. on p. 35).

[JGR19] D. Jakubovitz, R. Giryes, M. R. Rodrigues. “Generalization error in deep learning”. In: *Compressed Sensing and Its Applications*. Springer, 2019, pp. 153–193 (cit. on p. 50).

[KB14] D. P. Kingma, J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 43).

[KCCC08] J. Kong, J. Choi, L. Choi, S. W. Chung. “Low-Cost Application-Aware DVFS for Multi-core Architecture”. In: *2008 Third International Conference on Convergence and Hybrid Information Technology*. IEEE, 11.11.2008 - 13.11.2008, pp. 106–111. ISBN: 978-0-7695-3407-7. DOI: 10.1109/ICCIT.2008.124 (cit. on p. 21).

[LBH15] Y. LeCun, Y. Bengio, G. Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444 (cit. on p. 29).

[LJR+20] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur, M. Hardt, B. Recht, A. Talwalkar. “A system for massively parallel hyperparameter tuning”. In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 230–246 (cit. on p. 53).

[LLN+18] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, I. Stoica. “Tune: A Research Platform for Distributed Model Selection and Training”. In: *arXiv preprint arXiv:1807.05118* (2018) (cit. on p. 31).

[LTF+14] K. Livingston, N. Triquenaux, T. Fighiera, J. C. Beyler, W. Jalby. "Computer using too much power? Give it a REST (Runtime Energy Saving Technology)". In: *Computer Science - Research and Development* 29.2 (2014), pp. 123–130. ISSN: 1865-2034. DOI: 10.1007/s00450-012-0226-0 (cit. on p. 22).

[MS10] G. Martínez-Muñoz, A. Suárez. "Out-of-bag estimation of the optimal sample size in bagging". In: *Pattern Recognition* 43.1 (2010), pp. 143–152 (cit. on p. 26).

[NAS22] NASA Advanced Supercomputing Division. *NAS Parallel Benchmarks*. 2022. URL: https://www.nas.nasa.gov/software/npb.html (cit. on p. 34).

[Nie15] M. A. Nielsen. *Neural Networks and Deep Learning*. 2015. URL: http:/neuralnetworksanddeeplearning.com (cit. on p. 28).

[PM21] J. Pérez Rodríguez, P. Meumeu Yomsi. "An Efficient Proactive Thermal-Aware Scheduler for DVFS-enabled Single-Core Processors". In: *29th International Conference on Real-Time Networks and Systems*. Ed. by A. Queudet, I. Bate, G. Lipari. New York, NY, USA: ACM, 4072021, pp. 144–154. ISBN: 9781450390019. DOI: 10.1145/3453417.3453430 (cit. on p. 21).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 30).

[Ray22] Ray Team. *Tune: Scalable Hyperparameter Tuning*. 2022. URL: https://docs.ray.io/en/latest/tune/key-concepts.html (cit. on p. 31).

[RHW86] D. E. Rumelhart, G. E. Hinton, R. J. Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536 (cit. on p. 29).

[RRS+14] T. Rauber, G. Rünger, M. Schwind, H. Xu, S. Melzner. "Energy measurement, modeling, and prediction for processors with frequency scaling". In: *The Journal of Supercomputing* 70.3 (2014), pp. 1451–1476. ISSN: 0920-8542. DOI: 10.1007/s11227-014-1236-4 (cit. on p. 19).

[RTHW14] T. Roehl, J. Treibig, G. Hager, G. Wellein. "Overhead Analysis of Performance Counter Measurements". In: *43rd International Conference on Parallel Processing Workshops (ICCPW)*. 2014, pp. 176–185. DOI: 10.1109/ICPPW.2014.34 (cit. on p. 30).

[Sch20] J. Schmidt-Hieber. "Nonparametric regression using deep neural networks with ReLU activation function". In: *The Annals of Statistics* 48.4 (2020), pp. 1875–1897 (cit. on p. 26).

[sci22] scikit-learn. *Tuning the hyper-parameters of an estimator*. 2022. URL: https://scikit-learn.org/stable/modules/grid_search.html#exhaustive-grid-search (cit. on p. 41).

[SDSM21] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer. *GREEN500*. 2021. URL: https://www.top500.org/lists/green500/ (cit. on p. 17).

[SFML21] A. Shahid, M. Fahad, R. R. Manumachu, A. Lastovetsky. "Improving the accuracy of energy predictive models for multicore CPUs by combining utilization and performance events model variables". In: *Journal of Parallel and Distributed Computing* 151 (2021), pp. 38–51. ISSN: 07437315. DOI: 10.1016/j.jpdc.2021.01.007 (cit. on p. 19).

[SS15] F. Stulp, O. Sigaud. “Many regression algorithms, one unified model: A review”. In: *Neural networks : the official journal of the International Neural Network Society* 69 (2015), pp. 60–79. DOI: 10.1016/j.neunet.2015.05.005 (cit. on pp. 25–29).

[SSE+17] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, X. Xu. “DBSCAN revisited, revisited: why and how you should (still) use DBSCAN”. In: *ACM Transactions on Database Systems (TODS)* 42.3 (2017), pp. 1–21 (cit. on p. 46).

[THW10] J. Treibig, G. Hager, G. Wellein. “LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments”. In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216. ISBN: 978-1-4244-7918-4. DOI: 10.1109/ICPPW.2010.38 (cit. on p. 30).

[TLP+13] N. Triquenaux, A. Laurent, B. Pradelle, J. C. Beyler, W. Jalby. “Automatic estimation of DVFS potential”. In: *2013 International Green Computing Conference Proceedings*. IEEE, 6/27/2013 - 6/29/2013, pp. 1–6. ISBN: 978-1-4799-0623-9. DOI: 10.1109/IGCC.2013.6604501 (cit. on pp. 21, 22).

[Tor19a] Torch Contributors. *torch.nn: Basic Building Blocks for Graphs*. 2019. URL: https://pytorch.org/docs/stable/nn.html (cit. on p. 31).

[Tor19b] Torch Contributors. *torch.optim: Optimizing Algorithms*. 2019. URL: https://pytorch.org/docs/stable/optim.html (cit. on p. 31).

[Vap99] V. Vapnik. *The nature of statistical learning theory*. Springer science & business media, 1999 (cit. on p. 25).

[WCC14] C.-M. Wu, R.-S. Chang, H.-Y. Chan. “A green energy-efficient scheduling algorithm using the DVFS technique for cloud datacenters”. In: *Future Generation Computer Systems* 37.1 (2014), pp. 141–147. ISSN: 0167739X. DOI: 10.1016/j.future.2013.06.009 (cit. on p. 21).

[Wei22] Weisstein, Eric W. *Sigmoid Function.* 2022. URL: https://mathworld.wolfram.com/SigmoidFunction.html (cit. on p. 51).

[WKE+19] B. Wang, J. Klinkenberg, D. Ellsworth, C. Terboven, M. Müller. “Performance Prediction for Power-Capped Applications based on Machine Learning Algorithms”. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 842–849 (cit. on p. 24).

[WPS+19] C. Wenzel, J. M. Peter, B. Selent, M. B. Weinschenk, U. Rist, M. J. Kloker. “DNS of compressible turbulent boundary layers with adverse pressure gradients”. In: *High Performance Computing in Science and Engineering’18*. Springer International Publishing, 2019, pp. 229–242. ISBN: 978-3-030-13325-2 (cit. on p. 36).

A Event Sets

BDPO Event Set

Event set used by [FM17]:

1. Intructions & Cycles
    - INSTRUCTION_RETIRED
    - READ_TIME_STAMP_COUNTER
    - UNHALTED_REFERENCE_CYCLES
    - UNHALTED_CORE_CYCLES
2. Memory Usage
    - MEM_UOPS_RETIRED:ALLLOADS
    - MEM_UOPS_RETIRED:ALLSTORES
    - L1D:REPLACEMENT
    - L2_TRANS:L1D_WB
    - L2_RQSTS:MISS
3. Package Energy Consumption
    - rapl::RAPL_ENERGY_PKG
    - rapl::RAPL_ENERGY_DRAM

Zen2 Event Groups

Branch
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
RETIRED_BRANCH_INSTR
RETIRED_MISP_BRANCH_INSTR

Cache
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
DATA_CACHE_ACCESSES
DATA_CACHE_REFILLS_ALL

Clock
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
RAPL_PKG_ENERGY

CPI
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
RETIRED_UOPS

Data
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
LS_DISPATCH_LOADS
LS_DISPATCH_STORES

Divide
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
DIV_OP_COUNT
DIV_BUSY_CYCLES

Energy
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
RAPL_CORE_ENERGY
RAPL_PKG_ENERGY

FLOPS_DP
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
RETIRED_SSE_AVX_FLOPS_ALL
MERGE

FLOPS_SP
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
RETIRED_SSE_AVX_FLOPS_ALL
MERGE

iCache
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
ICACHE_FETCHES
ICACHE_L2_REFILLS
ICACHE_SYSTEM_REFILLS

L2 Cache
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
REQUESTS_TO_L2_GRP1_ALL_NO_PF

L3 Cache
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
L3_ACCESS
L3_MISS

Main Memory
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
DRAM_CHANNEL_0
DRAM_CHANNEL_1

Memory DP / SP
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
CPU_CLOCKS_UNHALTED
RETIRED_SSE_AVX_FLOPS_ALL
MERGE
DRAM_CHANNEL_0
DRAM_CHANNEL_1

TLB
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
RETIRED_INSTRUCTIONS
DATA_CACHE_ACCESSES
L1_DTLB_MISS_ANY_L2_HIT
L1_DTLB_MISS_ANY_L2_MISS

NUMA
ACTUAL_CPU_CLOCK
MAX_CPU_CLOCK
DATA_CACHE_REFILLS_LOCAL_ALL
DATA_CACHE_REFILLS_REMOTE_ALL
HWPREF_DATA_CACHE_FILLS_LOCAL_ALL
HWPREF_DATA_CACHE_FILLS_REMOTE_ALL

B Cluster Diagrams

Sankey diagram of benchmark to cluster assignment with $\epsilon = 2.0$ in addtion to the illustration in Chapter 6.

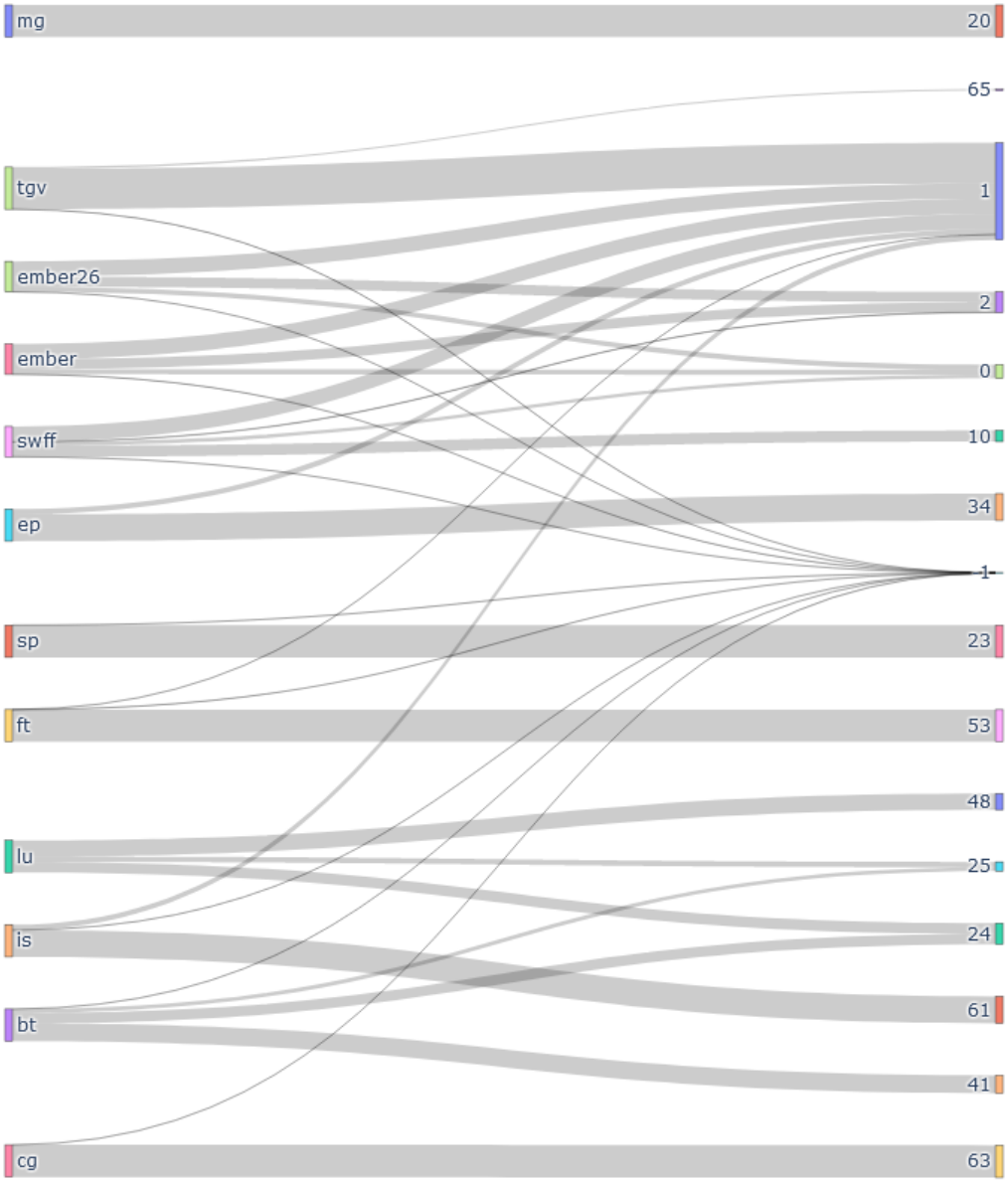
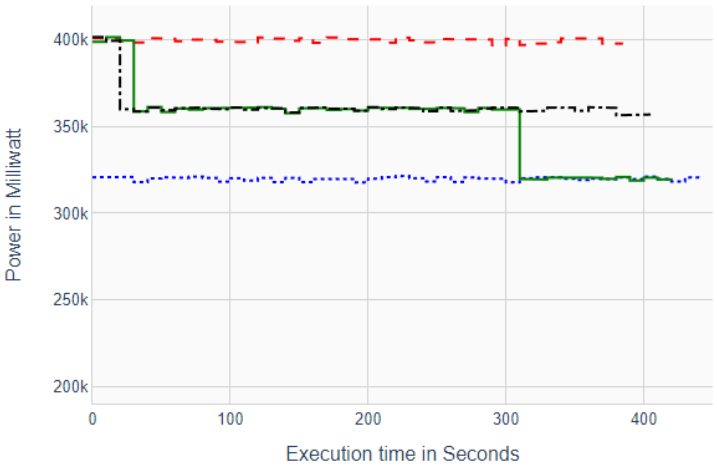


Figure B.1: Benchmark-Cluster Sankey Diagram

C Experiment Results Diagrams

BT Benchmark



	Energy Consumption	Execution Time
Without Power Cap	616061 Ws	385 s
	0.0 %	0.0 %
160W Power Cap	566792 Ws	443 s
	8.0 %	15.06 %
Dyn RF Power Cap	594038 Ws	421 s
	3.57 %	9.35 %
Dyn NN Power Cap	586760 Ws	405 s
	4.76 %	5.19 %

Without Power Cap
160W Power Cap
Dyn RF Power Cap
Dyn NN Power Cap

Figure C.1: Power Graphs of BT Benchmark

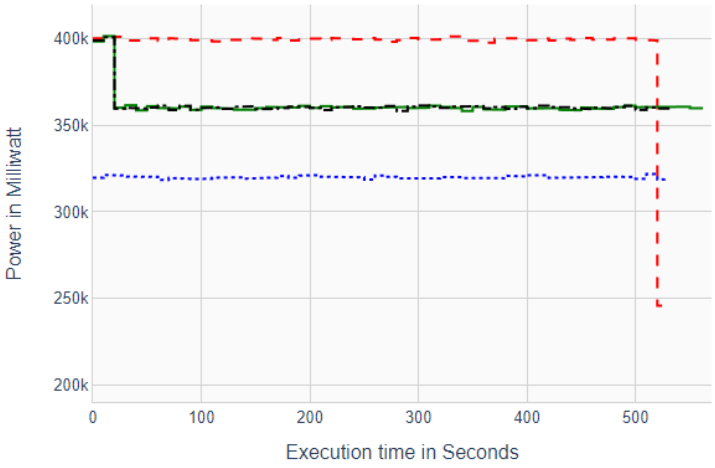
LU Benchmark

	Energy Consumption	Execution Time
Without Power Cap	627496 Ws	392 s
	0.0 %	0.0 %
160W Power Cap	552870 Ws	432 s
	11.89 %	10.2 %
Dyn RF Power Cap	574585 Ws	444 s
	8.43 %	13.27 %
Dyn NN Power Cap	602165 Ws	439 s
	4.04 %	11.99 %

Without Power Cap
160W Power Cap
Dyn RF Power Cap
Dyn NN Power Cap

Figure C.2: Power Graphs of LU Benchmark

SP Benchmark

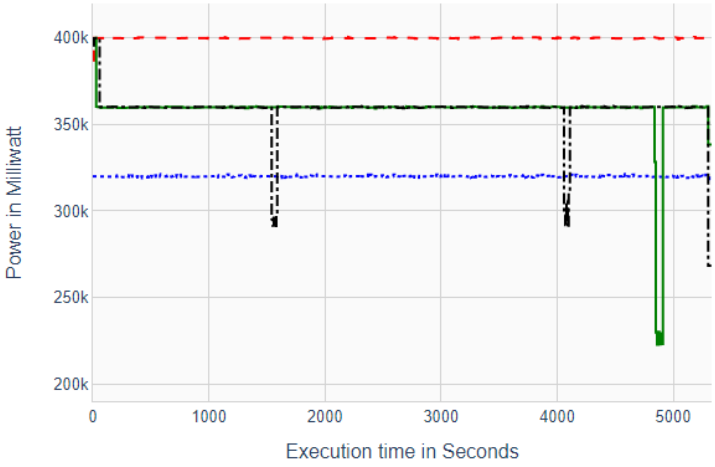


	Energy Consumption	Execution Time
Without Power Cap	835273 Ws	526 s
	0.0 %	0.0 %
160W Power Cap	677164 Ws	529 s
	18.93 %	0.57 %
Dyn RF Power Cap	792794 Ws	562 s
	5.09 %	6.84 %
Dyn NN Power Cap	770973 Ws	533 s
	7.7 %	1.33 %

Without Power Cap
160W Power Cap
Dyn RF Power Cap
Dyn NN Power Cap

Figure C.3: Power Graphs of SP Benchmark

Ember Benchmark



	Energy Consumption	Execution Time
Without Power Cap	8522623 Ws	5327 s
	0.0 %	0.0 %
160W Power Cap	6809533 Ws	5321 s
	20.1 %	-0.11 %
Dyn RF Power Cap	7660736 Ws	5331 s
	10.11 %	0.08 %
Dyn NN Power Cap	7642876 Ws	5328 s
	10.32 %	0.02 %

Without Power Cap
160W Power Cap
Dyn RF Power Cap
Dyn NN Power Cap

Figure C.4: Power Graphs of Ember Benchmark

OpenFOAM Benchmark

	Energy Consumption	Execution Time
Without Power Cap	6984866 Ws	7511 s
	0.0 %	0.0 %
160W Power Cap	6304343 Ws	7556 s
	9.74 %	0.6 %
Dyn RF Power Cap	7006196 Ws	7846 s
	-0.31 %	4.46 %
Dyn NN Power Cap	6824883 Ws	7805 s
	2.29 %	3.91 %

Figure C.5: Power Graphs of OpenFOAM Benchmark

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature