GreenPipe: Energy-Efficient Data-Processing Pipelines for Resource-Constrained Systems

Benedict Herzog
Ruhr University Bochum (RUB)

Jakob Schubert
Fraunhofer Institute for Integrated Circuits (IIS)

Tim Rheinfels
Friedrich-Alexander University Erlangen-Nürnberg (FAU)

Christian Nickel
Fraunhofer Institute for Integrated Circuits (IIS)

Timo Hönig
Ruhr University Bochum (RUB)

Abstract

Billions of resource-constrained systems, such as embedded devices and cyber-physical systems, are in operation worldwide. These systems process input data (e.g., sensor data) into control signals for actuators or human-readable information, thereby providing valuable services and insights. Modern software methods, such as machine learning, have the potential to enhance the performance of these systems even further. However, machine learning is often associated with excessive energy demand, which urgently needs to be resolved. To address this issue, we present GreenPipe, an approach that creates energy-efficient data-processing pipelines tailored for embedded systems known for their low power demand. GreenPipe combines traditional AutoML techniques with energy models and thereby enables the selection of energy-efficient and accurate data-processing pipelines. We implemented GreenPipe on an ARM Cortex-M4 platform and evaluated its performance and energy efficiency. We demonstrate GreenPipe's capabilities through a comprehensive evaluation, including a practical real-world application for predicting machinery-bearing faults. GreenPipe demonstrates that it can reduce the energy footprint by up to 90 % while maintaining high accuracy.

CCS Concepts

• **Hardware** → **Power estimation and optimization**; • **Computer systems organization** → *Embedded systems.*

Keywords

energy efficiency, automated machine learning, embedded systems

1 Introduction

The broad deployment of resource-constrained systems is rapidly gaining momentum. This growth is driven by the widespread adoption of embedded and cyber-physical systems across various application domains, such as production facilities and Industry 4.0 applications [22], smart home automation [34], and wearable systems [8]. A pivotal reason contributing to this momentum is the capability to integrate machine learning within resource-constrained systems. In particular, this capability provides embedded systems with a previously unattainable degree of independence from cloud systems: local data processing and storage that is now possible opens the way to application domains that would otherwise have been unfeasible to realise [1, 24].

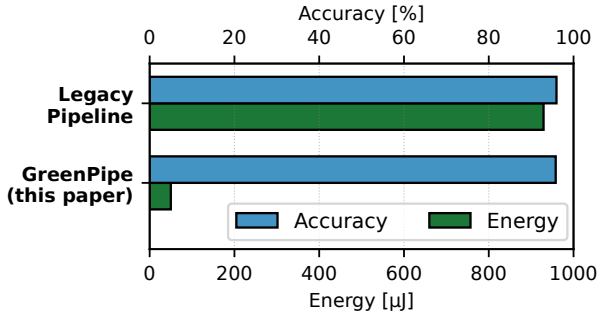


Figure 1: GreenPipe reduces the energy demand of automatically generated data-processing pipelines by up to 90 % (49.9 µJ vs. 929.3 µJ) with almost identical accuracy.

The utilisation of machine learning has significant implications for the energy demand, though. This is particularly critical as many embedded systems [14] need to be energy-efficient due to battery operation or intermittent power supply. Consequently, embedded applications must be designed from the ground up to manage resources, particularly energy, efficiently. While numerous tools and expert knowledge are available for developing such systems with traditional software methods, such tools are often lacking for employing machine learning in embedded systems. Therefore, to fully benefit from machine learning in these contexts, new energy-aware tools are necessary.

An established method for developing a machine learning (ML) application is the design of a data-processing pipeline comprising several small, simple processing steps. These steps are carefully selected and arranged so that the pipeline, as a whole, implements a complex application. Hence, the capabilities and quality of a pipeline are determined by its overall composition. What constitutes a good composition, including the number and types of steps involved, depends on the specific use case. A well-designed pipeline for object classification in images differs significantly from a pipeline that is optimised for Industry 4.0 applications. Consequently, constructing an effective pipeline for a particular use case requires a lot of expert knowledge and significant development time. Automated pipeline generation is an alternative to the manual construction of these pipelines by experts. The pipelines can be generated much quicker and with fewer developers compared to the

manual construction of pipelines. This leads to considerable advantages in cost efficiency for application development. The automated generation of such pipelines is referred to as automated machine learning (AutoML) and is well-researched [7, 11, 21]. However, existing AutoML approaches assume systems with unrestricted power supply and do not consider or optimise for energy demand.

In this paper, we propose GreenPipe, an approach for the automatic generation of energy-efficient data-processing pipelines for resource-constrained systems. Figure 1 illustrates the functional performance (i.e., the ML system's accuracy) and the energy demand of two pipelines, implemented with and without GreenPipe. Both pipelines show nearly identical accuracy (95.8 % vs. 96.0 %), yet their energy consumption differs significantly. The legacy pipeline requires 929.3 µJ, whereas the pipeline developed with GreenPipe requires only 49.9 µJ. In both cases, the measurements solely include the energy consumption for executing the pipelines once. This means that, for example, the energy for generating the input data (e.g., reading hardware sensors) is not included. Traditional AutoML approaches focus solely on the functional quality of a pipeline. When a pipeline contains steps that neither positively nor negatively impact its performance, these steps are simply ignored when evaluating the pipeline's quality and thus potentially remain part of the pipeline. Although irrelevant to the functional properties, these steps still contribute to the pipeline's overall energy consumption. By contrast, GreenPipe not only considers the functional quality of the pipeline, but also its energy consumption. As a result, steps that do not contribute to the functional capabilities or steps with unnecessary complexity are eliminated from the pipeline. In the example above, the legacy AutoML approach generates a pipeline that consists of 16 steps that also include relatively complex steps with regards to the energy consumption. In contrast, the pipeline developed with GreenPipe consists of only 8 steps and in average less complex steps, which is the reason for the lower energy consumption. Thus, GreenPipe reduces the energy footprint for the data processing in this example by over 90 % while maintaining identical functionality. The key component of GreenPipe is the extension of existing AutoML approaches with an energy model to guide the automatic generation of energy-efficient pipelines. In particular, this paper makes the following contributions:

- The concept and design of GreenPipe, an approach that creates energy-efficient data-processing pipelines tailored for embedded systems.
- An implementation of GreenPipe exploring two energy models with different flexibility and portability trade-offs.
- An in-depth analysis of GreenPipe based on energy measurements, as well as an evaluation of generated pipelines and a real-world application.

The rest of this paper is structured as follows. In Section 2, the concept and design of GreenPipe are outlined. Section 3 presents background knowledge and related work. Details about the energy models are presented in Section 4 and Section 5. Section 6 describes our implementation and Section 7 introduces our measurement setup and presents an analysis of several components of GreenPipe. Finally, Section 8 evaluates GreenPipe's capability to create energy-efficient pipelines in microbenchmarks and real-world application code and Section 9 concludes this paper.

2 Concept and Design of GreenPipe

This section presents GreenPipe's concept and design for automatically creating energy-efficient data-processing pipelines. Figure 2a illustrates how an end user utilises GreenPipe. The user defines the (1) pipeline input(s) and output(s), and (2) a sample data set. GreenPipe's objective is then to find an energy-efficient pipeline that maps the pipeline input to the outputs according to the sample data set. The pipeline composition, that is, the number of steps, which steps, and the order of steps, is the vital criterion for both the effectiveness and efficiency of such a pipeline. We use automated machine learning (AutoML; cf. Section 3.1) as a basis to generate pipelines. AutoML is capable of automatically creating *effective* but not necessarily *(energy-)efficient* pipelines. In GreenPipe, we therefore complement traditional AutoML with an energy model to select effective *and* energy-efficient pipelines. GreenPipe is executed only on the development system (i.e., a desktop system). Due to the energy model, an execution of pipelines on the actual target hardware (i.e., the embedded system) during the development is not necessary. Only the final pipeline, that is, the result of GreenPipe, is installed on the embedded system for production. The full hardware capabilities of desktop systems make the application of GreenPipe convenient for developers and enable fast development cycles.

We measure the effectiveness of a pipeline by its accuracy and its efficiency by its energy demand. Optimising both accuracy and energy demand, instead of only accuracy, requires strategies to balance these properties. The appropriate strategy depends on the use case and there is no one-size-fits-all solution. Therefore, GreenPipe implements the following three strategies:

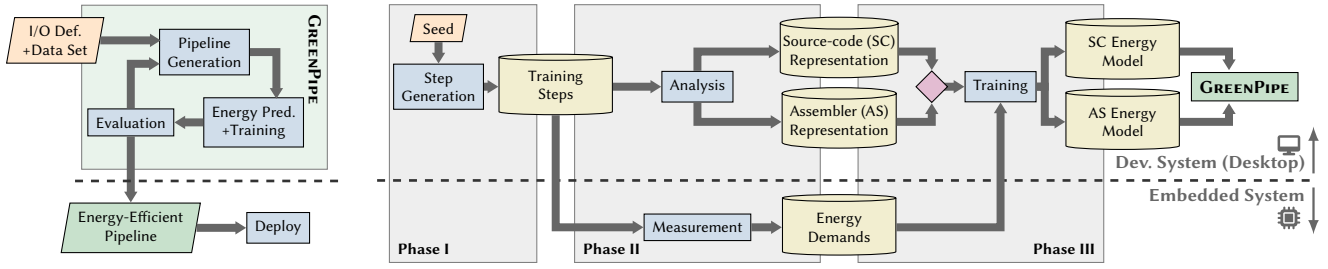
Energy Budget Considering only pipelines within a given energy budget and maximising accuracy within this budget.

Worst-Case Accuracy Considering only pipelines with a minimum accuracy and minimising energy demand for this accuracy.

Energy Efficiency Maximising a combined accuracy and energy-demand metric such as the energy-accuracy product (analogous to the commonly used energy-delay product [9]).

These automatic strategies cover the majority of use cases. However, GreenPipe also supports a manual, tailor-made selection based on the Pareto front of energy demand and accuracy (as demonstrated for a real-world application in Section 8.3).

Knowledge about the energy demand is a precondition for creating energy-efficient pipelines. However, energy measurements during day-to-day application development are often not feasible due to the unavailability of measurement equipment for developers and the excessive time and effort involved. Therefore, a key component of GreenPipe is the energy model that reliably predicts the energy demand of data-processing pipelines. An energy model has the advantage that it enables strategies such as the *Energy Budget* strategie described above. Understanding the energy budget is particularly valuable during the application design phase in embedded systems. Especially, when battery capacity or lifetime requirements are constraining the budget. Simpler approaches that only sort the pipelines by their energy efficiency, without predicting the absolute energy consumption, would prevent such strategies. We decided to implement and analyse the energy model based on two model inputs: (1) a source-code–based input (cf. Section 4) and (2) an



(a) Application of GreenPipe

(b) Implementation of GreenPipe.

Figure 2: The application and implementation of GreenPipe. For its application, an end user defines the pipeline's input and output and provides a sample data set. GreenPipe, then, generates different pipelines and evaluates their energy efficiency. Eventually, the most energy-efficient pipeline is returned. For implementing GreenPipe, the key component is the creation of an energy model capable of predicting the energy demand of pipelines.

assembler-based input (cf. Section 5). Both model inputs are independent of the pipeline input (e.g., the sensor data in Figure 3).[1] Not considering the pipeline inputs makes GreenPipe applicable during development, which we consider more important than being able to capture data dependencies of the pipeline inputs. In the remainder of this paper, the model input based on source code will be referred to as the *source code representation*, and the model input based on assembler code will be referred to as the *assembler representation*. Accordingly, the energy model based on the source-code representation is called the **source-code (SC) model** and the model based on the assembler representation is called the **assembler (AS) model**. These representations are described in more detail in Section 4 and Section 5, respectively.

The implementation of GreenPipe is designed to be portable and extensible: (1) GreenPipe can support new hardware with a minimal set of energy measurements. (2) GreenPipe easily supports new pipeline steps, ideally without additional energy measurements. The rest of this section describes the implementation of GreenPipe (illustrated in Figure 2b) and the design choices made to achieve GreenPipe's portability and extensibility.

Phase I: Training-Data Generation. The first phase consists of a pseudo-random generator that generates pipeline steps that serve as training data for the creation of the energy model. In total, we implemented support for 27 different pipeline steps. Additionally, the steps are configurable (e.g., the size of the input window). We denote a step and its specific configuration as a *step variation*. The number of potential step variations quickly becomes vast due to the combinatorial explosion of steps and configurations. Our analysis in Section 7.2 shows that not all variations are necessary and not even all steps are required in the training data. Instead, a relatively small subset of steps is sufficient for a precise energy model.

Phase II: Energy-Model Preparations. In Phase II, the steps generated in Phase I are complemented with energy measurements and an analysis. The energy measurements from the measurement campaign are used as *labels* (i.e., the expected output) for the energy-model training. The analysis determines representations of the pipelines' computational demand, which are used as *features* (i.e., the input to the energy model). To collect the energy data, we execute each step on the target hardware (i.e., the embedded system) and measure its energy demand. It is important to note that the energy measurements only need to be taken during the implementation of GreenPipe. Once GreenPipe is implemented and end users apply GreenPipe, the energy model substitutes the measurements and the target hardware is only required for installing the final pipeline for production. The computational demands of the steps are determined independently of the energy measurements. As mentioned above, we explore two different representations. Both representations do not rely on pipeline inputs.

Phase III: Energy-Model Training. After Phase II, all the necessary information is available to create the energy model. This includes the inputs to the energy model (i.e., representations) and labels for the training process (i.e., energy measurements). As we explore two representations, we execute the training, relying on supervised learning, for each representation. Section 8 analyses the representations' properties in detail. After training, the energy model is capable of predicting the energy demand of any pipeline step and, therefore, entire pipelines.

System Model

GreenPipe is designed for embedded systems and systems with constrained resources (e.g., low-power CPUs). In particular, we implemented GreenPipe for the nRF52840 system-on-chip (SoC) running an ARM Cortex-M4 CPU. However, portability to hardware with the same instruction set architecture (ISA) (e.g., different SoCs and boards) or similar hardware platforms with a different ISA (e.g., ARM Cortex-M0+) is within the scope of GreenPipe.

Currently, GreenPipe does not support machine-learning accelerators for embedded devices (e.g., Google Coral Edge TPU [6]). However, for future work, we plan to extend GreenPipe to also support acceleration hardware. Related work has shown that small accelerators can execute machine-learning workloads very efficiently under the right circumstances [12], so we see potential for GreenPipe here. We consider powerful SoCs and CPUs backed by

[1] It is important to distinguish between the input data for the pipeline and for the energy model. The input for the pipeline is, for example, the data of a sensor as shown in Figure 3 (denoted as *pipeline input*). The input to the energy model is a description of the pipeline to determine its energy demand (denoted as *[energy] model inputs*).

GPUs (e.g., high-end smartphones, desktop and server CPUs) out of scope for GreenPipe. The complexity and imposed constraints of such hardware are difficult to accommodate in (a) the energy model and (b) the automatic composition of pipelines and require different approaches.

Portability and Extensibility

Besides the functional capability of generating energy-efficient pipelines, *portability* and *extensibility* are key non-functional properties of GreenPipe. Portability denotes the support for new hardware (platforms), while extensibility refers to the support for new pipeline steps. The critical factor for both properties is the necessary effort to support a new hardware platform or step, respectively. To this end, we propose three ways to organise the energy model:
Global Energy Model: All steps in one combined training set; the model predicts the energy demand for all steps.
Single-Step Energy Model: One training set per step; training several sub-models, where one sub-model predicts the energy demand for exactly one step.
Hybrid Energy Model: A combination of both; a global energy model as default and single-step sub-models for steps where the accuracy of the global model is not sufficient. Suitable candidates for sub-models are steps with high code complexity or already known high error (cf. Section 8.1).

Using the global model, new steps can be supported out-of-the-box, as long as the corresponding representation is available. In particular, no additional energy measurements are required. This also applies to the hybrid model as long as the accuracy of the global model is sufficient for the new steps. The single-step sub-models require one training set per step. Therefore, additional measurements are necessary for new steps.

Supporting new hardware (platforms) requires additional energy measurements in all cases. However, as our analysis in Section 7.2 shows, energy measurements for a subset of steps already suffice for well-performing global energy models. This drastically reduces the effort needed to support new hardware (platforms). For single-step sub-models, new measurements for all steps are required. Our evaluation in Section 8 shows that the global energy model is already accurate enough for the intended use case of GreenPipe. However, if more accurate energy predictions are necessary, the evaluation shows that accuracy can be significantly increased using a single-step sub-model. Hence, GreenPipe is capable of supporting new hardware platforms by means of a global or hybrid model with minimal effort. Therefore, it fulfils its non-functional requirements regarding both portability and extensibility.

3 Background and Related Work

In this section, we discuss the necessary background knowledge and related work for understanding automated machine learning and energy-demand predictions. Additionally, we outline the *implicit path enumeration technique* (IPET), which is used to implement one representation.

3.1 Automated Machine Learning (AutoML)

Machine learning (ML) has pervaded many application domains [8, 22, 34]. However, the development of ML applications often requires detailed ML knowledge and demands corresponding experts.

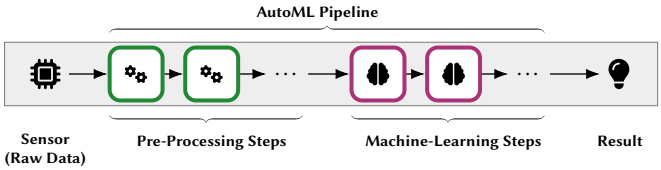


Figure 3: Exemplary AutoML pipeline consisting of several pre-processing and machine-learning steps.

Automated machine learning (AutoML) is one approach to mitigate these requirements and enables non-experts to develop ML applications [7, 11, 21]. AutoML arranges small ML steps into a machine-learning pipeline. Figure 3 exemplifies such a pipeline.

These steps can be divided into two phases: pre-processing and the actual ML steps. During the pre-processing phase, data-type conversion, data omission, and feature extraction take place. The goal is to get the input data into an easy-to-use form for the actual ML steps. The ML steps themselves can consist of any type of machine-learning algorithm. Although machine learning is often associated (or even equalised) with the use of neural networks [21, 40], this is not a must [35]. GreenPipe considers embedded devices (e.g., small ARM and RISC-V SoCs) whose processing capabilities and latency requirements typically severely limit, restrict, or even preclude the use of neural networks. For this class of devices, it is beneficial to replace complex ML methods (e.g., neural networks) with simpler ML methods, for example, statistical metrics (e.g., standard deviation, interquartile range) and simple regressors (e.g., ridge regression, random forest), which have a low resource demand. In proper combination, these simpler steps implement complex applications with drastically reduced system requirements [35]. Additionally, ML methods often provide very predictable latency behaviour during execution [12, 40], which, in turn, yields very predictable execution times for ML pipelines. As these embedded systems often require real-time properties, this is a very useful characteristic.

Composing an appropriate pipeline is usually a task for an experienced ML developer. AutoML automates this process and enables non-experts to leverage ML capabilities. Typically, AutoML starts with an initial composition, which is then iteratively changed together with the pipeline configuration, thereby improving the pipeline's accuracy. After a predefined termination condition is met (e.g., number of iterations, minimum accuracy), AutoML returns the pipeline with the highest accuracy as a result. Using this procedure, it has been shown [11] that AutoML is capable of implementing complex applications. However, we observe that not only the accuracy varies but also the energy demand of individual steps. GreenPipe exploits this property to drastically reduce the energy footprint of pipelines, as demonstrated in our evaluation (cf. Section 8). Zhao et al. [40] also consider the power demand in their AutoML implementation of neural networks on edge devices. Their application scenario, however, consists of GPU-backed systems, which provide a different class of processing capabilities. Furthermore, they utilise only a coarse-grained energy model based on GPU frequency not backed by measurements.

3.2 Energy-Demand Prediction

The resource demands of machine-learning applications are a major concern in data centres. Consequently, numerous works examine the respective resource demands (e.g., latency, memory consumption) [19, 37]. Advances in training, inference, and the respective hardware have enabled ML for mobile and embedded devices [1, 36]. As the resource demand for these devices is inherently critical due to limited computational and power resources, related work has investigated the impact of ML on mobile and embedded devices [12, 17, 23].

For the implementation of energy-efficient pipelines, however, not only an analysis of *specific* pipelines but knowledge about *arbitrary* pipelines is necessary. The use of energy models is a suitable method to analyse the energy efficiency of pipelines. Energy models, which predict the energy demand, have been applied in several related works for all hardware classes (e.g., data centres, HPC, and embedded devices [12, 13, 28]). The granularity varies from system models with limited accuracy but great generalisation [5] to very specialised instruction models [13, 28] with high accuracy.

In comparison to energy models, energy measurements usually provide more accurate results. However, in this context, energy measurements are unsuitable for several reasons: the hardware platform must be at hand, the developer must have access to and be capable of operating the measuring equipment, and the measurements are time-consuming and error-prone. In summary, measurements significantly slow down the application development and thus are unfavourable. To keep energy measurements to an absolute minimum, GreenPipe is designed to fully exploit energy models, which are trained using only a small initial measurement campaign.

To our knowledge, GreenPipe is the first work that combines energy models and AutoML to automatically create energy-efficient pipelines for embedded devices. For its energy model, GreenPipe partly uses the implicit path enumeration technique (IPET), which is briefly presented in the following section.

3.3 Implicit Path Enumeration Technique (IPET)

GreenPipe's assembler energy model (cf. Section 5) uses the implicit control flow to predict the energy demand. The implicit control flow of software describes which instruction sequences are executed and how often. In contrast, the explicit control flow also contains the order in which the sequences are executed, which is not necessary in our case. The implicit path enumeration technique (IPET) is often used to determine the worst-case (or best-case) execution time by reformulating the implicit flow as an integer linear program [20]. Similarly, it can be used to predict the worst-case energy demand [38]. The resulting instruction sequences are used to predict the energy demand of a pipeline, as described in detail in Section 5. It is important to note that although IPET is often used in best-/worst-case analyses, we aim for good prediction results in the average case.

4 Source-Code Representation

A key aspect of GreenPipe is the performance and accuracy of the energy model. The energy model relies heavily on the provided input data. Ideally, the input data precisely resembles the computational demand of the pipeline under test. In this paper, we examine two different representations to describe the computational demand of a pipeline: on the one hand, a representation based on assembler (AS) code outlined in Section 5 and on the other hand, a source-code (SC) representation explained in this section.

Table 1: Operations of the SC representation.

Operation	Description
`add{_i,_f}`	Number of additions and subtractions
`mul{_i,_f}`	Number of multiplications
`div{_i,_f}`	Number of divisions
`comp{_i,_f}`	Number of comparisons
`sqrt`	Number of calls to the squared-root function
`exp`	Number of calls to the exponential function
`log`	Number of calls to the logarithmic function
`load`	Number of memory loads
`store`	Number of memory stores

The basis for the SC representation is the source code of a pipeline step (e.g., in C/C++ or Python). Implementation-independent approaches, such as those proposed by Jensen et al. [16] or based on the big-O complexity, solely rely on the number of mathematical operations. An implementation-independent description for the arithmetic mean, for example, consists of:

```
n-1 additions      1 division
```

where n is the number of elements. However, we find that such implementation-independent approaches are too coarse-grained for accurate energy predictions. Early tests showed that such approaches were not sufficient to correctly sort the pipelines by their enery demand. In particular, the tests showed that the energy consumption for steps with the same number of mathematical operations differed by a factor of more than four. In contrast, also considering the implementation allows for the inclusion of implementation-dependent factors (e.g., loop overhead, integer versus floating-point data types, memory accesses). Of course, using only the implementation but not the compiled binary introduces some imprecision. This is due to the missing knowledge about compiler optimisations, hardware caches, and actual memory accesses. The advantage of this approach, however, is that the representation remains hardware and platform-independent and thus can be reused.

In summary, we track 13 different operations to describe the computational demand. Table 1 shows an overview and a short description of all operations. Additions and subtractions are tracked together as they are usually the same operation in hardware. Where applicable, we differentiate between integer ("`_i`" suffix) and floating-point operations ("`_f`" suffix). In addition to the 13 operations, the input to the energy model is complemented by one derived value, which is the sum of all operations. For the arithmetic mean implementation, the SC representation is as follows (omitting zero values)[2]:

```
n-1 add_i      n-1 comp_i      1    mul_f
n-1 add_f      n   load        4n-2 ops
```

[2] In this case, the final division can be mapped to a multiplication, because our implementation always uses input sizes that are a power of 2. In the general case, the `mul_f` would be a `div_f`.

This representation allows a much more fine-grained and precise energy-demand description compared to the implementation-independent approach described above.

Currently, the description for a new step is created manually. However, the creation process can be augmented by tool support (e.g., static analysis). Consequently, only modest technical knowledge is needed, as no deep understanding of the algorithm is required to count the respective operations. In particular, it is not necessary to have a deep machine-learning background, which may be required to develop new steps. This allows for the straightforward inclusion of new steps (developed by ML experts) into GreenPipe with little effort by non-experts.

One challenge is determining loop bounds, which is often difficult in the general case. In AutoML, however, almost all steps work on an input window (e.g., raw sensor data). As the window size is configured by GreenPipe, this value is known and directly or indirectly determines the upper loop bounds. Thus, determining loop bounds is often straightforward in this domain. Once created, the description can be utilised for all hardware (platforms) as it is completely hardware-independent.

Table 2: Instruction groups for the ARMv7E-M ISA.

CPU Component	Cycles	Mem.	Example
CPU	1	✗	`ADD`
	2	✗	`MLA`
	1+x	✗	`B`
	2	✓	`LDR`
	2+x	✓	`LDR(PC)`
	2 to 12	✗	`SDIV`
FPU	1	✗	`VADD.F32`
	2	✓	`VLDR.F32`
	3	✗	`VMLA.F32`
	3	✓	`VLDR.64`
	14	✗	`VDIV.F32`
DSP	—	—	`SMMLA`

5 Assembler Representation

In addition to the source-code (SC) representation presented in the previous section, we examine an assembler-based (AS) representation in this section. As the assembler code usually has a one-to-one mapping to hardware instructions, it closely resembles the operations executed in hardware. In particular, relying on the assembler code eliminates some imprecision of the higher-level SC representation (e.g., compiler optimisations). The trade-off, however, is that the AS representation is not directly transferable to other hardware. This disadvantage is mostly mitigated by the fact that both the compilation process and the assembler analysis to generate the representation are fully automated. This means that as long as a working toolchain to compile code for the other hardware is available, the AS representation can be generated automatically.

With the assembler code at hand, the total number of executed instructions for all instruction types (e.g., `ADD`, `LDR`) can be determined. Combined with an instruction-based energy model, which tracks the energy demand for executing specific instructions, the total energy demand for executing a pipeline step can be calculated. Creating an instruction-based energy model, however, requires significant training data to accurately determine the energy demand for each instruction type. Using only the total number of executed instructions is insufficient, as the energy demand may vary significantly between instruction types [13].

Therefore, we group instructions into groups with similar complexity to limit the amount of training data while retaining all information necessary for precise energy estimations. We determine the complexity of an instruction based on three factors: (1) active CPU components (e.g., ALU, FPU, DSP) (2) number of CPU cycles (3) memory accesses. If necessary, additional instruction groups can be specified based on the specific hardware platform's properties. In particular, we implemented our approach for the ARMv7E-M instruction set architecture (ISA). Table 2 shows an overview of the 12 different groups that we identified for the ARMv7E-M ISA. The `B` and `LDR(PC)` instructions are handled in separate groups as the modification of the program counter (`PC`) induces additional pipeline stalls. Similarly, `SDIV` has a separate group as the execution time may vary between 2 and 12 cycles. In addition to the respective number of executed instructions for the 12 groups, the total number of instructions is added.

This approach allows for a good trade-off between detailed information and reasonable input size. Grouping instructions into instruction groups, however, requires a correct mapping for all possible instructions, including all instruction variations (e.g., addressing modes). Furthermore, the ARMv7E-M ISA allows the specialisation of instructions by means of suffixes, resulting in an enormous space of possible instruction variations. Instead of creating and maintaining a full table with mappings for all possible instruction variations, we rely on the Levenshtein distance [18] between the mnemonic of an actual instruction and its corresponding base instruction (i.e., without addressing mode and specialisation). The Levenshtein distance considers all modifications (i.e., deleting, adding, replacing) to transform one sequence of characters into another. Additionally, the modifications can be weighted (e.g., a deletion is more expensive than an addition). As the assembler suffixes mainly add new characters, we assume the costs for *replacing* and *adding* to be ten times the costs for *deleting* characters. Although only approximate, this approach maps almost all instructions into the correct assembler group. Additionally, as this approach is independent of the hardware platform, it mitigates most of the efforts required to support a new ISA.

To determine which instructions are executed and how often, we rely on the implicit path enumeration technique (IPET). The basic principle of the IPET is summarised in Section 3.3. We chose to maximise our target function, which may lead to a tendency for overprediction of the energy demand. Similar to the SC representation, the (upper) loop bound must be provided for the IPET. Because individual steps in pipelines usually have either a fixed number of iterations or work on fixed window sizes (known at compile time), this requires only minimal effort.

6 Energy-Model Implementation

This section describes the implementation of the energy model based on the source-code (cf. Section 4) or assembler representation (cf. Section 5). Section 6.1 summarises the supported steps and Section 6.2 describes the implementation of the energy model.

6.1 GreenPipe Steps

Typically, an AutoML pipeline consists of many different steps. In total, we added support for 27 different steps to GreenPipe. Examples of steps include the calculation of absolute values, the arithmetic mean, the zero-crossing rate, and the root variance frequency [39]. The AutoML and the steps are implemented in Python. In particular, we use implementations from NumPy [10] and SciPy [33] for the steps. The AutoML approach and pipeline training are implemented using the Scikit-learn framework [26]. The execution of Python on embedded devices is either impossible or at least undesirable due to the energy and execution overhead of Python. Therefore, the *trained* pipelines are transpiled from Python to C code. For compiling C code, we utilise the ARM GNU Embedded Toolchain (`gcc-arm-none-eabi-9-2019-q4-major`)[3]. Thus, we can use the extensive ML toolset provided by the Python ecosystem without suffering the respective overheads during execution on the embedded devices.

6.2 Energy-Model Training

The prediction quality of the energy model plays a key role in our GreenPipe approach. For the training process, we test and evaluate three different regressors: decision trees, random-forest regressors, and ridge regressors. We use the Scikit-learn framework for the regressor implementations [26]. For all regressors, we use the random-search strategy [3] to optimise their hyperparameters and *leave-one-group-out* for cross-validation.

A prerequisite for successful training is the selection of a suitable loss function. We considered three typical loss functions: mean absolute error (MAE), mean squared error (MSE), and mean absolute percentage error (MAPE). One important selection criterion for the loss function is that it needs to work well for the complete range of expected values. For GreenPipe, this relates to pipelines that have very different energy demands (i.e., very simple or very complex pipelines). In this case, due to the quadratic nature of the MSE, pipelines with higher energy demand are considered more important during training, causing the energy model to inadvertently focus on complex pipelines. Similarly, the *relative error* of pipelines with low energy demand is higher, causing the MAPE to inadvertently focus on simple pipelines. For these reasons, we use the MAE between predicted and measured energy demand as the loss function.

6.3 Energy-Model Input

We explored two different inputs (i.e., representations) for the energy model based on either the assembler code or the source code of a pipeline. Although both representations are able to capture the computational demand of a pipeline and consequently lead to precise energy predictions (cf. Section 8.1 and Section 8.2), they differ in their applicability. The choice of representation depends on the user scenario. The SC model operates at a higher level and can be used in scenarios where a toolchain for the examined hardware platform is not yet available. However, it requires a manually created description based on the source code of a pipeline. In contrast, the AS model operates on the assembler code and therefore requires a functioning toolchain but can determine its inputs automatically.

[3] Flags: `-O2`, `-fsingle-precision-constant`, `-mfloat-abi=hard`

7 Energy-Model Analysis

In this section, the hardware and measurement setup are presented and in-depth properties of the global energy model are analysed to answer the following questions: (1) How many steps are necessary to build a global energy model? (cf. Section 7.2) (2) Which steps are especially important for training? (cf. Section 7.3) (3) Are the selected representations suitable for energy predictions? (cf. Section 7.4). An evaluation of the prediction quality for individual steps and whole pipelines for the global and single-step energy model is presented in Section 8. Furthermore, Section 8 presents an end-to-end evaluation with a real-world application.

7.1 Hardware and Measurement Environment

We implemented and evaluated GreenPipe on a SparkFun MicroMod ATP carrier board [29] hosting a SparkFun MicroMod nRF52840 processor board [30], serving as our device under test (DUT). The board contains a Nordic Semiconductor nRF52840 System-on-Chip [27] with an ARM Cortex-M4 CPU [2]. The Cortex-M4 is a 32-bit CPU running at 64 MHz with an instruction cache but no data caches. For deterministic energy measurements, we disabled the instruction cache. Furthermore, it includes a single-precision hardware floating-point unit (FPU). These features, along with the low cost and wide availability, make the Cortex-M4 a popular candidate for embedded devices. Due to the carrier board, it is also possible to reuse the energy measurement setup with other processors.

The energy measurement setup is powered by a source measure unit (SMU), specifically the National Instruments PXIe-4145 [15], which also measures the energy consumption of the DUT. The SMU provides a resolution of 6 µV and 1 nA at 600 kHz. The beginning and end of measurements are indicated by a GPIO pin. The nRF52840 SoC also includes an internal temperature sensor. As the SoC's temperature influences power draw [4], we ensure a stable temperature between 26 °C and 28 °C. Temperature readings are only conducted between measurements. New executables are flashed onto the DUT via a GDB server from the host PC. To determine the repeatability of our measurements, we measured 10 iterations for each step. The average energy consumption was 22.9 µJ and the average deviation was only 0.013 %. Therefore, we consider the measurements highly deterministic, which is supported by related work on the predictability of ML methods [12].

7.2 Training-Data Size Study

For GreenPipe, portability (i.e., support of new hardware platforms) and extensibility (i.e., support of new steps) are major design goals. In both cases, the limiting factor is the number of necessary energy measurements. Therefore, this analysis examines how many steps and energy measurements are required to train a global energy model capable of predicting the energy demand for arbitrary steps.

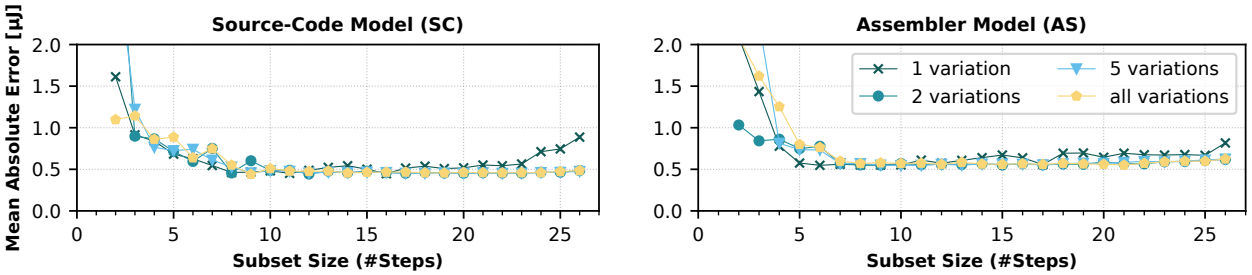


Figure 4: Mean absolute error for energy predictions by the SC and AS model. The x-axis denotes the size of the step subset. The coloured plots denote the number of variations per step used during training.

For this analysis, we conducted energy measurements for all 27 different steps with 8 to 9 variations per step. Variations of a step are created by, for example, varying the pipeline-input size (e.g., the window size of considered sensor readings). In total, we conducted 236 energy measurements.

Subsequently, we tested the prediction quality for a global energy model trained only with a subset of steps using a ridge regressor. We test up to a 1000 different subsets and select the model with the highest accuracy. For testing the prediction quality, all 236 measurements are used. Hence, good prediction quality is a strong indicator that the energy model is capable of generalising from the training data and accurately predicting the energy demand of arbitrary steps. Figure 4 shows on the y-axis the mean absolute error (MAE) for predicting the energy demand using either the source-code (SC) or assembler (AS) model. The x-axis denotes the subset size (i.e., number of steps) used during training. The differently coloured plots denote the number of variations used per step. The figure shows that even a limited subset of steps is sufficient to create an accurate energy model irrespective of whether the SC model or the AS model is used. The specific number of steps beyond which no further improvement is observed varies slightly between the models. For the AS model it is 8 steps and for the SC model it is 11 steps. Thus, both models need significantly fewer than 27 steps although the AS model converges slightly faster. In both cases, it is helpful to include more than one variation of a step to stabilise the prediction results. However, we find that only 2 variations are needed and more variations do not improve the prediction quality.

In conclusion, our analysis shows that it is *not* necessary to provide energy measurements for all steps when using GreenPipe. Even with measurements for 8 to 11 out of 27 steps and 2 variations per step, an accurate energy model can be trained. This reduces the necessary measurements by 214 (91 %) to 220 (93 %) compared to the full set for the global energy model.

7.3 Training-Data Sensitivity Study

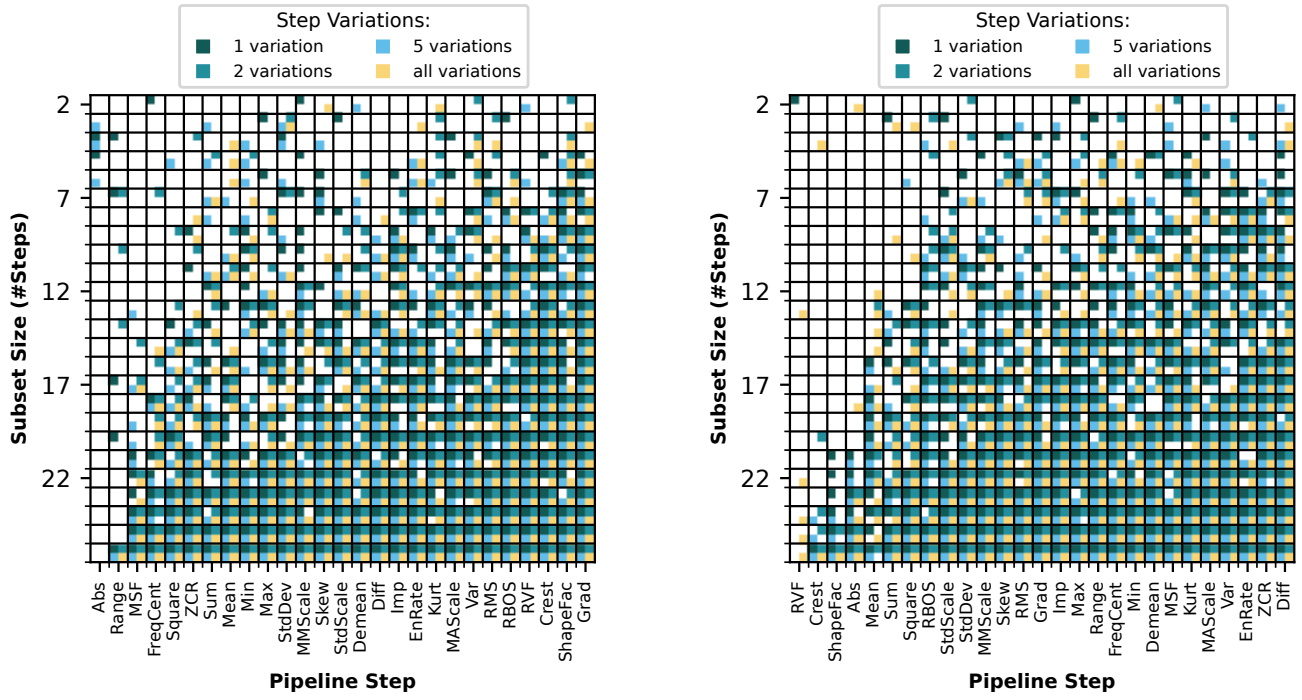
Based on the results of the previous section, Figure 5 provides a visualisation of which steps contribute most to the training process. To determine how much a step contributes to the training process, we analysed different step subsets. First, we fix the subset size (e.g., 5 different steps). Then, we train energy models for up to 1000 different subsets. From these 1000 trained models, we choose the model with the highest accuracy (tested with all steps) and visualise the steps of the chosen subset in Figure 5. Subsequently, we repeat the process with the subset size incremented by one (e.g., 6 instead of 5 steps). This was done for subset sizes from 2 to 26. Additionally, we tested different numbers of variations per step, that is, 1, 2, 5, and all step variations.

The x-axis in Figure 5 shows the steps (sorted from least frequently to most frequently used) and the y-axis the subset size. When a step is part of the most accurate pipeline for this subset size, the respective box is coloured. For example, for the SC model, a subset of 2 steps (see y-axis) and 2 variations per step (see light-blue boxes) include the *Variance* (Var) and *Shape Factor* (ShapeFac) steps as part of the most accurate pipeline. With this approach, it is possible to identify the steps that contribute most to the training and hence should be included in the training data as a priority. In both energy models, some steps are more frequently part of the most accurate pipeline. For example, the *Energy Rate* (EnRate), *Variance* (Var), and *Max/Abs Scale* (MAScale) steps are used relatively often. These steps tend to be more complex (e.g., compared to calculating the *absolute value* in the abs step). Such steps are good candidates for a comprehensive yet small training set. However, there are several examples, where it depends on the model whether a step is often part of the most accurate pipeline. For example, the *Shape Factor* (ShapeFac) and *Crest Factor* (Crest) steps are frequently used for the SC model, but almost never for the AS model. In such cases, our analysis identifies the most beneficial steps for a compact training set.

7.4 Energy-Model Feature Correlation Study

This section analyses the input features of the energy model. Unsuitable input features have no positive effect or even impair the model's quality. Hence, identifying and removing such inputs can improve the model quality and reduce the training overhead. One method to identify such features is the use of correlation coefficients. Inputs that are not correlated to the output can be removed, while highly correlated inputs should be retained.

Figure 6 shows the Spearman correlation coefficients for the input features and the energy demand. A value of 1 (or −1) denotes a perfect positive (or negative) correlation, whereas no correlation is indicated by a value of 0. In both models, the total number of operations and instructions, respectively, show a very high correlation of 0.96 and 0.98 (near perfect correlation). This is why we included this derivative feature in addition to the raw values. Almost all input features in both cases show at least some correlation to the output. The correlation strength varies depending on whether an operation



(a) Source-Code Model **(b) Assembler Model**

Figure 5: Visualisation of the steps comprising the most-accurate pipeline for different subset sizes and variations. The y-axis shows the subset size (i.e., number of steps) and the x-axis the steps (ordered from least-frequently to most-frequently used).

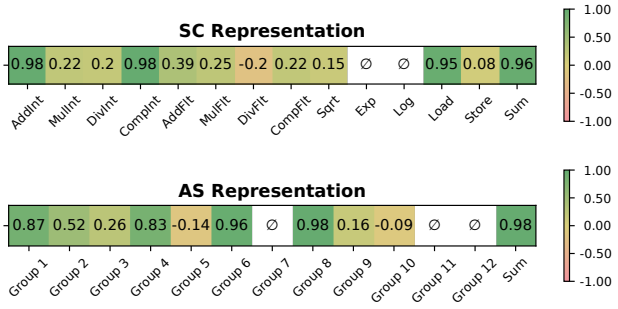


Figure 6: Spearman correlation coefficients for the input features stemming either from the SC or AS representation.

or instruction is more common in the training set (e.g., `AddInt`) or less common (e.g., `Sqrt`). Some operations and instructions are not represented in the training data (e.g., the `Log` operation or instructions from *Group 11*). For these categories, no Spearman coefficient could be calculated (indicated by the ∅ symbol in Figure 6).

As almost all features of both models show at least some correlation and due to the relatively compact feature set (i.e., 14 and 13 features), we decided to retain the inputs as they are. However, if a more compact feature set is required, this analysis provides the necessary information to merge features.

8 Evaluation

This section evaluates the ability of GreenPipe to generate energy-efficient pipelines. Therefore, the section answers the following questions: (1) What is the prediction quality for predicting the energy demand of single steps? (Section 8.1) (2) What is the prediction quality for predicting the energy demand of complete pipelines? (Section 8.2) (3) What is GreenPipe's performance for a real-world application (i.e., detecting machinery-bearing faults)? (Section 8.3).

8.1 Step Prediction Quality

This section evaluates the prediction quality of the energy models for single pipeline steps. In total, we evaluate two global models and one single-step model. As for the in-depth analysis for the global models in Section 7, all models are based either on the assembler (AS) or the source-code (SC) representation. *Global* and *single-step* refers to the way the energy model is organised (cf. Section 2). For GreenPipe, global means that one model is used to predict the energy demand for all steps. In contrast, single-step means that a separate sub-model per step is used.

The global models are based on decision trees and random-forest regressors. For the single-step sub-models, a ridge linear regressor is trained for each step. Figure 7 shows the mean absolute error (MAE) of the models in predicting the energy demand. The global models have an MAE of 0.69 µJ to 0.86 µJ for the source-code (SC) model and 0.92 µJ to 1.08 µJ for the assembler (AS) model. Both

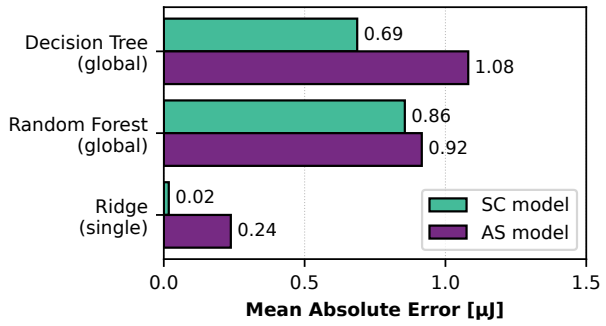


Figure 7: Evaluations of different regressors to predict the energy demand for single steps. We evaluated different regressors for global and single-step energy models.

regressors show relatively similar performance. For most steps, the AS model has a lower error than the SC model with the exception of three steps that are mainly responsible for the higher error of the AS model (see detailed per-step analysis below). For the SC model, the decision-tree model shows slightly better performance, while for the AS model, the random-forest model performs better. In summary, we consider decision trees and random-forest regressors suitable for predicting the energy demand for steps.

However, if energy-demand data for all or at least some critical steps is available, the sub-models trained for each step separately can significantly reduce the prediction error. In particular, the single-step sub-model based on the ridge regressor has an MAE of 0.02 µJ to 0.24 µJ. Hence, using single-step sub-models can increase the accuracy at the cost of significantly increased measurement efforts. The two representations for the computational demand show relatively similar performance. Although the AS representation shows a slightly higher error, both representations are capable of comprehensively describing steps. However, the prediction performance varies across different steps.

Figure 8 shows the absolute error of the global energy model based on the random-forest regressor broken down by each step, individually. The green box plots denote predictions made based on the SC representation, and the purple box plots represent the AS representation. The figure demonstrates that the better representation depends on the step under consideration. In most cases, the AS representation performs better than the SC representation, with some noticeable exceptions (i.e., ZCR, ShapeFac, Imp). Refining the AS representation to better capture these steps is promising for future work. However, the better representation ultimately depends on the use case, whether the manual yet platform-independent approach (SC) or the automatic approach (AS) is better suited.

8.2 Pipeline Prediction Quality

This section evaluates the prediction quality for complete pipelines. To this end, we generated and analysed the prediction quality of 1000 randomly generated pipelines. In the next Section 8.3, we complement this evaluation with the application of GreenPipe on a real-world scenario.

Table 3: Mean absolute error for predicting the energy demand for complete pipelines.

Repr.	Scope	MAE	Impl.
SC	global	4.0 µJ	Decision Tree
	single step	0.1 µJ	Ridge
AS	global	2.2 µJ	Random Forest
	single step	0.9 µJ	Ridge
Mean Pipeline Energy Demand: 98.7 µJ			

Each of the 1000 pipelines consists of 5 to 15 randomly selected steps and one pre-processing step. The window sizes for the pipeline input (e.g., sensor data processed by the pipeline) are always a power of 2 and range between 32 to 4096. Hence, these generated pipelines resemble typical pipelines as presented in Figure 3. Table 3 shows the mean absolute error (MAE) in predicting the pipelines' energy demand. For both representations, the MAE is remarkably low: 4.0 µJ (SC) and 2.2 µJ (AS), especially compared to the pipelines' mean energy demand of 98.7 µJ. Again, this error can be significantly reduced if a single-step sub-model is used. In that case, the MAE drops to 0.1 µJ (SC) and 0.9 µJ (AS). One interesting finding is that the prediction errors of the steps do not necessarily add up but, due to over- and underpredictions, partially cancel each other out. This explains the still relatively low MAE in predicting the energy demand of whole pipelines.

8.3 Practical Real-World Evaluation

Last but not least, we present a case study in which we apply GreenPipe to a practical real-world problem (cf. Figure 2a). We use the Case Western Reserve University (CWRU) data set [25, 31][4]. The purpose of the application is to identify and categorise machinery-bearing faults by analysing acceleration time-series data collected from an engine test stand. Therefore, GreenPipe automatically generates random pipelines utilising traditional AutoML. The pipelines use the acceleration data as input and produce the fault categorisation as output. The pipelines are assessed by their accuracy to detect bearing faults and by their energy consumption. For assessing the energy consumption, we use the global energy model based on the AS representation and a ridge regressor.

Figure 9 visualises GreenPipe's results. The plot displays the achieved application accuracy (i.e., correct fault classifications) along the y-axis and the predicted energy demand along the x-axis. The pipelines constituting the Pareto front (i.e., pipelines with the best accuracy for a given energy demand) are shown in purple. All other pipelines are shown as small light-grey dots. In Section 2, we proposed three different strategies to trade off energy demand and accuracy. For this scenario, the resulting pipeline for each of these strategies is as follows:

Energy Budget (50 µJ): ⟨96 %, 49.9 µJ⟩
Worst-Case Accuracy (90 %): ⟨92 %, 18.8 µJ⟩
Energy Efficiency (EAP)[5]: ⟨82 %, 4.7 µJ⟩

[4] Specifically, the data set with 12 kHz sampling rate [32]
[5] Energy-Accuracy Product (EAP) = $(100\% - accuracy) * energy$

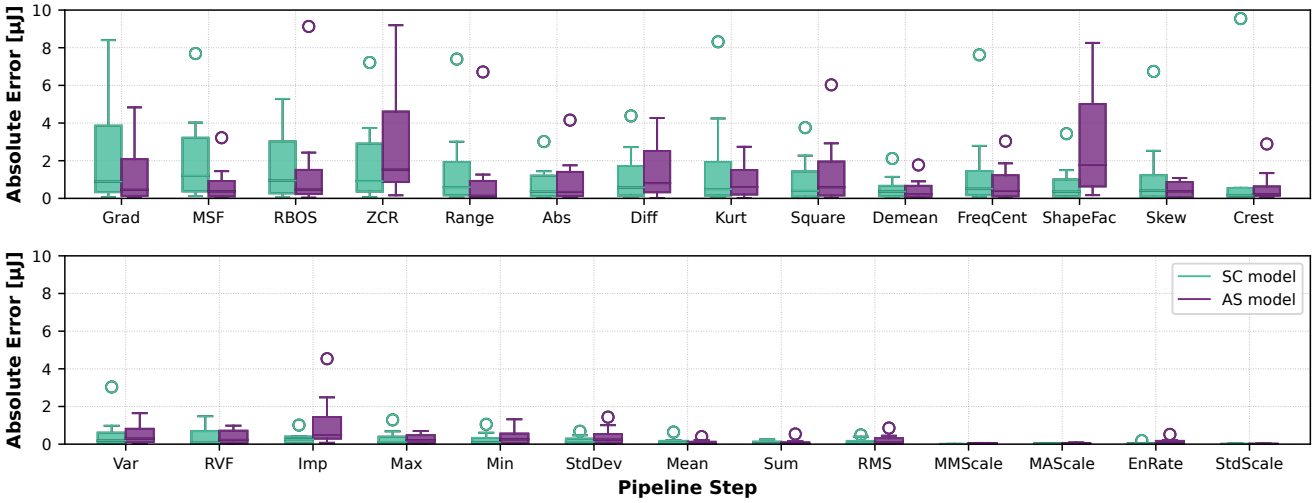


Figure 8: Absolute errors for energy predictions by the global energy model based on a random forest regressor for individual steps. The predictions are either based on the SC (green) or AS representation (purple).

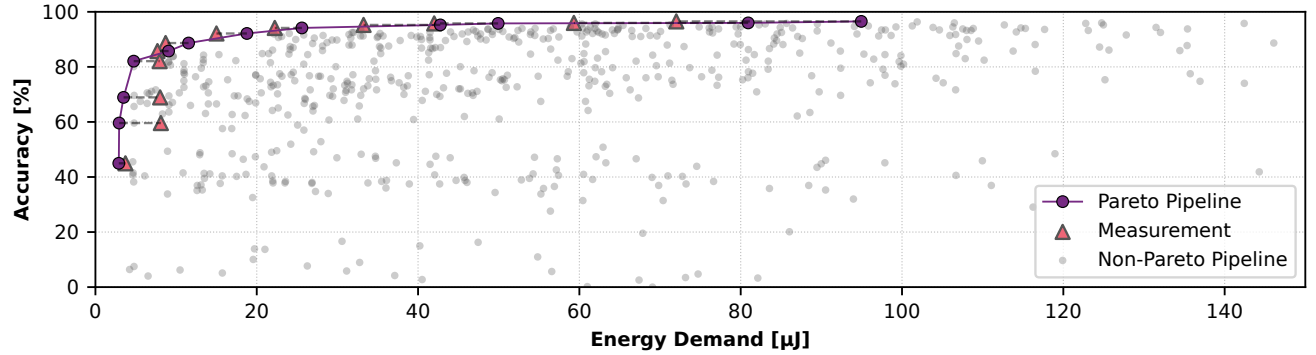


Figure 9: GreenPipe pipelines for the CWRU data set (cut off at 150 µJ). Purple points constitute the Pareto front, red triangles energy measurements, all other pipelines are shown in small light-grey points.

These results and the Pareto front clearly show that there are significant differences in energy demand for pipelines with comparable accuracy. An accuracy as high as 82 % can be achieved with only 4.7 µJ. For 92 % accuracy, only 18.8 µJ are required. To exploit the remaining accuracy headroom of 5 percentage points (i.e., for the most accurate pipeline with an accuracy of 97 %), an additional 76.1 µJ are needed.

It is important to note that Figure 9 is cut off at 150 µJ for visibility reasons. However, the energy demand of *good* pipelines (i.e., >95 % accuracy) ranges from 42 µJ to 1601.1 µJ. In energy-unaware legacy scenarios, the pipeline with the highest accuracy would be selected. By coincidence, in this example, this is a pipeline with relatively low energy demand (94.9 µJ). In general, however, this could have been a pipeline with up to 1600 µJ. Using GreenPipe, developers get explicit control over both the accuracy and the energy demand of their application.

We also measured all pipelines on the Pareto front using our energy-measurement setup. The measurements are shown in Figure 9 as red triangles. The MAE between predicted and measured energy demand for the Pareto front is 7.3 µJ. We identify two main contributors to this error: (1) imprecision due to varying pipeline inputs and (2) inter-step overhead. Currently, GreenPipe does not model the inter-step overhead, but we plan to incorporate this in future work. We argue that the imprecision due to varying pipeline inputs should *not* be incorporated into GreenPipe (as discussed in Section 2). However, we consider an MAE of 7.3 µJ acceptable to avoid energy demands of up to 1600 µJ.

9 Conclusion

In this paper, we presented GreenPipe, an approach to automatically generate energy-efficient data-processing pipelines for resource-constrained systems. Our GreenPipe implementation explores two pipeline representations, one at the source-code level and one at the assembler level. As shown in the evaluation, both representations precisely predict the energy demand of pipelines on an ARM Cortex-M4 system with an error as low as 4.0 µJ and 2.2 µJ, respectively. With these predictions, we can reduce the energy footprint

of pipelines by up to 90 % compared to energy-unaware approaches without sacrificing accuracy. We demonstrated the practicality of GreenPipe by implementing a real-world application that uses energy-efficient data-processing pipelines to identify machinery-bearing faults. Thus, GreenPipe enables the development of energy-efficient data-processing pipelines in resource-constrained systems.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project numbers 465958100 ("NEON"); 502228341 ("Memento"); and 539710462 ("DOSS") and supported by the Bavarian Ministry of Economic Affairs, Regional Development and Energy through the Center for Analytics-Data-Applications (ADA-Center) within the framework of "BAYERN DIGITAL II" (20-3410-2-9-8). We thank the anonymous shepherd and reviewers for their insightful comments, suggestions, and constructive feedback, which greatly helped us to improve this work.

References

[1] Sergei Alyamkin, Matthew Ardi, Alexander Berg, Achille Brighton, Bo Chen, Yiran Chen, et al. Low-power computer vision: Status, challenges, opportunities. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):411–421, 2019.

[2] ARM. Cortex-M4 technical reference manual (revision r0p0), 2010. https://documentation-service.arm.com/static/5f19da2a20b7cf4bc524d99a; Accessed October 4th, 2024.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012.

[4] Jacob Borgeson, Stefan Schauer, and Horst Diewald. Benchmarking MCU power consumption for ultra-low-power applications (White Paper). *Texas Instruments*, pages 1–8, 2012. https://www.ti.com/lit/wp/slay023/slay023.pdf; Accessed October 4th, 2024.

[5] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. Dynamic power management using adaptive learning tree. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design (ICCAD'99)*, pages 274–279, 1999.

[6] Google Coral. Edge tensor processing unit (edgetpu), 2024. https://coral.ai/products/; Accessed October 4th, 2024.

[7] Tijl De Bie, Luc De Raedt, José Hernández-Orallo, Holger H. Hoos, Padhraic Smyth, and Christopher K. I. Williams. Automating data science. *Communications of the ACM*, 65(3):76–87, 2022.

[8] F. John Dian, Reza Vahidnia, and Alireza Rahmati. Wearables and the Internet of Things (IoT), applications, opportunities, and challenges: A survey. *IEEE Access*, 8:69200–69211, 2020.

[9] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, 1996.

[10] Charles R. Harris, K. Jarrod Millman, Stéfan J. Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[11] Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:1–27, 2021.

[12] Benedict Herzog, Stefan Reif, Judith Hemp, Timo Hönig, and Wolfgang Schröder-Preikschat. Resource-demand estimation for edge tensor processing units. *ACM Transactions on Embedded Computing Systems (Special Issue on Accelerating AI on the Edge)*, 21(5):1–24, 2022.

[13] Timo Hönig, Benedict Herzog, and Wolfgang Schröder-Preikschat. Energy-demand estimation of embedded devices using deep artificial neural networks. In *Proceedings of the 34th Symposium on Applied Computing (SAC'19)*, pages 617–624. ACM, 2019.

[14] Transforma Insights. Current IoT forecast highlights, 2024. https://transformainsights.com/research/forecast/highlights; Accessed October 4th, 2024.

[15] National Instruments. PXIe-4145 specifications, 2024. https://www.ni.com/docs/de-DE/bundle/pxie-4145-specs/page/specs.html; Accessed October 4th, 2024.

[16] Ulf Jensen, Patrick Kugler, Matthias Ring, and Bjoern M. Eskofier. Approaching the accuracy–cost conflict in embedded classification system design. *Pattern Analysis and Applications*, 19:839–855, 2016.

[17] Luke Kljucaric, Alex Johnson, and Alan George. Architectural analysis of deep learning on edge accelerators. In *Proceedings of the 24th High Performance Extreme Computing Conference (HPEC'20)*, pages 1–7. IEEE, 2020.

[18] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[19] Da Li, Xinbo Chen, Michela Becchi, and Ziliang Zong. Evaluating the energy efficiency of deep convolutional neural networks on CPUs and GPUs. In *Proceedings of the 2016 International Conferences on Sustainable Computing and Communications (SustainCom'16)*, pages 477–484. IEEE, 2016.

[20] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd Design Automation Conference (DAC'95)*, pages 456–461. ACM, 1995.

[21] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*, pages 19–34. Springer, 2018.

[22] Christoffer Löffler, Christian Nickel, Christopher Sobel, Daniel Dzibela, Jonathan Braat, Benjamin Gruhler, Philipp Woller, Nicolas Witt, and Christopher Mutschler. Automated quality assurance for hand-held tools via embedded classification and AutoML. In *Proceedings of the Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track*, pages 532–535. Springer, 2020.

[23] Zongqing Lu, Swati Rallapalli, Kevin Chan, and Thomas La Porta. Modeling the resource requirements of convolutional neural networks on mobile devices. In *Proceedings of the 25th International Conference on Multimedia (MM'17)*, pages 1663–1671. ACM, 2017.

[24] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. *ACM Computing Surveys*, 54(8), 2021.

[25] Dhiraj Neupane and Jongwon Seok. Bearing fault detection and diagnosis using Case Western Reserve University dataset with deep learning approaches: A review. *IEEE Access*, 8:93155–93178, 2020.

[26] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[27] Nordic Semiconductor. nRF52840 System-on-Chip product specification (v1.1), 2019. https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.1.pdf; Accessed October 4th, 2024.

[28] Volkmar Sieh, Robert Burlacu, Timo Hönig, Heiko Janker, Phillip Raffeck, Peter Wägemann, and Wolfgang Schröder-Preikschat. An end-to-end toolchain: From automated cost modeling to static WCET and WCEC analysis. In *Proceedings of the 20th International Symposium on Real-Time Distributed Computing (ISORC'17)*, pages 158–167. IEEE, 2017.

[29] SparkFun. MicroMod ATP carrier board (DEV-16885). https://github.com/sparkfun/MicroMod_ATP_Carrier_Board; Accessed October 4th, 2024.

[30] SparkFun. MicroMod nRF52840 processor. https://github.com/sparkfun/MicroMod_Processor_Board-nRF52840; Accessed October 4th, 2024.

[31] Case Western Reserve University. Bearing data center, 2023. https://engineering.case.edu/bearingdatacenter; Accessed October 4th, 2024.

[32] Case Western Reserve University. Bearing data center (12 kHz), 2023. https://engineering.case.edu/bearingdatacenter/12k-drive-end-bearing-fault-data; Accessed October 4th, 2024.

[33] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. SciPy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, 2020.

[34] Satyendra K. Vishwakarma, Prashant Upadhyaya, Babita Kumari, and Arun Kumar Mishra. Smart energy efficient home automation system using IoT. In *4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU)*, pages 1–4, 2019.

[35] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. FLAML: A fast and lightweight AutoML library. In *Proceedings of Machine Learning and Systems 3 (MLSys'21)*, pages 434–447, 2021.

[36] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, et al. Machine learning at Facebook: Understanding inference at the edge. In *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA'19)*, pages 331–344. IEEE, 2019.

[37] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU performance and power estimation using machine learning. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA'15)*, pages 564–576. IEEE, 2015.

[38] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS'18)*, pages 1–25, 2018.

[39] Zhanguo Xia, Shixiong Xia, Ling Wan, and Shiyu Cai. Spectral regression based fault feature extraction for bearing accelerometer sensor signals. *Sensors*, 12(10):13694–13719, 2012.

[40] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. EdgeML: An AutoML framework for real-time deep learning on the edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation (IoTDI'21)*, pages 133–144. ACM, 2021.