ConvReflex: Efficient Ultra-Low-Power CNN Inference via Clamping Prediction

Shiming Li
Uppsala University
Sweden
shiming.li@it.uu.se

Luca Mottola
Politecnico di Milano, RISE,
and Uppsala University
Italy and Sweden
luca.mottola@polimi.it

Yuan Yao
Uppsala University
Sweden
yuan.yao@it.uu.se

Stefanos Kaxiras
Uppsala University
Sweden
stefanos.kaxiras@it.uu.se

Abstract

We present ConvReflex, a compile-time toolchain enabling shortcuts based on value prediction in convolutional neural networks (CNNs) on ultra-low-power devices. Convolution kernels may produce extreme values that are out of the boundaries allowed by the output neuron, and the result of the convolution operation is necessarily clamped at the boundaries. Execution time is thus wasted, because an exact computation result is unnecessary, as long as the value fed to the output neuron is extreme enough to be eventually clamped. We build shortcuts to skip such ineffectual computations in the serially executed convolution kernels on ultra-low-power devices, where the CNN workloads exhibit frequent value clamping. The toolchain we design and implement, named ConvReflex, finds shortcuts in convolution kernels via compile-time profiling, allowing the deployed convolution kernel code to jump to termination when the intermediate results satisfies specific requirements that yield a performance gain in exchange of a controlled degradation in accuracy. Our results obtained on a Cortex-M0+ core show that ConvReflex enables up to 21% time saving, and thus a corresponding energy benefit, when allowing less than 1% accuracy loss, or up to 27% time gain when the accuracy loss budget is 3%.

CCS Concepts

• **Computer systems organization** → **Embedded software**.

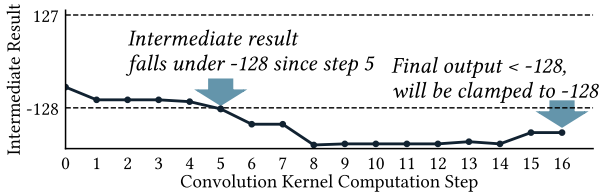
Keywords

Ultra-Low-Power MCUs, Convolutional Neural Network (CNN) Inference, Energy Efficiency

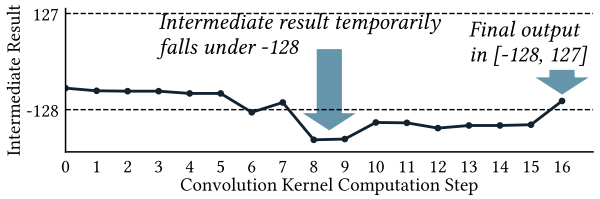
ACM Reference Format:
Shiming Li, Luca Mottola, Yuan Yao, and Stefanos Kaxiras. 2026. ConvReflex: Efficient Ultra-Low-Power CNN Inference via Clamping Prediction. In *ACM/IEEE International Conference on Embedded Artificial Intelligence and Sensing Systems (SenSys '26), May 11–14, 2026, Saint Malo, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3774906.3802781

SenSys '26, Saint Malo, France

ACM ISBN 979-8-4007-2309-4/2026/05
https://doi.org/10.1145/3774906.3802781



(a) *A convolution kernel may execute computations with no effect on the final output. In this example, step 5 to 16 can be omitted.*



(b) *However, a convolution kernel's intermediate result can temporarily fall under 0, while eventually producing a value that is not clamped.*

Figure 1: Computations in a convolution operation can be useless (a), yet this behavior can be tricky to detect (b).

1 Introduction

The appeal of deploying neural inference on resource-constrained edge devices is multifold. First, energy efficiency across the entire application operation: Microcontroller Units (MCUs) operating in the tens of MHz with milliwatt power usage enable battery-powered [32, 42] or even energy-harvesting deployments [1, 5, 12, 14]. Second, low-latency responses: running inference locally avoids the unpredictability of network latency [27], a crucial factor for sensor-based systems that require immediate reactions. Finally, preserving privacy: by performing basic machine learning tasks locally on the MCU, the device avoids transmitting raw sensor data, which may contain sensitive information. Instead, only high-level, task-relevant outputs are sent, reducing both communication bandwidth and potential privacy leakage [27]. Due to the tight energy budgets, however, improving the efficiency of processing is key.

Neuron value clamping wastes computations. In convolutional neural network (CNN) workloads on ultra-low-power MCUs, execution time can be wasted on neurons producing clamped values [29].

In these workloads, after a convolution operation, the result fed to the output neuron may be clamped to the boundaries allowed by the output neuron, which is introduced by the limitation of the quantized data type or activation functions [7, 29, 44] such as ReLU [2, 35] or ReLU6 [28]. Value clamping can be formalized as:

$$a_{clamped} = MIN(MAX(b_{lower}, a), b_{upper}) \quad (1)$$

where a is the accumulation result of the convolution operation, $a_{clamped}$ is the accumulation value that is eventually fed into the output neuron after trying to clamp a, b_{lower} and b_{upper} are the lower and upper boundaries of the output neuron, respectively.

When the neuron value is clamped, the exact computation result for the neuron's output value becomes unnecessary, as long as the value fed to the output neuron is clamped to the same boundary. These situation is common in CNNs on ultra-low-power MCUs, where ReLU-like activation functions are often adopted for because of resource-constraints and ease of implementation [29, 48].

In Figure 1a, we exemplify this sitation with an execution trace in a convolution kernel of a Mobilenet [19] model from the STM AI Model Zoo Suite [48]. This convolution kernel is composed of 16 Multiply-accumulate Operations (MACs), and the output neuron only accepts values between $[-128, 127]$, which is the quantized range of the layer's ReLU6 activation function. The plot illustrates the value change of the intermediate results of the 16 MACs. This convolution operation leads to a result less than -128, which will be clamped to -128 when the result is fed to the output neuron. However, the intermediate computation result between the 8th and the final step is constantly under -128 in the last 12 steps. Even if the execution of the convolution operation is terminated after the 5th step, the final output neuron value is still clamped to the lower boundary. This means that *nearly 70% of the computations in this convolution operation have no effect* on the neuron's output.

However, at run-time, it is generally impossible to know whether the final result is clamped or not. We show such an example in Figure 1b. This plot is based on the execution trace of the same convolution kernel as in Figure 1a, but comes from another convolution operation. This time, the intermediate result falls out of the boundaries between the 8th and the 15th step, but the convolution result eventually returns back to the $[-128, 127]$ interval. Existing researches attempt to tackle this problem by considering the maximum or minimum future results into consideration, in principle [29]. However, this makes overly conservative assumptions on the potential change of the intermediate result and makes the condition to determine value clamping too difficult to meet, leaving less potential for optimizations.

There exists much potential to save execution time if these meaningless operations can be omitted. An investigation into the workloads involved in our experiments show that on average 28% of the computations in convolutional layers bear no effect on the final value of the neuron, as shown in Figure 2.

ConvReflex predicts and omits unecessary computations. We design a compile-time toolchain, named ConvReflex[1], to create shortcuts in convolution kernels that spare likely unnecessary computations. Similar to conditioned reflexes in higher animals' nervous systems triggering responses upon certain stimuli, these shortcuts enable rapid reactions in convolution kernels whenever they are speculated to produce a clamped final result. This is done by omitting the remaining computations and producing an output with the intermediate result already.

[1]We open-source ConvReflex at https://github.com/shm-li/convreflex.

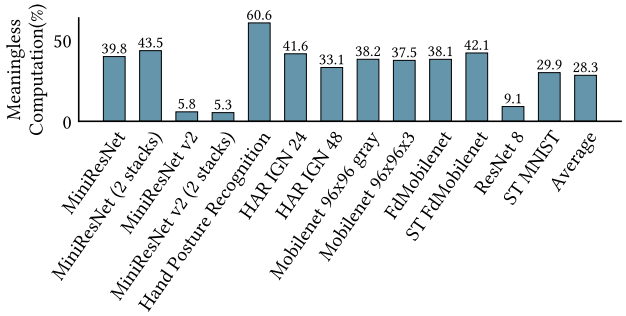


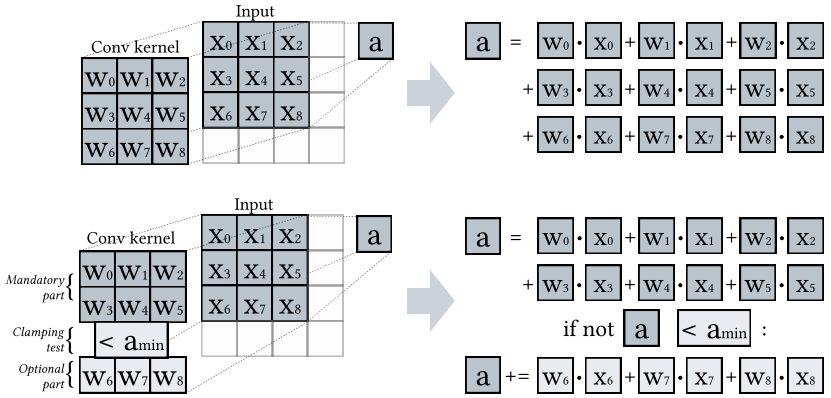
Figure 2: The percentage of computations with no effect on the output neuron values in convolutional layers in the workloads involved in our experiments. *These computations become unncessary due to value clamping. On average, 28% of the computations in convolutions have no effect on the neuron outputs.*

In ConvReflex, we adopt a statistical approach to predict whether the result of an ongoing convolution operation is likely to be clamped. Based on compile-time profiling, ConvReflex observes the relations between the intermediate result and the final result in each kernel, and finds out what intermediate result values are likely to ultimately lead to a convolution result that will be clamped when fed to the output neuron.

ConvReflex provides flexibility in navigating the tradeoff between the time saving and the error brought by the shortcuts, by allowing the user to run its pipeline under a configurable *accuracy loss budget*, similar to existing works [12, 13]. By iteratively adjusting the shortcut condition and evaluating the revised model's accuracy, the toolchain ultimately finds the best shortcut placement and configuration, which leads to the most computation omission, while ensuring the accuracy loss of the CNN model stays within the budget during evaluation.

ConvReflex provides support for generating and executing the CNN inference code for MCUs with shortcut enabled. We illustrate the execution logic of ConvReflex's in Figure 3. On ultra-low-power MCUs, a convolution operation is simply executed as a series of multiplication and addition instructions. In ConvReflex, the execution of the convolution kernel is split into a mandatory and an optional part, with a conditional clause in between. The condition here is to trigger the shortcut; the optional part is executed only if the condition is not met. Otherwise, the convolution operation terminates after only the mandatory part, and the intermediate result is believed to be sufficient to produce the same, clamped final result, as if all computations were executed.

Benefits. We evaluate ConvReflex on a Cortex-M0+ MCU across a wide variety of open-source CNN workloads [48]. We use ConvReflex to create shortcuts for CNN models and generate C code to be deployed on the device. We compare the model inference code generated by ConvReflex with shortcuts with their vanilla baselines. Two setups are used in evaluation: we evaluate models



(a) *A conventional convolution operation is executed as a sequence of multiplications and accumulations. The behavior of the convolution kernel is determined, and all computations in the sequence are always executed.*

(b) *Our approach adds a test customized for the kernel. The remaining computations are omitted if the test passes, which indicates the current intermediate result is likely to lead to the same clamped value as the complete computation.*

Figure 3: Conventional convolution and convolution with shortcuts.

with shortcuts generated with a 1% accuracy loss budget, as this level of accuracy loss is generally considered negligible and "lost in noise" [12, 37]. We also use a setup with a 3% accuracy loss budget, which is somehow perceivable, but enables additional time savings. We measure execution times and energy consumption on a different subset of the test set that is not seen for compile-time profiling.

Experiments show that with a maximum 1% accuracy loss budget, we obtain a time saving of up to 21%. This is increased to 27% when we raise the accuracy loss budget to 3%. ConvReflex negligibly influences on current draw, and hence the energy saving is nearly proportional to execution time saving, which is up to 20% and up to 26%, for 1% and 3% accuracy loss budget, respectively. The gains in execution time and energy consumption come with only 3% space overhead on average and around 1% increase in current draw. Further, we investigate the real accuracy loss in the dataset used for our real-board evaluation, and confirm that the accuracy loss seen in deployment is only up to 0.67% and 1%, for the setup with 1% and 3% accuracy loss budget in the compile-time pipeline, respectively.

The remainder of the paper is organized as follows. Section 2 provides the background and context. Section 3 elaborates on ConvReflex's design and implementation. Experimental results and analysis are presented in Section 4. Section 5 discusses design choices and limitations of our work. Finally, Section 6 ends the paper.

2 Background and Related Work

Our work touches upon several inter-related areas. In the following, we provide necessary background information while briefly surveying related work.

2.1 Low-power Inference

The key challenge lies in the severe architectural and computational constraints of ultra-low-power MCUs, which make deep learning models difficult to deploy without significant optimizations [32].

To mitigate these, model augmentation and pruning are often employed to run intermittent inference. Kang et al. [25, 26] and Yen et al. [55] append components to existing models to allow progress tracking information to be piggybacked onto output features. Without affecting accuracy, this allows the system to efficiently recover the inference process after an energy failure. iPrune [30] embeds an ad-hoc pruning strategy that produces compact models for intermittent systems, whereas RAD [22] employs block circulant matrices and structured pruning to exploit vector operation accelerators.

Network architecture search is also investigated for intermittent inference. HarvNet [24] includes two complementary techniques, one enabling architecture search that optimizes multi-exit features based on memory and energy constraints, the other returning efficient inference policies by taking into account energy constraints. iNas [33] seeks to strike a trade-off between data reuse and energy overhead of state persistence operations necessary to cross energy failures. EVE [23] uses custom network search algorithms to produce different models, enabling run-time selection based on energy constraints. Differently, Neuro-C [41] employs a custom architecture that eliminates multiply-accumulate operations for efficient inference on hardware platforms without dedicated hardware support. Unlike these works, we do not change the architecture but only the inference process, taking advantage of an inherent feature of how this process unfolds on ultra-low-power devices.

Approximation-based techniques to improve energy efficiency for intermittent systems are also investigated. Intercept [12] trades off data write errors for reduced STT-MRAM write current, achieving energy gains during inference on intermittently-powered systems. Approxify [46] develops an automated framework to apply various approximations to intermittent computing applications, while CheckMate [43] proposes a framework utilizing LLMs for context-aware code approximation. Unlike these works, ConvReflex utilizes compile-time knowledge of the neural network's behavior to open up opportunities for approximation, effectively reducing computation energy by directly cutting off computations at very low computation error cost.

2.2 Early-exiting Neural Network Inference

Works exist that apply a variety of compression techniques and design multi-exist network architectures for energy efficiency. As an example, Wu et al. [53] design a network compression algorithm working with multi-exit deep neural networks (DNNs) that selects exits based on energy predictions. In other techniques [11], the system steps out of the inference process depending on energy constraints. ConvReflex exploits the same trade-off: we accept

a limited decrease in accuracy to improve execution latency and thus energy consumption. Unlike existing works, however, runtime overhead is reduced by taking care of the heavy-lifting at compile-time.

Previous works explore skipping ineffectual computations on hardware accelerators. SnaPEA [7] proposes a technique through software-hardware co-design that terminates computations in neurons once they cannot produce a positive result, while relying on the condition that the inputs of each layer are *all positive values*, which is impractical for modern quantized models where the inter-layer normalization and re-quantization may change the sign and values of layer outputs. Song et al. [44] define ineffectual output neurons (iEONs), whose values essentially have no influence on the subsequent layers, and propose a two-stage DNN execution model that first predicts and then skips iEONs; this technique is designed for accelerators and relies on checking the higher order bits of operands for making predictions. BitSET [39] exploits similar properties in CNNs and allows the system to skip the computations leading to a clamped zero with a bit-serial accelerator. TangramFP [54] proposes a MAC unit design that skips ineffectual partial products of floating point operations. Unlike these works, ConvReflex is a software-based solution without relying on dedicated accelerators or bit-level operations on the operands.

Closest to our work is a technique [29] that examines the theoretical maximum or minimum value of a computation to determine whether clamping is applicable. This inherently assumes that all unfinished computations generate the most extreme results possible. A convolution operation can be terminated if the final result is bound to leave the range. Although this technique retains the exact output of the original process, it makes unnecessarily conservative assumptions about the future computations, losing potential performance gains only to account for the unlikely extreme results. This technique also relies on reordering the computation steps in a kernel to prioritize the steps with more impact on the result in order to expose more computations that can be omitted without introducing error. This introduces extra space overhead to store the computation order. In contrast, we do not impose any significant space overhead and allow a limited loss in the accuracy of the output to unlock much greater performance gains.

2.3 Error Resilience of CNNs

Extensive literature [3] studies hardware faults in DNN execution and, in particular, the related data errors in CNNs. The general observation is that the large information redundancy in the model and weights give CNNs high error resilience, that is, the CNN can produce the correct outcome even if some data errors corrupt the processing. This capability is widely exploited [10]; systems are modified at hardware or software level to elaborate data in a slightly inexact way, aiming at reducing resource consumption at the cost of a limited accuracy loss.

CNN targeting resource-constrained devices are normally quantized to meet memory, processing, and energy requirements. Quantization makes the networks more robust. Hoang et al. [18] demonstrate that the narrower the range that a value in the intermediate layer can assume, the more this layer is robust to errors. With 8-bit quantization intermediate values assume a smaller range compared to their non-quantized counterpart [53]. The limited range acts as a hardening mechanism [15]. Ibrahim et al. [20] also show that when injected with errors, different layers of a CNN bear different impacts on accuracy. They identify the most critical layers and harden their values, increasing the network's error resilience.

3 ConvReflex

We start with an overview of the workflow of ConvReflex. We then describe the technical details in the key steps, namely, how the shortcuts are selected, and how the selection is adjusted and updated in iterations to make full use of the accuracy loss budget. We conclude the section with a brief explanation of the implementation.

3.1 Overview

ConvReflex aims to provide CNN execution support for MCUs where shortcuts are infused into convolution kernels, which are triggered when the convolution operation is speculated to generate a clamped value. The shortcuts are carefully chosen so that they reduce as much computation as possible, while introducing no or little error to the output of the the CNN inference. This is achieved by conducting compile-time profiling on the model's behavior and finding what intermediate convolution results are more likely to result in a clamped final result.

The CNN shortcuts can be further adjusted, if they induce too much accuracy loss beyond the user's configured budget. The compile-time pipeline results in a modified CNN model with improved efficiency and minimal expected error.

We show the overview of ConvReflex's workflow in Figure 4. The compile-time pipeline consists of four key steps:

① *Profiling*: In this step, ConvReflex works on a CNN model and a subset of its test set. ConvReflex conducts profiling by feeding the inputs into the model and observing the intermediate values and final results in the convolution operations. This step generates profiled data reflecting the behavior of each convolution kernel.

② *Creating shortcuts*: based on the output of the profiling step and a parameter *conf* representing the lowest acceptable confidence of the shortcut not introducing error. ConvReflex selects and infuses shortcuts into the vanilla model. ConvReflex decides the position of the shortcuts and the conditions to trigger based on the configurable parameter *conf*. A higher *conf* creates less aggressive shortcuts where the bar of the shortcut condition is higher, so that the accuracy loss is lower, but also leaves less room for skipping computations. This step generates the shortcut-enabled CNN model.

③ *Evaluating*: ConvReflex takes in a different subset of the test set which is not seen during profiling, the model generated in the last step, and a parameter k which represents the accuracy loss budget. ConvReflex evaluates the model's accuracy using the test subset and checks whether the accuracy loss of the model with shortcuts exceeds k. In our experiments, we test two values for k: 1% and 3%.

④ *Adjusting*: this step is taken if the accuracy loss is still smaller than k. ConvReflex goes back to step ②, adjusts the parameter *conf* to generate more agressive shortcuts, which may

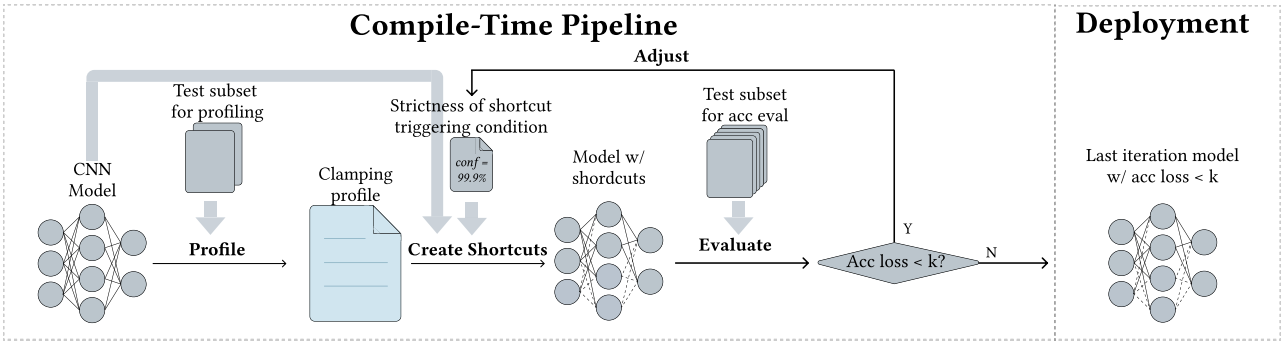


Figure 4: Overview of ConvReflex. *ConvReflex starts with ①profiling the CNN model's intermediate values and final results of neurons with different inputs fed to the network. Based on the value clamping profile, ConvReflex ②creates shortcuts in the model by inserting conditional clauses in the convolution kernels, guided with configurable parameters indicating how progressive the predictions should be made. The accuracy of the model will be ③evaluated with another subset of the test set, and if the accuracy loss is within the budget allowed by the user, the parameters controlling shortcut selection will be ④adjusted to try to produce more relaxed shortcut triggering conditions, which starts another iteration of step ②, ③ and ④. The pipeline stops when a model with accuracy loss exceeding the user's tolerance is generated, and the last iteration's model will be kept.*

skip more computations in tradeoff for increased error, and start a new iteration from step ②. Otherwise, the pipeline stops and takes the model generated in the last iteration as the final output.

Note that the data gathered in profiling are platform-agnostic, meaning that *the compile-time pipeline can be run on any device*, without needing the target MCU where the models are to be deployed. Eventually, ConvReflex generates a modified CNN model that contains shortcuts in convolution kernels where applicable. The model generated by the compile-time pipeline is ready to be deployed on the real MCU devices. Next, we explain these steps in further detail.

3.2 Compile-time Profiling

To omit the useless computations from a convolution operation, our goal is to predict whether the convolution will result in a clamped value before all computations complete. However, as shown in the counter example in Figure 1b, it might happen that even if the intermediate result has stayed out of the output neuron's allowed boundaries, its final result is still between $[-128, 127]$, which should be retained instead of clamped. Without the knowledge of future computation results, determining whether the intermediate result will keep staying out of the boundary of the neuron until the computation finishes is difficult.

Goal. Our solution stems from the simple insight that more extreme intermediate accumulation results are more likely to lead to clamped final results. To this end, ConvReflex examines the behavior of convolution kernels on sample inputs at each computation step, investigates the relation between the intermediate values and the final results, and tries to find for each convolution kernel *how extreme the intermediate result should be to guarantee it leads to a clamped final result.*

For each computation step in a convolution kernel, the profiling process gathers two kinds of data:

(1) How many times each possible value appears;

(2) How many times a value leads to a clamped final result.

Result of profiling. Figure 5 demonstrates a real example from the same convolution kernel as in Figure 1. In this example, we visualize the process of profiling for one computation step in the kernel, which is step 8. Note that *we only exhibit the case for profiling lower-bound clamping*, that is, the final result being clamped to the neuron's lower boundary b_{lower}.

For each input, each time the convolution kernel moves to a new part of the layer's input and executes a convolution operation, the value change of the convolution output is recorded by ConvReflex, as exemplified by the first three line charts in Figure 5. For step 8, ConvReflex collects profiles of the frequency of each unique value of the intermediate result at this step, as well as whether the final result is a clamped value. These are marked with light blue arrows and dark blue arrows in the line charts, respectively. For example, in the first line chart, ConvReflex observes that the intermediate value at step 8 is −230, and the final result is clamped. As for the second line chart, ConvReflex observes that the intermediate value at step 8 is −208, while the final result is not clamped.

The profiled data would construct the bar chart at the bottom of Figure 5. The light blue bars stand for the occurrence of intermediate results at step 8, and the dark blue bars stand for the occurrence of clamped final results associated with the intermediate values. The length of the bars reflects the normalized frequency. Formally, a light blue bar at value = n can be represented as:

$$freq(a_8 = n) \tag{2}$$

while a dark blue bar at value = n can be represented as:

$$freq(a_8 = n \cap a_{16} < -128) \tag{3}$$

The profiled data provide directions to answering the question about what intermediate values at step 8 are more likely to lead to a clamped final result. For example, the figure shows that if the intermediate result is smaller than -250 at step 8, the final convolution result is almost always clamped to −128.

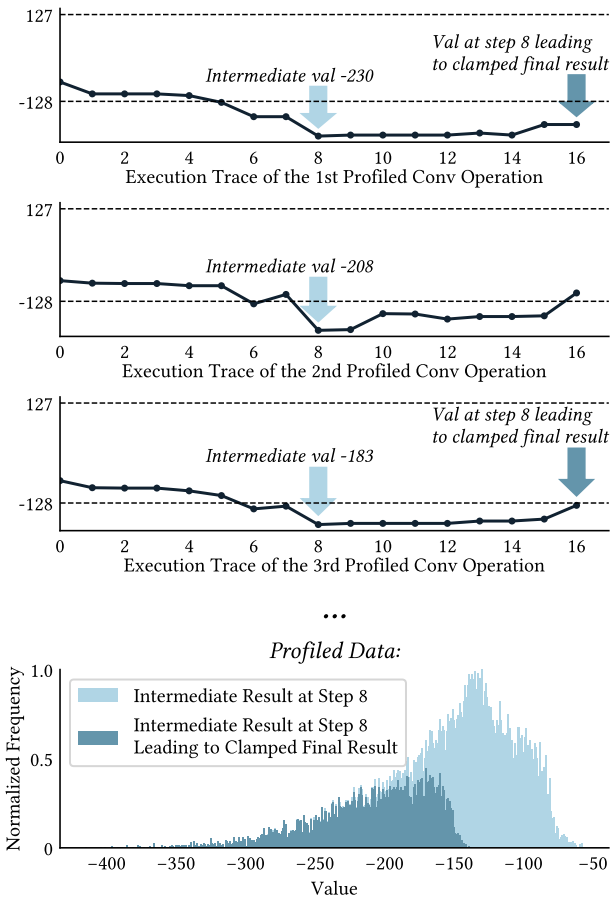


Figure 5: Visualization of the profiling process and the profiled data for a certain computation step in a convolution kernel. *The first three line charts are example traces of the intermediate result of the convolution kernel's operation on different parts of the layer input or network input. The intermediate results at step 8 and whether they lead to a clamped final result will be recorded during the profiling process. The last plot shows the profiled data, which is the normalized frequency of the intermediate result's value at computation step 8 of the example in Figure 1, and the value leading to a clamped final result. The profiled data can indicate, for example, that if the intermediate result at step 8 is smaller than -250, then the convolution operation almost always leads to a clamped final result.*

3.3 Selecting and Creating Shortcuts

To construct a shortcut in a convolution kernel, two pieces of information are required: (i) what is the condition to trigger the shortcut, and (ii) at which computation step, i.e. the position, the shortcut should be made. For simplicity, in this subsection, we still only elaborate on the lower-bound clamping case, which is when the final result is clamped to b_{lower}, the output neuron's lower boundary.

Shortcut triggering condition. We start with deciding the condition to trigger a shortcut at a certain computation step i. With the idea that the smaller the intermediate result is, the higher chance it has to lead to a final clamped result, ConvReflex expects shortcut triggering conditions in the format of $a_i < a_{min}$, where a_i is the value of the intermediate result at step i, and a_{min} is the unknown triggering value. The key is to find the suitable a_{min}.

We use the profiled data in the last step to guide the decision of a_{min}. The probability of lower-bound value clamping when the intermediate result at a given step i is smaller than a certain value a_{min} can be represented as the following conditional probability which can be calculated with information from the profiled data:

$$P(a_m < b_{lower} | a_i < a_{min}) = \frac{\sum_{j=-\infty}^{a_{min}} freq(a_i = j \cap a_m < b_{lower})}{\sum_{j=-\infty}^{a_{min}} freq(a_i = j)} \tag{4}$$

where m is the total number of computation steps.

From the visualization of the profiled data in the last bar plot in Figure 5, it can be observed that the greater a_{min}, the lower the probability $P(a_m < b_{lower} | a_i < a_{min})$, meaning a shortcut triggering condition with this a_{min} value has a higher risk of mispredicting value clamping. However, a greater a_{min} builds a shortcut triggering condition that is easier to meet, as $P(a_i < a_{min})$ is greater.

We leave the choice of the suitable a_{min} to the user by allowing a configurable parameter *conf*. The parameter simply sets the minimum acceptable probability $P(a_m < b_{lower} | a_i < a_{min})$ that the chosen a_{min} should satisfy. ConvReflex then tests all possible a_{min} from the greatest to the least. For example, for the case in Figure 5 where we look for a a_{min} for step 8 of the convolution operation, when the target *conf* is set to 100%, we find the first a_{min} to satisfy $P(a_{16} < -128 | a_8 < a_{min}) = 100\%$ is -284. In other words, if we choose to trigger the shortcut when the intermediate result at step 8 is smaller than -284, we expect a 100% chance of seeing a clamped final result according to the profiled history behavior of the convolution kernel.

Note that even when *conf* is set to 100%, it might happen that the accuracy loss introduced by the shortcuts is beyond the budget, because some corner cases where an intermediate result does not lead to clamping may not be seen in the profiling process. Thus, for *conf* at 100%, ConvReflex allows ignoring a certain proportion of edge cases that already satisfies the requirement when sweeping all possible a_{min}'s from the greatest to the least. For example, for a *conf* parameter of 100%, if ignoring 20% edge cases, the first a_{min} that satisfies the requirement in Figure 5 will not be -284, but -291 instead. This design enables having a safety margin in the step to create shortcuts, and gives ConvReflex the ability to limit the accuracy loss to an arbitrarily small budget.

Shortcut position. ConvReflex performs the profiling such as in Figure 5 and the process to choose a shortcut for each computation step of a convolution kernel. In the example in Figure 6, we show the profiled data for all 16 computation steps in the convolution kernel. For each step, we also mark the choice of shortcut triggering condition with vertical dotted lines and notations about a_{min}, if any. The notations additionally contain extra information, $P(a_i < a_{min})$, which is the probability that the shortcut can be triggered.

ConvReflex allows only one shortcut in each convolution kernel, and thus, the best shortcut will be chosen out of the 16 steps. The choice is simply based on the expected number of steps they

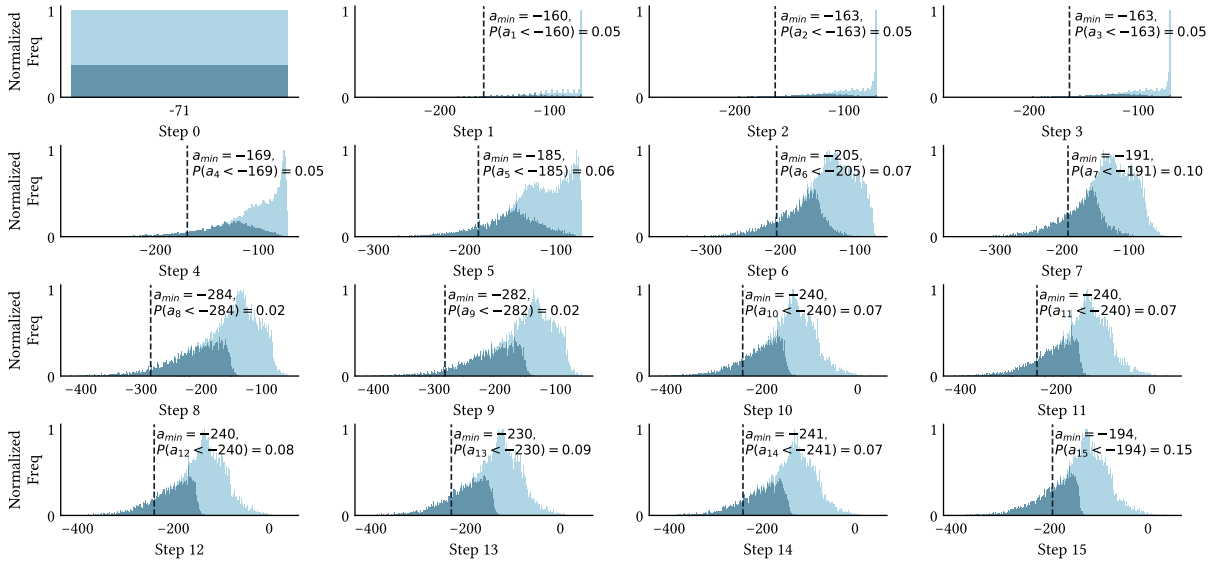


Figure 6: Visualization of the profiled data at each computation step of a convolution kernel, together with the choice of shortcut triggering condition chosen at each step with $conf = 100\%$**.** *In each sub-figure, the bar chart shows the profiled data in the format of normalized frequency of the intermediate result's value and the value leading to a clamped final result at that step. The vertical dotted line and the notation in each sub-figure, if plotted, describe the chosen shortcut triggering condition at* $conf = 100\%$*. Additionally, the notations display the probability that the shortcut can be triggered, i.e.* $P(a_i < a_{min})$*, which will be used when selecting the best shortcut position from the 16 steps.*

can skip, described by:

$$(m - i) * P(a_i < a_{min}) \tag{5}$$

which is the product of the remaining computation steps that can be skipped and the probability that a shortcut will be taken. In the example in Figure 6, the shortcut will be set after computation step 7, as the shortcut here can skip 9 steps, with a 0.10 probability of being triggered, which results in an expected computation step omission of 0.9, ranking the highest among all choices.

3.4 Evaluating and Adjusting Shortcut Selection

It might be not straightforward to adjust the *conf* parameter directly since there is no direct feedback as to how the model will be affected by different *conf* settings. Thus, what ConvReflex exposes to the user is another parameter: the budget for accuracy loss, denoted as k. The user only needs to set up a upper limit for the accuracy loss in the shortcut-enabled CNN model, and ConvReflex iteratively looks for the best shortcuts which yields the most gain in time saving, while strictly limiting the accuracy loss below k.

With the constraint k, after profiling, ConvReflex iteratively attempts to create shortcuts and evaluating the accuracy of the produced model, following a series of *conf* settings with increasingly relaxed tolerance for error. Our preset *conf* setting series start from 100% and end at 90% with gradually increasing strides from 0.1% to 5%. The user can also modify this default setup according to their needs. The accuracy loss of the generated model compared to the baseline model will be evaluated based on a different subset of the test set than the subset used for profiling. If the accuracy loss is still within the budget, the next, more relaxed *conf* parameter will be used in the next iteration when re-selecting shortcuts and evaluating. The process stops when the generated model can no longer stay within the accuracy loss budget k, and the model generated in the last iteration will be selected as the final output of the pipeline.

3.5 Implementation

In this subsection, we explain the implementation of the components of ConvReflex.

Model execution support. We implement support for code generation and execution for `.tflite` format CNN models, which produces C code that can be deployed on MCUs. We base our code generation framework on MCUNet [31]'s TinyEngine module [34] and use this as the baseline for our comparison, since TinyEngine exhibits state-of-the-art performance surpassing other popular edge AI inference frameworks, such as Google's TF-Lite Micro [16], ARM's CMSIS-NN [8] or STMicroelectronics' X-CUBE-AI [50]. We tailored TinyEngine to fit the instruction set architecture of the Cortex-M0+ MCU [9] used in our evaluation and added supports for the operators involved in the neural networks we use. We also extend the existing code generation functionalities to support generating inference code with the extra information describing the shortcuts, and modify library functions in TinyEngine to support the shortcut functionalities. We do not compare our work with saturation-aware convolution [29], because its space overhead exceeds our experiment platform's memory limitations for several workloads, rendering the comparison partial. However, *it is indeed feasible to implement ConvReflex based on other frameworks* which supports `.tflite` model execution on MCUs.

We add in support for generating inference code with extra shortcut information based on TinyEngine's code generation. In the generated code, the position and the triggering condition of each kernel's shortcut are written as C arrays and placed alongside each layer's weight array. Our modified convolution function accepts additional parameters that pass in the position and condition for each of the convolution kernels in a layer in the format of C arrays, which will be further passed to the matrix multiplication function that is called by the convolution function after `im2col` conversion. The execution logic of the modified matrix multiplication in our convolution function is similar to the example in Figure 3.

Our implementation only consider lower-bound value clamping, i.e. the type of clamping where the final value is clamped to the output neuron's lower boundary, since we find that lower-bound value clamping takes up more than 99.5% of all the observed value clamping on average across all workloads involved in our evaluation. While there is no additional complication or significant memory overhead to include upper-bound clamping check, the computation overhead of one extra conditional branch is unlikely to be worthwhile, because this check likely does not skip any computation most of the times, giving very limited gain, if any.

Profiling support. ConvReflex's profiling functionality involves modified TinyEngine libraries to output computation traces when executing CNN inference and a Python script to parse the traces.

To generate the computation traces, we integrate the data gathering and printing functionalities into TinyEngine's library functions. With associated compilation flags, the convolution and matrix multiplication functions print the trace of the intermediate result of each convolution operation, alongside the information about the final result before and after clamping. Note that the trace can be gathered *on any platform*, but not limited to the particular MCU where the CNN model will be deployed. We develop a Python script to parse and organize the data in the execution trace files. Data are organized at the granularity of computation steps. The script saves the profiled data as `pickle` files, and supports incremental updates to saved data, so that profiled data of new computation traces can be merged into existing data.

Shortcut selection and accuracy evaluation. We develop a Python script to implement the shortcut selection process. Besides the profiled data, the script requires the setting for the *conf* parameter and the proportion of ignored edge cases if $conf = 100\%$, as explained in Section 3.3. For one convolution kernel, the shortcut selection script first finds the suitable shortcut triggering condition at each computation step, and then compares the expected gain in computation step omission at all computation steps to find the one with the most potential gain.

Accuracy evaluation is supported by a series of bash scripts automating the process. Taking as input k, the accuracy loss budget, the scripts drive the workflow to iteratively evaluate the accuracy of the CNN model produced by ConvReflex and adjust the *conf* parameter to generate a new model with different shortcuts. The process starts with generating code for, compiling and executing the CNN model. The model is executed in a loop that runs all inputs from the subset of test set used exclusively for accuracy evaluation in pipeline. The scripts then gather accuracy information from the outputs of the executions, and decide whether the pipeline can finish, or it should modify the *conf* parameter and start another iteration. Note that, just like profiling, during accuracy evaluation, the C code of the candidate CNN models can be run on any platform.

Table 1: Models evaluated. *We test various of benchmarks with different task types and a wide range of sizes.*

Model	Task Type	Model Size in .tflite Format (kB)
Mini ResNet [17]	Audio event detection	140
Mini ResNet (2 stacks)		467
Mini ResNet v2		142
Mini ResNet v2 (2 stacks)		470
Hand posture recognition CNN	Hand posture recognition	6.5
HAR IGN [21] 24	Human activity recognition	6.7
HAR IGN 48		6.5
Mobilenet 96x96 grayscale	Image classification	301
Mobilenet 96x96x3		301
FdMobilenet		192
ST FdMobilenet		217
ResNet 8		94
ST MNIST CNN		19

4 Evaluation

We conduct experiments to investigate the time and energy saving ConvReflex can provide on real development boards. We also investigate the overhead brought by having the shortcuts in the convolution kernels, namely, flash space overhead and accuracy loss. Our evaluation results show that:

- Compared to a vanilla model with no shortcuts, ConvReflex enables on average 11% and 14% time saving when using 1% and 3% accuracy loss budget (k), respectively.
- With ConvReflex, 15% and 20% of the computations in convolutional layers are skipped, when using 1% and 3% accuracy loss budget (k), respectively.
- ConvReflex causes small space overhead and little changes in the current draw, and thus the energy savings are nearly proportional to the time savings.
- On the test set, the accuracy drop introduced by ConvReflex is still within the 1% and 3% budget.

4.1 Experimental Setup

Platform and benchmarks. We carry out experiments on a STM32 G0B1RE [51] development board with a Cortex-M0+ MCU [9], with 144 kB of RAM and 512 kB Flash memory.

We evaluate with CNN models from STM32 Model Zoo Suite [48, 49], and we select models that do not exceed the Flash and RAM capacity of our board. They are summarized in Table 1. The models we select cover a wide range of task type, including audio event detection, hand posture recognition, human activity recognition and image classification. The size of the models vary from several kBs to hundreds of kBs. The architectures of the models also include various structures, such as depth-wise convolutional layers and residual connections. We use models in `.tflite` format, as this is the format that ConvReflex supports as input. For several models whose `.tflite` versions are not released in STM32 Model Zoo, we

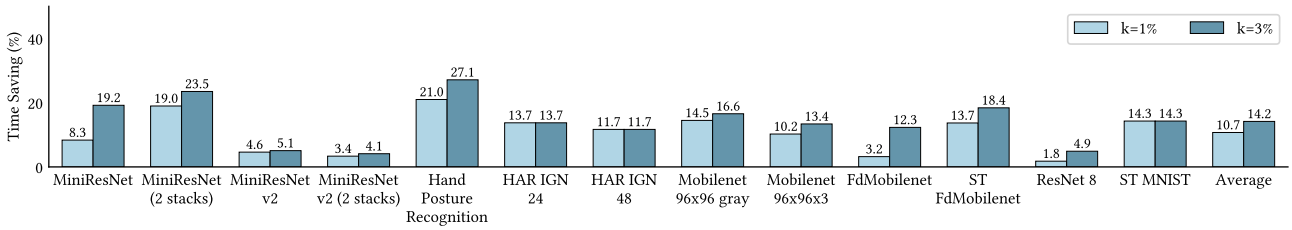


Figure 7: CNN inference time reduction by enabling ConvReflex. *ConvReflex enables up to 27% time saving.*

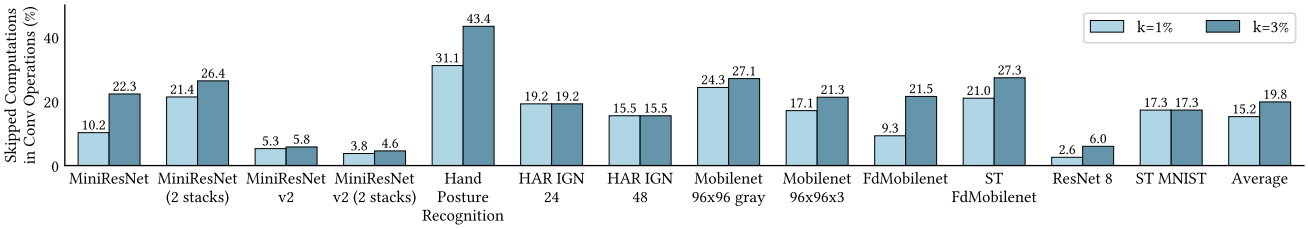


Figure 8: Skipped computation steps in convolution kernels by enabling ConvReflex. *ConvReflex helps models to skip up to 43.4% of the computation steps in convolution kernels. On average, 15.2% and 19.8% computation steps are skipped, for $k = 1\%$ and $k = 3\%$, respectively.*

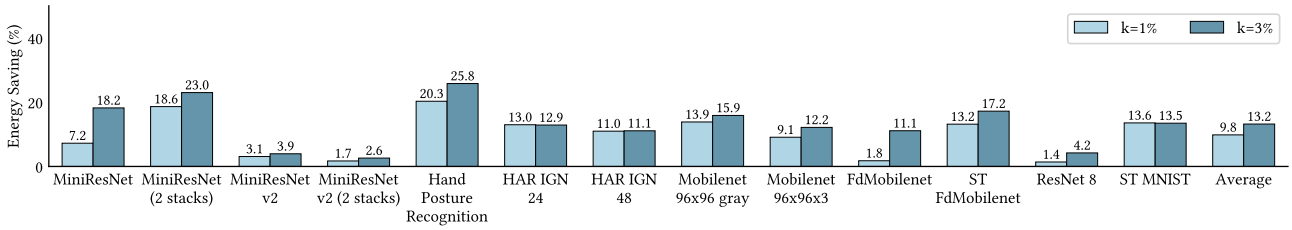


Figure 9: Energy saving by enabling ConvReflex. *The energy saving is close to the percentage of time saving.*

convert them into `.tflite` format using the quantization scripts provided in STM32 Model Zoo [49].

Metrics. We investigate the following aspects:

(1) *Average execution time.* We measure execution time on the MCU using the `HAL_GetTick()` function in STM32 libraries. This function returns time at a millisecond resolution.
(2) *Skipped computation steps.* We modify the convolution functions to record and print information regarding skipped computation steps. For this metric, we execute the models on a desktop machine, because no platform-dependent information is generated.
(3) *Average energy consumption.* We first measure the average current consumption for each run on the MCU using a Nordic Power Profiler Kit II. This tool can provide a stable 3.3V voltage and supports high-precision current measurements (10kHz sample rate, 100nA resolution) via direct pin connections. Then, the energy consumption is computed as the product of the average current, supply voltage and average execution time.
(4) *Space overhead.* We obtain information of the Flash memory usage with the help of the tools in STM32CubeIDE.
(5) *Accuracy at run-time.* As the accuracy loss budget k only strictly constrains the accuracy loss in the compile-time evaluation, we further investigate the run-time accuracy of the shortcut-enabled models using a different subset of the test set which is not seen in the compile-time pipeline.

We compare the models generated by ConvReflex with the baseline models without shortcuts. For a fair comparison, the C code of the models are generated with the same, TinyEngine-based code generation tool in ConvReflex, the only difference being whether the shortcuts are enabled.

Preparation. Using ConvReflex, we produce CNN inference code for MCUs under two setups for the accuracy loss budget: a conservative setup where we set the accuracy loss budget k to 1%, and a more aggressive setup where we set k to 3% to tolerate more error and trade it off for time saving.

First, we split the test set for each model into three: a small split of tens of inputs for compile-time profiling, a relatively larger split with hundreds of inputs for compile-time accuracy evaluation, and the rest as the actual test set. We do not conduct experiments with any data known to the models, and make sure the input data used by ConvReflex's pipeline are not reused for further tests.

Table 2: Characterization of the result after running ConvReflex's pipeline on the models. *We show the eventual conf parameter chosen by ConvReflex's pipeline, and the accuracy comparison between the baseline and shortcut-enabled model.*

Model	Baseline Acc. (%)	k=1%		k=3%	
		Chosen conf (%)	Acc. in eval (%)	Chosen conf (%)	Acc. in eval (%)
Mini ResNet	85.63	99.9	84.69	97	84.06
Mini ResNet (2 stacks)	85.63	97	85	95	83.8
Mini ResNet v2	86.89	92	86.56	90	84.38
Mini ResNet v2 (2 stacks)	89.06	95	89.38	92	86.25
Hand posture recognition CNN	98.44	100*	97.5	99.9	95.62
HAR IGN 24	92.19	95	91.56	95	91.56
HAR IGN 48	91.56	90	94.69	90	94.69
Mobilenet 96x96 grayscale	76.56	97	76.88	95	73.75
Mobilenet 96x96x3	90.63	99.5	89.69	98	89.06
FdMobilenet	85	100*	84.38	97	83.13
ST FdMobilenet	86.56	99.2	87.19	97	85.31
ResNet 8	84.38	99.9	84.06	95	82.19
ST MNIST CNN	90.31	99	90	99	90

* Profiled with 1/6, or 16.7% edge cases ignored, as explained in Section 3.3. The highest preset $conf$ at 100% cannot keep the models' accuracy loss within the $k = 1\%$ budget.

We then run ConvReflex's pipeline under the two k setups for all of the models to be evaluated, on a Macbook pro M1 machine. In the iterative *conf* adjusting stage, we use up to 12 *conf* parameter ranging from 100% to 90%. For small models such as HAR IGN 24, it takes about 10 minutes to run the whole pipeline for profiling, shortcut creation, accuracy evaluation, and iterative *conf* adjusting. For larger models such as ResNet, the whole process takes 5 to 7 hours. The profiling stage usually takes the most time, which can be 1 hour to 1.5 hours for larger models, but the result of profiling can be reused for later iterative stages.

The pipeline eventually produces CNN inference code written in C that is ready to be deploy onto the MCU. In Table 2, we show the eventual choice of *conf* parameters after the iterative adjusting process, and the accuracy comparison between each model's baseline version and their shortcut-enabled version. Some models, namely, HAR IGN 24, HAR IGN 48 and ST MNIST CNN end up with the same *conf* parameter with different k setups, since none of the *conf* parameters can produce a model with more than 1%, but less than 3% accuracy loss. With a $k = 1\%$ accuracy loss budget, Hand posture recognition CNN and FdMobilenet use a $conf = 100\%$ but additionally ignores 1/6 edge cases, since the highest preset $conf = 100\%$ alone cannot keep the model's accuracy loss within the 1% budget. There are also instances where the evaluated accuracy of a model produced by ConvReflex is greater than the baseline, especially with the $k = 1\%$ setup. We believe the reasons to this is two-pronged: on the one hand, the 1% difference is actually smaller than the natural deviation of the accuracy evaluation result; on the other hand, injecting noise may in turn help improve the accuracy of a network [37].

4.2 Time Saving

We report the time saving by ConvReflex in Figure 7. With a $k = 1\%$ accuracy loss budget, ConvReflex enables the MCU to save on average a 10.7% fraction of the execution time, which goes up to a 21% saving for the hand posture recognition CNN. When the accuracy loss in compile-time accuracy evaluation is allowed to go up to 3%, the average time saving is 14.2% (up to 27%). Indeed, hand posture recognition CNN shows the highest time saving. According to Figure 2, as much as 61% of the computations in the convolutional layers of this model have no effect on the output neuron's final value; we believe its high potential in the computation steps that can be skipped contributes to its high on-board time saving.

ConvReflex yield high performance gains for most of the CNN models, with exceptions for MiniResNet v2 and 8-layered ResNet. This is because these models contain convolutional layers with no activation function. For such layers, value clamping happens very rarely because there are no extra constraints like ReLU or ReLU6 on the layer's output, and thus ConvReflex finds no opportunities for creating shortcuts. For example, more than half of the convolution layers in ResNet 8 have no activation function, and these layers consist of 59% of all convolution computation steps of the model.

4.3 Skipped Computation Steps

We conduct further investigation on the number of skipped computation steps in our experiments, shown in Figure 8. On average, 15.2% and 19.8% of the computation steps in convolutional layers are skipped, with setups of $k = 1\%$ and $k = 3\%$, respectively. For the hand posture recognition CNN which exhibits the highest time saving, up to 43.4% of the computation steps is skipped ($k = 3\%$).

The models we consider consist mainly of convolutional layers [48]. The vast majority of the models' execution time is spent in convolutional and fully-connected layers, which are implemented as a special case of 1×1 convolution in TinyEngine [34]; other operations such as pooling, reduce, point-wise addition and multiplication take negligible time. Thus, the data in Figure 8 and Figure 7 indicates a positive correlation between the skipped computation and the time saving. The relation appears to not be strictly proportional. We believe the variance is related to how many times the shortcuts are triggered. For example, a kernel with more computation steps is less affected by the overhead of executing the shortcut condition checks, as the overhead is relatively small compared to the kernel's total execution time.

Unfortunately, even if convolution takes up the majority of execution time of these CNN models, the relation between the two cannot be described by a simple linear model, because the execution overhead of the shortcuts is affected by multiple complex factors. For example, both the shortcut triggering condition and taking the shortcut require branching instructions, but the latter is not always executed for each and every shortcut, but depends on whether the triggering condition is met.

Increasing k from 1% to 3% brings an increase in the percentage of skipped computations for all of the models, except the ones where the two setups generate the same shortcut settings. Among these, MiniResNet, Hand Posture Recognition, FdMobilenet and ResNet 8 enjoy the most increase. As shown in Table 2, these four benchmarks have the most strict *conf* setting when $k = 1\%$. We believe that there is a diminishing marginal effect in relaxing the *conf* parameter, i.e. as *conf* becomes more and more relaxed, the return in skipped computations becomes less.

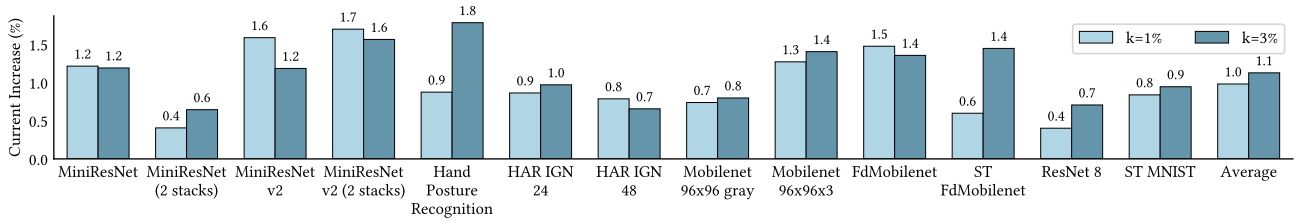


Figure 10: Increase in current draw. *ConvReflex incurs very small increase in the current draw.*

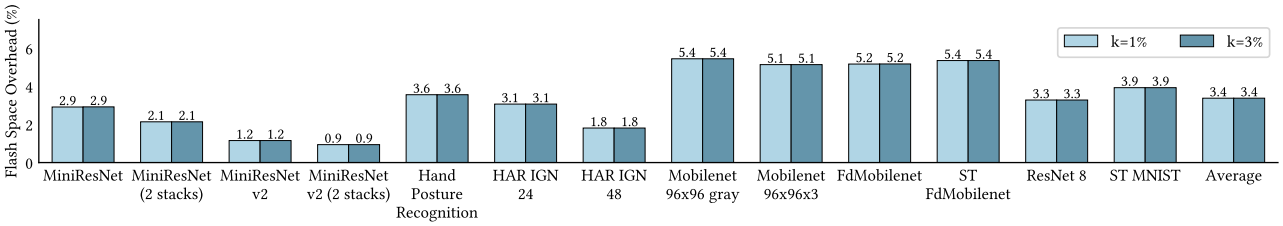


Figure 11: Flash space overhead introduced by ConvReflex. *ConvReflex introduces negligible space overhead. The average overhead in flash usage is merely 3.4%, and the highest overhead is 5.4%. The space overhead is nearly the same for $k = 1\%$ or $k = 3\%$, because k seldom affects the number of shortcuts.*

4.4 Current Draw and Energy Saving

We run the models on a real MCU and measure the average current consumption of the development board with a power profiler, which provides stable supply voltage and profiles current and energy consumption. With the measurement of average execution time and current draw for each model, we estimate the energy consumption with the product of execution time, supply voltage and current.

We report the energy saving in Figure 9. The reduction in energy consumption is close to the reduction in execution time; ConvReflex saves 9.8% and 13.2% energy with k setups of 1% and 3%, respectively. This is expected, since on ultra-low-power MCUs, the energy consumption is nearly proportional to the active time [38, 41, 45, 47], as they mostly lack features such as DVFS [4, 6].

Additionally, we report the current draw in Figure 10. We observe very small changes in current draw when running the models optimized by ConvReflex compared to the baseline models. On average, the current only increases by 0.98% and 1.13% with $k = 1\%$ and $k = 3\%$, respectively. However, as small as the changes are, the current draw always increases on models produced with shortcuts. We believe the small increase of current draw is due to the extra branch instructions introduced by the shortcuts, since branching contributes more to energy consumption than common instructions [36].

4.5 Space Overhead

As ConvReflex uses extra C arrays to pass the shortcut information, i.e. the position of the shortcut in the convolution kernel's computation steps and the condition to trigger the shortcut, to convolution kernels, we also investigate its space overhead. As in Figure 11, we find the model inference code generated by ConvReflex only takes up 3.4% extra flash memory on average, and the overhead is no more than 5.4%.

In Figure 11, the space overheads of using setup $k = 1\%$ and $k = 3\%$ appear the same. The actual difference between two setups is usually only a few bytes, if any. This is because the space needed for one shortcut is always the same, and the space overhead is proportional to the number of shortcuts. Changing the accuracy loss setups k seldom affect the number of shortcuts, but will only change their position and triggering conditions.

MiniResNet v2 has the least overhead, and this is due to the network having only a few layers with shortcuts. On the other hand, the four Mobilenet or FdMobilenet models have the most space overhead, since almost all layers in these models have shortcuts.

4.6 Deployed Model Accuracy

As the accuracy loss budget k only puts constraints on the accuracy during compile-time, we further investigate whether the accuracy of models during our real-board evaluation can stay within the constraints of k, and report it in Figure 12. Note that this investigation does not include the four MiniResNet models, since they are trained and tested on the ESC-10 dataset [40] which contains very limited number of inputs. As we need to split out parts of the test set for profiling and compile-time accuracy evaluation, we do not have enough input left in the rest of the test set to observe the accuracy change with a resolution under 1%. Thus, the evaluation for the four MiniResNet models would not be trustworthy, and we omit them from this subsection.

As shown in Figure 12, For the models generated by ConvReflex with $k = 1\%$, the accuracy loss is at most 0.7%. For those generated with $k = 3\%$, the accuracy loss is at most 1%. Similar to the accuracy evaluated at compile-time in Table 2, there are also models exhibiting positive accuracy change, i.e. increased accuracy, despite ConvReflex infusing shortcuts into them which in theory can bring in error. We also believe this is because injecting noise into the network in turn helps improve its accuracy [37].

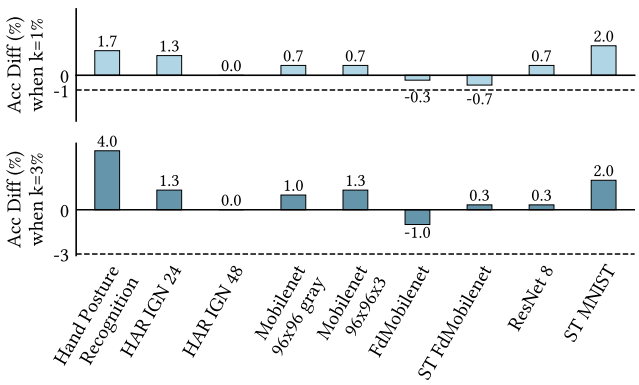


Figure 12: Deployed model accuracy compared to baseline. *The accuracy loss budget k only guarantees a constraint in compile-time evaluations. When the model is out of the compile-time pipeline, we re-evaluate the model accuracy with unseen test inputs. The result confirms that the accuracy loss is still under 1% and 3%, which are the k values used at compile time.*

5 Discussion

In the following, we articulate the rationale behind some key design choices and discuss limitations of our work.

Generalizability. The design principles of ConvReflex are applicable to neural network layers or structures with the following properties: (i) the value of each neuron is decided via a sequence of computations, so that there are opportunities for early-exits, and (ii) each neuron's value is capped at a set of fixed boundary, so that it is practical to predict the final value. Thus, ConvReflex is not limited to CNNs. It can be directly applied to fully-connected layers, as they are essentially special cases of convolution layers with 1×1 filters. ConvReflex may also be applied to transformer models [52], provided that the model contains non-linear activation functions that enforce clear upper and/or lower boundaries on the computation result. However, the trade-off between the overhead and gain of ConvReflex on more complex neural networks are to be empirically analyzed.

ConvReflex may be ported to more capable hardware platforms. There are no explicit hardware requirements in the design of ConvReflex. We do acknowledge that ConvReflex may be more effective on scalar, in-order platforms because SIMD vectorization can weaken the effect of ConvReflex since the integrity of vectorized computation must be retained, leaving fewer choices for placing shortcuts. We also acknowledge that the overhead of conditional branches may be relatively larger in more complex processor pipelines. We design ConvReflex prioritizing energy efficiency, targeting energy-scarce use cases where the computing system may even operate on harvested energy, and thus ultra-low-power MCUs [41] are usually the target platform. We experiment with a Cortex-M0+ platform, because it is a truthful representative of such platforms and supports the underlying ARM CMSIS-NN [8] libraries which our baseline TinyEngine [31, 34] relies on.

Profiling data. Data from the test set can be trusted as profiling set inputs. In the first place, a neural network model is trained on a dataset that is believed to be representative of real-world use cases. If a drastically different, never-seen data input is fed into the network, indeed the ConvReflex shortcut triggering conditions may fail to work properly, but the neural network itself cannot correctly classify the input, either.

The profiling set does not need to be large to be representative of real-world data. In our experiments, the profiling sets are limited to only 32 inputs. Because the same convolution kernel can operate on the same layer input up to hundreds of times, even a small profiling set suffices to collect information about the behavior of each convolution kernel.

6 Conclusion

We presented ConvReflex, a toolchain aimed at supporting efficient CNN inference on ultra-low-power MCUs by predicting value clamping. For a convolution operation that generates a value that will be clamped to the boundaries defined by the allowed value range of the output neuron, it is unnecessary to compute the exact convolution result, since the result eventually will be clamped to a certain value. ConvReflex enables convolution operations to predict such value clamping behavior, and take a shortcut to jump to termination, skipping the remaining computations. By analyzing the data obtained from compile-time profiling, ConvReflex carefully selects the statistically optimal condition to trigger the shortcuts as well as their best location in the execution flow of a convolution kernel, so that the shortcuts may yield the most gain in time saving, while introducing little to no error. These procedures are done under the constraint of an accuracy loss budget given by the user, representing the maximum tolerated accuracy loss induced by the shortcuts in a CNN model. We evaluate ConvReflex on 13 various CNN models on a development board equipped with a Cortex-M0+ MCU, and obtain on average 11% (up to 21%) execution time reduction with an accuracy loss budget of 1%. With 3% accuracy loss budget, the gain in time saving is 14% (up to 27%).

Acknowledgments

This work is supported by the Swedish Foundation for Strategic Research (SSF) grant FUS21-0067. Part of the data analysis was enabled by resources in project UPPMAX 2025/2-258 provided by Uppsala University at UPPMAX.

References

[1] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *Proc. of the 18th ACM Conf. on Embedded Networked Sensor Systems (SenSys '20).*

[2] Abien Fred Agarap. 2019. *Deep Learning using Rectified Linear Units (ReLU).* arXiv:1803.08375

[3] Mohammad Hasan Ahmadilivani, Mahdi Taheri, Jaan Raik, Masoud Daneshtalab, and Maksim Jenihhin. 2024. A Systematic Literature Review on Hardware Reliability Assessment Methods for Deep Neural Networks. *ACM Comput. Surv.* 56, 6, Article 141 (Jan. 2024), 39 pages.

[4] Saad Ahmed, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. The Betrayal of Constant Power× Time: Finding the Missing Joules of Transiently-Powered Computers. In *Proc. of the 20th Intl. Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES '19).*

[5] Saad Ahmed, Bashima Islam, Kasim Sinan Yildirim, Marco Zimmerling, Przemysław Pawełczak, Muhammad Hamad Alizai, Brandon Lucia, Luca Mottola,

Jacob Sorber, and Josiah Hester. 2024. The Internet of Batteryless Things. *Commun. ACM* 67, 3 (Feb. 2024), 64–73.

[6] Saad Ahmed, Ain Qurat, Junaid Siddiqui, Luca Mottola, and Muhammad Hamad Alizai. 2020. Intermittent Computing with Dynamic Voltage and Frequency Scaling. In *Intl. Conf. on Embedded Wireless Systems and Networks (EWSN '20)*.

[7] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *Proc. of the 45th Annual Intl. Symposium on Computer Architecture (ISCA '18)*.

[8] ARM. 2024. CMSIS NN Software Library. https://arm-software.github.io/CMSIS-NN/latest/index.html

[9] ARM. 2025. Cortex-M0+ Product Support. https://developer.arm.com/Processors/Cortex-M0-Plus

[10] Giorgos Armeniakos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. 2022. Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey. *ACM Comput. Surv.* 55, 4, Article 83 (Nov. 2022), 36 pages.

[11] Fulvio Bambusi, Francesco Cerizzi, Yamin Lee, and Luca Mottola. 2022. The Case for Approximate Intermittent Computing. In *Proc. of the 21st Intl. Conf. on Information Processing in Sensor Networks (IPSN '22)*.

[12] Rei Barjami, Antonio Miele, and Luca Mottola. 2024. Intermittent inference: Trading a 1% accuracy loss for a 1.9 x throughput speedup. In *Proc. of the 22nd ACM Conference on Embedded Networked Sensor Systems (SenSys '24)*.

[13] Rei Barjami, Antonio Miele, and Luca Mottola. 2025. On the Sweet Spot of Intermittent Inference in the Battery-less Internet of Things. In *the 33rd Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '25)*.

[14] Naveed Anwar Bhatti, Muhammad Hamad Alizai, Affan A. Syed, and Luca Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Trans. Sen. Netw.* 12, 3, Article 24 (Aug. 2016), 40 pages.

[15] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. 2021. A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction. In *the 51st Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN '21)*.

[16] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. 2021. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. In *Proc. of Machine Learning and Systems (MLSys' '21)*.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR '16)*.

[18] Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. 2020. FT-ClipAct: Resilience Analysis of Deep Neural Networks and Improving Their Fault Tolerance Using Clipped Activation. In *Proc. of the 23rd Conference on Design, Automation & Test in Europe (DATE '20)*.

[19] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. arXiv:1704.04861

[20] Younis Ibrahim, Haibin Wang, Man Bai, Zhi Liu, Jianan Wang, Zhiming Yang, and Zhengming Chen. 2020. Soft Error Resilience of Deep Residual Networks for Object Recognition. *IEEE Access* 8 (2020), 19490–19503.

[21] Andrey Ignatov. 2018. Real-time human activity recognition from accelerometer data using Convolutional Neural Networks. *Applied Soft Computing* 62 (2018).

[22] Sahidul Islam, Jieren Deng, Shanglin Zhou, Chen Pan, Caiwen Ding, and Mimi Xie. 2022. Enabling Fast Deep Learning on Tiny Energy-Harvesting IoT Devices. In *Proc. of Conference on Design, Automation & Test in Europe (DATE '22)*.

[23] Sahidul Islam, Shanglin Zhou, Ran Ran, Yu-Fang Jin, Wujie Wen, Caiwen Ding, and Mimi Xie. 2022. EVE: Environmental Adaptive Neural Network Models for Low-Power Energy Harvesting System. In *Proc. of IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD '22)*.

[24] Seunghyeok Jeon, Yonghun Choi, Yeonwoo Cho, and Hojung Cha. 2023. HarvNet: Resource-Optimized Operation of Multi-Exit Deep Neural Networks on Energy Harvesting Devices. In *Proc. of the 21st Annual Intl. Conf. on Mobile Systems, Applications and Services (MobiSys '23)*.

[25] Chih-Kai Kang, Hashan Roshantha Mendis, Chun-Han Lin, Ming-Syan Chen, and Pi-Cheng Hsiu. 2020. Everything Leaves Footprints: Hardware Accelerated Intermittent Deep Inference. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3479–3491.

[26] Chih-Kai Kang, Hashan Roshantha Mendis, Chun-Han Lin, Ming-Syan Chen, and Pi-Cheng Hsiu. 2022. More is Less: Model Augmentation for Intermittent Deep Inference. *ACM Trans. on Embedded Computing Systems* 21, 5, Article 49 (Oct. 2022), 26 pages.

[27] Mehrdad Khani, Pouya Hamadanian, Arash Nasr-Esfahany, and Mohammad Alizadeh. 2021. Real-Time Video Inference on Edge Devices via Adaptive Model Streaming. In *Proc. of the IEEE/CVF Intl. Conf. on Computer Vision (ICCV '21)*.

[28] Alex Krizhevsky. 2010. *Convolutional Deep Belief Networks on CIFAR-10*. Technical Report.

[29] Shiming Li, Luca Mottola, Yuan Yao, and Stefanos Kaxiras. 2026. Efficient CNN Inference on Ultra-Low-Power MCUs via Saturation-Aware Convolution. In *Conference on Design, Automation & Test in Europe (DATE '26)*.

[30] Chih-Chia Lin, Chia-Yin Liu, Chih-Hsuan Yen, Tei-Wei Kuo, and Pi-Cheng Hsiu. 2023. Intermittent-aware neural network pruning. In *Proc. of the 60th Annual ACM/IEEE Design Automation Conference (DAC '23)*.

[31] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Advances in Neural Information Processing Systems (NeurIPS '20)*.

[32] Ioan Lucan Orășan, Ciprian Seiculescu, and Cătălin Daniel Căleanu. 2022. A Brief Review of Deep Neural Network Implementations for ARM Cortex-M Processor. *Electronics* 11, 16, Article 2545 (2022).

[33] Hashan Roshantha Mendis, Chih-Kai Kang, and Pi-cheng Hsiu. 2021. Intermittent-Aware Neural Architecture Search. *ACM Trans. on Embedded Computing Systems* 20, 5s, Article 64 (Sept. 2021), 27 pages.

[34] MIT Han Lab. 2020. TinyEngine. https://github.com/mit-han-lab/tinyengine

[35] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Pro. of the 27th Intl. Conf. on Machine Learning (ICML'10)*.

[36] Kris Nikov, Kyriakos Georgiou, Zbigniew Chamski, Kerstin Eder, and Jose Nunez-Yanez. 2022. Accurate Energy Modelling on the Cortex-M0 Processor for Profiling and Static Analysis. In *the 29th IEEE Intl. Conf. on Electronics, Circuits and Systems (ICECS '22)*.

[37] Hyeonwoo Noh, Tackgeun You, Jonghwan Mun, and Bohyung Han. 2017. Regularizing Deep Neural Networks by Noise: Its Interpretation and Optimization. In *Advances in Neural Information Processing Systems (NIPS '17)*.

[38] Fredrik Österlind, Luca Mottola, Thiemo Voigt, Nicolas Tsiftes, and Adam Dunkels. 2012. Strawman: Resolving Collisions in Bursty Low-Power Wireless Networks. In *the 11th Intl. Conf. on Information Processing in Sensor Networks (IPSN '12)*.

[39] Yunjie Pan, Jiecao Yu, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2023. BitSET: Bit-Serial Early Termination for Computation Reduction in Convolutional Neural Networks. *ACM Trans. Embed. Comput. Syst.* 22, 5s, Article 98 (Sept. 2023).

[40] Karol J. Piczak. 2015. ESC: Dataset for Environmental Sound Classification. In *ACM Conference on Multimedia (MM '15)*.

[41] Diletta Romano, Luca Mottola, and Thiemo Voigt. 2026. Neuro-C: Neural Inference Shaped by Hardware Limits. In *European Conference on Computer Systems (EuroSys '26)*.

[42] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. 2022. Machine Learning for Microcontroller-Class Hardware: A Review. *IEEE Sensors Journal* 22, 22 (2022), 21362–21390.

[43] Abdur-rahman Ibrahim Sayyid-Ali, Abdul Rafay, Muhammad Abdullah Soomro, Muhammad Hamad Alizai, and Naveed Anwar Bhatti. 2025. CheckMate: LLM-Powered Approximate Intermittent Computing. In *Proc. of the 23rd ACM Conf. on Embedded Networked Sensor Systems (SenSys '25)*.

[44] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. 2018. Prediction Based Execution on Deep Neural Networks. In *the 45th Annual Intl. Symposium on Computer Architecture (ISCA '18)*.

[45] Weining Song, Stefanos Kaxiras, Thiemo Voigt, Yuan Yao, and Luca Mottola. 2024. TaDA: Task Decoupling Architecture for the Battery-less Internet of Things. In *Proc. of the 22nd ACM Conf. on Embedded Networked Sensor Systems (SenSys '24)*.

[46] Muhammad Abdullah Soomro, Naveed Anwar Bhatti, and Muhammad Hamad Alizai. 2025. Approxify: Automating Energy-Accuracy Trade-offs in Batteryless IoT Devices. In *IEEE Wireless Communications and Networking Conf. (WCNC '25)*.

[47] STMicroelectronics. 2016. How to Optimize Power Consumption on STM32 MCUs (AN4777). https://www.st.com/resource/en/application_note/an4777-how-to-optimize-power-consumption-on-stm32-mcus-stmicroelectronics.pdf

[48] STMicroelectronics. 2023. Github page: STM32 Model Zoo. https://github.com/STMicroelectronics/stm32ai-modelzoo

[49] STMicroelectronics. 2023. Github page: STM32 Model Zoo Services. https://github.com/STMicroelectronics/stm32ai-modelzoo-services

[50] STMicroelectronics. 2024. X-CUBE-AI. https://www.st.com/en/embedded-software/x-cube-ai.html

[51] STMicroelectronics. 2025. STM32G0B1RE. https://www.st.com/en/microcontrollers-microprocessors/stm32g0b1re.html

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS '17)*.

[53] Yawen Wu, Zhepeng Wang, Zhenge Jia, Yiyu Shi, and Jingtong Hu. 2020. Intermittent Inference with Nonuniformly Compressed Multi-Exit Neural Network for Energy Harvesting Powered Devices. In *Proc. of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC '20)*.

[54] Yuan Yao, Xiaoyue Chen, Hannah Atmer, and Stefanos Kaxiras. 2024. TangramFP: Energy-Efficient, Bit-Parallel, Multiply-Accumulate for Deep Neural Networks. In *the 36th Intl. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '24)*.

[55] Chih-Hsuan Yen, Hashan Roshantha Mendis, Tei-Wei Kuo, and Pi-Cheng Hsiu. 2022. Stateful Neural Networks for Intermittent Systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 41, 11 (Nov. 2022), 4229–4240.