Lupe: Integrating the Top-down Approach with DNN Execution on Ultra-Low-Power Devices

Mingyuan Xiang
University of Chicago
Chicago, Illinois, USA
myxiang@uchicago.edu

Pouya Mahdi Gholami
University of Chicago
Chicago, Illinois, USA
pouya@uchicago.edu

Henry Hoffmann
University of Chicago
Chicago, Illinois, USA
hankhoffmann@cs.uchicago.edu

Abstract

Executing deep neural networks (DNNs) on ultra-low-power (ULP) microcontrollers creates enormous opportunities for new intelligent edge applications. However, manually writing optimized DNN programs for ULP devices is time consuming and error prone due to the difficulty of managing on-device accelerators. Many prior works address this problem by creating special libraries that tailor common DNN building blocks for unique accelerators of ULP devices. This is a bottom-up approach, as developers build DNNs by assembling library calls. Unfortunately, the encapsulation overhead inherent in this approach greatly reduces accelerator utilization and overall performance. Instead, we advocate for a top-down approach. We present Lupe, a code generation framework, that converts high-level DNN algorithm descriptions to ULP-optimized code. Lupe provides top-down intermittent support that significantly reduces overhead while maintaining intermittent safety. We demonstrate Lupe's benefits on an MSP430 [54], achieving 12.36× and 2.22× average speedup over two prior works across a variety of DNN models in continuous power. Moreover, Lupe reduces the average intermittent runtime costs of prior works by 96.65% and 71.15%, respectively.

CCS Concepts

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

Keywords

Software Optimization, Deep Neural Network (DNN) Inference, Embedded System, Intermittent Computing

ACM Reference Format:
Mingyuan Xiang, Pouya Mahdi Gholami, and Henry Hoffmann. 2025. Lupe: Integrating the Top-down Approach with DNN Execution on Ultra-Low-Power Devices. In *The 23rd ACM Conference on Embedded Networked Sensor Systems (SenSys '25), May 6–9, 2025, Irvine, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3715014.3722059

1 Introduction

The emergence of small, but powerful, deep neural network (DNN) models [20, 21, 32, 34, 58, 73] enables intelligent applications on ultra-low-power (ULP) devices. Although these devices include on-chip accelerators, they are extremely constrained by small on-chip memories (16–256 KB), 16-bit fixed point formats, and 4–32 MHz CPU frequency [53]. Despite the limited hardware resources, executing these models on device saves tremendous energy—as much as two orders of magnitude—compared to sending raw data back to a cloud or edge server [38]. Additionally, some ULP devices are deployed without batteries and operated intermittently, i.e., they use only power harvested from the environment and stored in small capacitors. Whether power is continuous or intermittent, ULP devices pose challenges to deliver real-time DNN inference. Specifically, efficient DNN execution requires skilled use of the accelerators, but making efficient use of these accelerators requires tedious data manipulation.

ULP accelerators perform common linear algebra functions with low latency. Prior works [13, 16, 28, 35, 72] provide libraries of DNN components where each library function encapsulates the data manipulation and accelerator calls, abstracting these confusing details away from users. We refer to this method as a **bottom-up** approach because **users assemble their complete DNN from many calls to individual library components**. For example, Tails [16] provides a library of DNN methods by wrapping vendor provided accelerator functions, each of which independently initializes and deallocates the function's parameters in the accelerator's dedicated scratchpad memory. This encapsulation is necessary to support the general use case—i.e., each one must manage the scratchpad properly so one function call will not interfere with another (e.g., by leaving the scratchpad or the accelerator in an undefined state)—but this generality creates overhead that lowers accelerator utilization.

In contrast, this paper explores a **top-down** methodology that **starts from the overall DNN structure and maps it directly into accelerated functions, reducing inter-function data manipulation overhead by leveraging the high-level context of the DNN structure**. Such a top-down approach has been explored for high-power devices (like GPUs) [37, 42, 59, 67, 71], yet—to the best of our knowledge—no one has integrated it with DNN optimization for ULP devices. Thus, we present Lupe, a code generation framework that takes DNN descriptions as input, and outputs optimized, accelerator-integrated code for ULP devices, e.g., the Texas Instruments (TI) MSP430. The advantages of the top-down approach are threefold:

First, it opens new opportunities **to optimize accelerator related data manipulation holistically**. Lupe breaks the bottom-up approach's inherent encapsulation constraints and organizes the accelerator routines for optimal data movement with minimal redundancy, considering the entire DNN layer structure.

SenSys '25, May 6–9, 2025, Irvine, CA, USA

ACM ISBN 979-8-4007-1479-5/25/05
https://doi.org/10.1145/3715014.3722059

Second, once the overhead of data movement is reduced, Lupe's top-down approach has a greater opportunity **to select the most efficient accelerator operation based on high-level DNN structure.** Thus, Lupe automatically offloads the burden of picking these operations by adapting the accelerator usage to DNN structures.

Finally, many ULP systems operate intermittently in a batteryless condition by harvesting energy from the surrounding environment. The top-down approach **allows highly efficient support for intermittent-safe DNN computation**. Lupe combines top-down DNN context with the loop continuation (LC) technique [16] to build a low-overhead, intermittent-safe DNN inference system.

Drawing these threads together, we construct Lupe as a code generation framework that converts ML models in ONNX [55] formats to a customized DNN inference library, which is compatible with the MSP430 toolchain [52]. Lupe converts models built from mainstream DNN frameworks—e.g., PyTorch [56] and TensorFlow [1]—to ONNX representations. Moreover, ONNX provides layer fusion operations [33], such as fusing batch normalizations with convolutions, that can optimize the DNN from the graph level. Lupe then takes the ONNX representation and generates efficient programs that utilize the MSP430's low-energy accelerator (LEA) [39] for either continuous or intermittent environments while maintaining inference accuracy.

Moreover, we compare Lupe with two prior works, Tails [16] and Hawaii [27], on 5 representative DNN models and 5 datasets. Our experiments show Lupe achieves an average speedup of 12.36× over Tails and 2.22× over Hawaii in continuous conditions. On average, Lupe's checkpointing schemes reduce intermittent runtime overhead by 96.65% over Tails and 71.15% over Hawaii.

We summarize our main contributions as follows:

- We recognize and analyze the significance of adopting the top-down approach for DNNs on ULP devices to achieve higher efficiency due to better accelerator utilization.
- Lupe's top-down approach greatly increases accelerator utilization by optimizing away unnecessary data movement.
- High accelerator utilization, brought by Lupe's top-down approach, creates new opportunities for optimization by tailoring the accelerator usage to the specific DNN structure.
- We propose a lightweight intermittent runtime, aided by Lupe's atomic logging scheme, that preserves runtime states efficiently.
- We open-source the implementation of Lupe[1].

With Lupe, ULP device developers can integrate DNN models into application design without manually implementing them, but instead generating accelerated ULP code from high-level DNN descriptions. Lupe greatly improves runtime efficiency in both intermittent and continuous conditions.

2 A Top-Down Approach to ULP Devices

We motivate the top-down approach by describing three opportunities. Section 2.1 provides background on implementing DNNs on the MSP430. Section 2.2 describes opportunities to reduce data manipulation overhead, while Section 2.3 discusses how the reduced overhead creates further opportunities for adaptive layer generation. Section 2.4 provides background on intermittent computing, while Section 2.5 describes how the top-down approach can reduce the overhead of supporting intermittent safety.

2.1 DNN Inference on the MSP430

The TI MSP430 processors are widely used in ULP computing. MSP430 devices contain 2 compute components: a CPU and a low-energy accelerator (LEA); each of these has a small SRAM scratchpad. For larger storage, the device has a non-volatile memory, FRAM. Finally, a direct memory access (DMA) unit can efficiently transfer data between SRAMs and FRAM.

The LEA implements efficient linear algebra and signal processing functions. To use an LEA function, one must move data from FRAM to the LEA's SRAM, allocate function parameters on the LEA's stack (also located on the scratchpad), and then invoke the LEA function from a C program. After the LEA function completes, these parameters must be freed. Additionally, the LEA manages an internal cache and when LEA functions are executed the LEA automatically copies some data from its SRAM to its internal cache, which creates some additional overhead.

There are often multiple ways to assemble DNN structures from LEA functions. For example, we can use the LEA's finite-impulse-response (FIR) filter and vector addition [51], to assemble 2D convolutions (the FIR is essentially a 1D convolution). Alternatively, we can also use the multiply and accumulate (MAC) operation that serves as a vector dot product.

Unfortunately, the LEA's tremendous performance improvements come with a cost. The required data movement and SRAM stack manipulation are tedious and error prone. Thus, the vendor packages LEA functions into special libraries to abstract away these tricky details. This is an example of the bottom-up approach, and breaking this approach's encapsulation introduces new optimization opportunities.

2.2 Reducing Data Manipulation Overhead

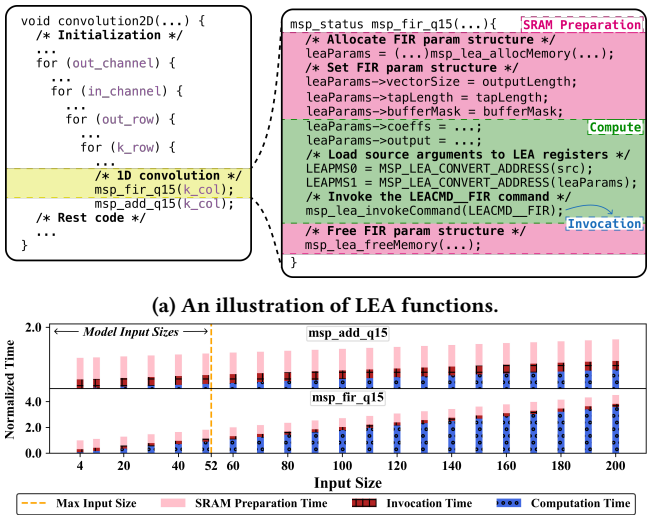
Figure 1a is a bottom-up instantiation, or implementation, of a 2D convolution (common in DNN models). This instantiation uses the vendor-supplied LEA FIR and vector addition functions in the innermost loop. Every LEA function can be divided into three parts, namely, SRAM preparation, invocation[2], and compute. SRAM preparation moves data to the scratchpad and manages the LEA stack, as described earlier. `msp_lea_invokeCommand` triggers an invocation stage inside LEA, that copies data from its SRAM to its internal cache. *Compute* refers to time spent purely on computation.

Figure 1a shows that much of SRAM preparation is redundant within a DNN layer because this instantiation repeatedly calls the same LEA functions with the same sizes. Figure 1b shows the latencies of these three parts while increasing the input data size. Preparation and invocation are dominant when input sizes are small; the TinyML [4] models we use in this paper operate on relatively small sizes (vector lengths of 52 or less).

Opportunity: *A top-down approach can (1) minimize data manipulation costs by decoupling unnecessary SRAM preparation from actual computation and (2) increase LEA function input sizes to reduce invocation overhead for every instantiation.*

[1] https://github.com/mingyuan-xiang/lupe.git

[2] We use linear regression to estimate the invocation time as the real invocation time is hard to measure.



(a) An illustration of LEA functions.

(b) Latency decomposition of LEA functions.

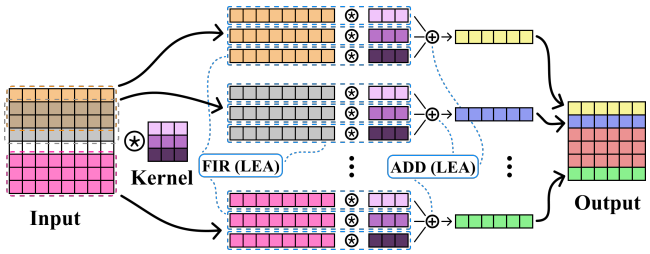
Figure 1: (a) An LEA function consists of three stages: SRAM preparation, invocation and compute, highlighted in pink, blue, green. (b) depicts latencies of LEA functions with different input sizes on an MSP430FR5994 with a 16 MHz CPU and an 8 MHz FRAM. When input sizes are small, the overhead of preparation and invocation is relatively higher.

In Figure 1a, an SRAM preparation stage dynamically allocates a data block on the scratchpad and initializes it with the operation's parameters. When computation finishes, it deallocates the data block. Although the data block needs to be properly set up, we can reorganize the program to move it outside LEA functions to minimize preparation costs. Specifically, we allocate and initialize the parameters at the beginning of DNN layer instantiations and deallocate them in the end. Therefore, we eliminate most of the preparation costs. Unlike the scratchpad, programmers have no control over the LEA's internal cache. However, we can use batched acceleration (i.e., replace many small FIR calls with a single FIR call to a much larger vector) to amortize the invocation overhead. Thus, by using the high-level DNN structure, the top-down approach provides a systematic way to reduce data manipulation costs for both preparation and invocation stages.

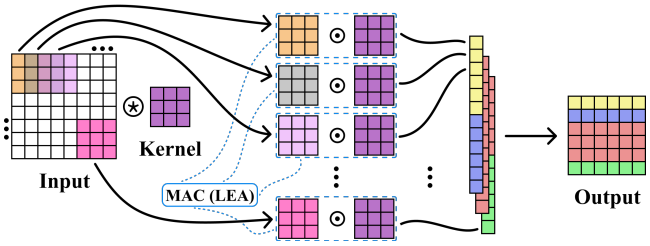
2.3 Adaptive DNN Layer Generation

As LEA supports multiple linear algebra and signal processing functions, there are multiple ways to implement a DNN layer using LEA. Making the most efficient use of LEA requires selecting the best function for a layer.

For example, Figure 2 illustrates two different instantiations of a DNN convolution layer for a single input/output channel using a $k \times k$ kernel with stride of 1, where one uses the FIR function and the other uses MAC. The FIR-instantiation breaks the input into n blocks, $n = 6$ in our example, corresponding to the height of the output, as shown in Figure 2a. We apply the FIR, i.e. 1D convolution, k times on each row within the block and accumulate k rows of



(a) Compute 2D convolution using the hardware-accelerated FIR and (vector) ADD functions.



(b) Compute 2D convolution using the hardware-accelerated MAC function.

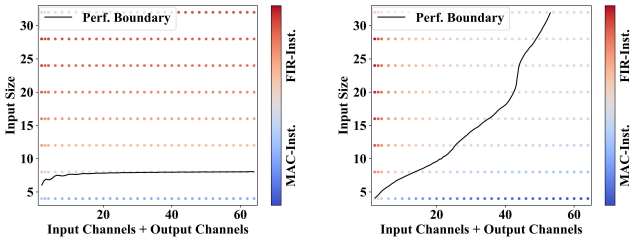
Figure 2: (a) and (b) demonstrate how to compute a 2D convolution on a $k \times k$ kernel using the hardware-accelerated FIR and MAC respectively. To use FIR, we apply it to k input rows separately and use vector addition to aggregate FIR results for every output row. For MAC-instantiations, we use a $k \times k$ MAC and write the results to output buffers directly. Although either approach can be used to compute convolutions, they perform differently based on convolution's sizes.

results using vector addition. The MAC-instantiation, in Figure 2b, breaks the input into $n \times n$ blocks. We apply a $k \times k$ MAC on every block to get the output elements.

Unfortunately, the overhead of SRAM preparation and invocation makes it hard to realize gains from tuning DNN implementations to the right LEA function. In fact, given the overhead, the best layer instantiation is typically the one that makes the fewest LEA calls. Our FIR-instantiation calls LEA functions $2 \times n \times k$ times, while the MAC-instantiation calls them $n \times n$ times. Given that $n \gg k$, FIR-instantiations will outperform MAC-instantiations in most cases. We benchmark the performance of these two instantiations on a variety of DNN settings, as shown in Figure 3. We highlight the latency differences in colors, where red means that the FIR-instantiation has better performance and blue means the vice versa. As seen in Figure 3a, without minimizing data manipulation, the MAC-instantiations are rarely higher in performance. However, if we reduce data manipulation as described in the previous section, a new opportunity arises.

Opportunity: *A top-down approach creates a richer optimization space for adapting DNN instantiations to accelerated functions.*

Once we enable the top-down optimization demonstrated in Section 2.2, neither of the two instantiations have a dominant performance advantage across all data sizes as shown in Figure 3b (note that the solid line denotes the boundary where one instantiation is faster than another). Specifically, FIR-instantiations tend



(a) Default data manipulation **(b) Optimal data manipulation**

Figure 3: Convolution performance of 3×3 kernels (a) without, and (b) with data manipulation optimization, both using an MSP430FR5994 with a 16 MHz CPU and an 8 MHz FRAM. Colors indicate performance, blue favors MAC instantiations, red favors FIR (darker shades represent higher performance). We highlight the boundary where one implementation would be favored with a solid line. With the default data manipulation (Figure 1), FIR is almost always faster. With optimal data manipulation (Section 3.1) there is a richer tradeoff space.

to perform better for small numbers of channels and large input sizes, while MAC-instantiations perform better for large number of channels and small input sizes. A top-down approach can adapt the layer instantiations to the most efficient accelerator functions.

2.4 Intermittent-Safe Computation

By harvesting energy from the environment, e.g. solar energy, radio frequency, etc [40], intermittent systems can be battery-free and self-sustainable. When energy is sufficient, the system executes programs as normal. As energy is depleted, it preserves runtime states, recharges, and then restores states to continue execution.

A major challenge of intermittent systems is to guarantee computation's correctness given unpredictable power outages. One approach is checkpointing, i.e., automatically recording and restoring runtime states [9, 19, 30, 41, 44, 45, 61]. Checkpointing can have high overhead due to the need to save and restore states. An alternative approach is to use a *task* system [12, 16, 18, 43, 46, 69, 70] such that the entire program is divided into multiple, atomic tasks and progress is automatically saved in between tasks. Tasks are atomic—they either execute completely or not at all—so they need to be short code blocks.

Since task-based systems encourage thinking about the program as small blocks of atomic code, they naturally encourage a bottom-up style that obscures high-level, algorithmic information. And while they may be lower-overhead than checkpointing there is still overhead of saving states at each task transition.

2.5 Top-down Intermittent Safety

The Tails system for intermittent-safe DNN execution introduces the idea of a Loop Continuation (LC) for intermittent safety [16]. The idea is to create longer-running tasks with loops (such as a 2D convolution), but use LC as a lightweight "checkpoint" within a loop-based task to reduce task maintenance overhead. LC works under the assumption that the program control flow—even under intermittent power disruptions—is deterministic if the current output and loop indices are preserved across power outages. This approach is thus lower overhead than either checkpointing or tasking alone. However, the Tails system uses a bottom-up programming approach, replacing some tasks with LC. While this reduces task maintenance time, the Tails evaluation still found that approximately 60% of total execution time is spent on task maintenance [16]. This overhead can be further reduced by applying a top-down approach to LC.

Opportunity: *A top-down approach could apply the key idea behind loop continuation—a lightweight checkpoint preserving the current output and loop indices—to an entire DNN program to avoid the overhead of task maintenance during DNN inference.*

Specifically, we treat the entire DNN program as a single task. Of course, to use LC for the whole DNN program, we need to *atomically log* multiple data words without causing data corruption. Section 3.3 describes how Lupe achieves this atomic logging.

3 Lupe Code Generation Framework Design

We implement Lupe as a code generation framework for both continuous and intermittent-safe DNN inferences. Figure 4 shows Lupe's internal workflow. Lupe takes an input DNN model in the ONNX [55] format and generates programs, for either continuous or intermittent computation, that can be deployed on an MSP430FR5994 [54]. The code generation consists of three phases: *Construction*, *Calibration*, and *Checkpoint Insertion*. The construction phase works on each DNN layer sequentially. It first reorganizes programs to minimize SRAM preparation and LEA invocation overhead (as motivated in Section 2.2). It then generates multiple instantiations for each DNN layer using different LEA functions (for example, generating both FIR and MAC instantiations of convolutional layers, as motivated in Section 2.3). The calibration step runs these different instantiations and times them to select the best performing one, completing the process of adaptive layer generation. Finally, the checkpoint insertion phase inserts loop continuations as described in Section 2.5.

3.1 DNN Layer Construction

In the DNN Layer Construction phase, Lupe parses the given DNN model and generates instantiations (LEA accelerated implementations) layer by layer. Multiple instantiations for a single layer may be generated (and final layer will be picked based on calibration). For each generated instantiation, Lupe attempts to **minimize the cost of accelerator usage** as motivated in Section 2.2. Specifically, it first generates code to minimize data preparation costs (see Section 3.1.1) and then batches accelerator function calls to reduce accelerator invocation overhead (see Section 3.1.2).

3.1.1 Reorganize Programs to Eliminate Preparation Costs. Every DNN layer that uses LEA can be reorganized to eliminate preparation costs. In this section, we use convolutional layers as an example of how to systematically lower these costs. The same ideas, however, apply to a wide range of DNN layer types.

Figure 2a shows how to compute a 2D convolution layer using the LEA's FIR functions. The implementation consists of four levels of `for` loop, where the LEA FIR function, i.e. `msp_fir_q15`, is called row by row. Figure 1a shows the internals of this function to demonstrate how it interacts with the LEA and the LEA scratchpad. The FIR function allocates its parameters on the LEA stack and

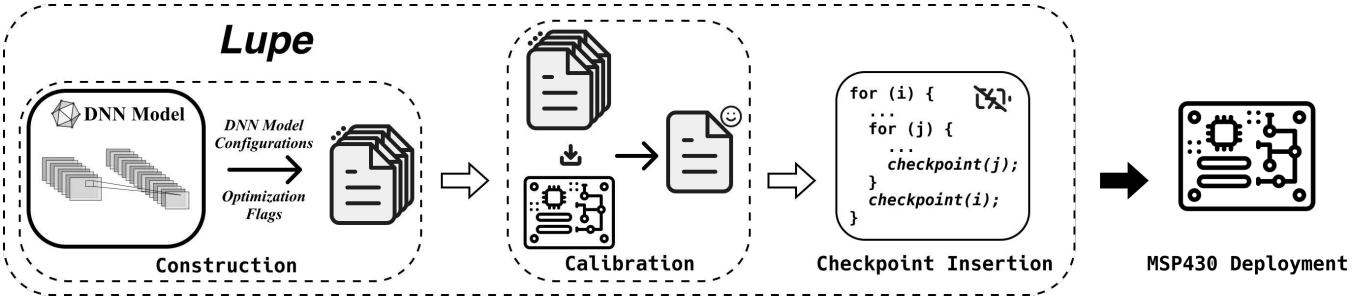


Figure 4: The system design of Lupe. Lupe takes a DNN model in the ONNX [55] format and generates an optimized program for MSP430FR5994 [54].

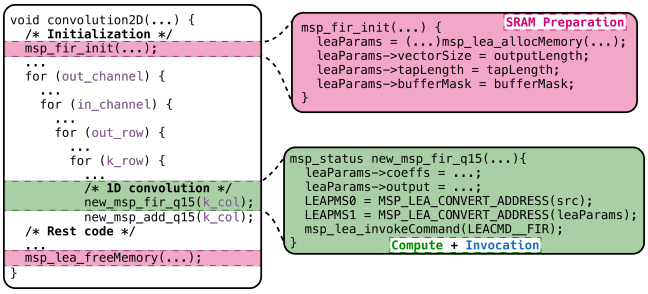


Figure 5: We illustrate how to reorganize instantiations on a 2D convolution using FIR functions. We reorganize our programs such that statements in SRAM preparation are called as few as possible. Most of them only need to be called once, at the beginning and end of the instantiation.

initializes them with the appropriate sizes (as specified in the function call). Then, it triggers LEA interrupts to invoke LEA execution. When execution finishes, it deallocates the parameters so the LEA can be used by another function.

Within a convolutional layer, many of the FIR's parameters are redundant—e,g, the vector and kernel size—and those parameters occupy a fixed location in the LEA SRAM, even across multiple FIR invocations, Therefore, we can move the allocation and deallocation of these repeated FIR parameters to the beginning and the end of the convolutional layer, respectively. As we break convolutions into blocks, that take the same input and kernel sizes, Lupe only needs to change the data addresses for the input, output, and kernel weights. Therefore, only actual LEA computation of FIR is called repeatedly. Figure 5 illustrates Lupe's reorganization of the FIR-instantiation.

Although Lupe minimizes changes to FIR parameters, the vector addition parameters may need to be modified. Specifically, Lupe uses vector addition on LEA to accumulate FIR results and compute bias offsets, where the input sizes for addition are different and need to be updated.

In general, Lupe allocates parameters of LEA functions on SRAM when entering the DNN layer and does not deallocate them until exiting this layer. We initialize operational sizes of LEA functions properly. These sizes may be changed when LEA functions are reused in the same instantiations for different purposes. Before every LEA execution, we need to update the addresses of related data, which is moved to the scratchpad before execution.

3.1.2 Batch Acceleration to Reduce Invocation Costs. Lupe uses the LEA's FIR and MAC functions to instantiate both normal 2D convolutions as well and depthwise separable ones [10]. Both instantiations are explained in Figure 2a and Figure 2b. The code must pay the invocation cost whenever an LEA computation is invoked. Therefore, to further improve LEA utilization, Lupe introduces batched-FIR-instantiations and batched-MAC-instantiations for convolutions which occupies more then 95% of the total execution time. Additionally, invocation costs of all LEA functions in these instantiations are reduced because we redesign the entire instantiations to batch data as much as possible.

As it is demonstrated in Section 2.3, FIR applies the same kernel to each block row by row. Batched-FIR, on the other hand, concatenates multiple input rows that corresponds to the same kernel row together and calls a single FIR on the concatenated row. Batched-FIR works better than batched-MAC when the input sizes are big because we can always fill the entire LEA's SRAM.

Batched-MAC concatenates input blocks and kernels across multiple input channels and does a MAC in one function call. It works well when there are a large number of channels so that we can exploit the whole SRAM. However, neither MAC nor batched-MAC is preferred under large input sizes because MAC operations need to reshape the input tensor and reshaping, that can be only done through CPU, is not efficient on MSP430.

Both batched operations try to enlarge data fed into LEA as much as possible so that Lupe produces the fewest possible LEA calls. The only limit on batch sizes is the amount of data that Lupe can fit into the LEA SRAM. Additionally, batched operations also help improve overall performance of DMA as we transfer data with larger sizes.

3.1.3 Generate Efficient Data Movement. Lupe uses two methods to manage data movement, DMA calls and unrolled loop copy. We first demonstrate how to optimize DMA calls from a top-down view based on the same insight of optimizing LEA function calls. Then we express how to select which method to use through statistics performance and DNN configurations.

An DMA call requires users to set DMA registers for input/output addresses, DMA transfer sizes, and DMA channels[3], and enable

[3]Each DMA channel can be configured independently. Lupe only uses one DMA channel for the entire program.

the DMA transfer by signaling the DMA control register. The device vendor, i.e. TI, provides packaged DMA calls to handle all such tedious, boilerplate code. However, as we discussed in previous sections, such a bottom-up approach introduces extra data manipulation overhead (of these registers). Specifically, the register offsets for different DMA channels are calculated dynamically, although register offsets are fixed, because general vendor functions make no assumption on which channel is used. Lupe, on the other hand, forces program to use the same DMA channels, i.e. channel 0. Therefore, Lupe can directly write values to these registers, diminishing the register offset calculation costs.

Although DMA is fast, its register preparation, or configuration, costs may exceed data movement costs when sizes are small, which is true for the last few layers of some DNN models. Therefore, Lupe uses unrolled loop copy instead, when faster. Figure 6 shows latencies for different data movement methods on MSP430FR5994. Particularly, Lupe uses unrolled loop copy for word sizes less than 10 as it has additional costs for keeping track of the array indices.

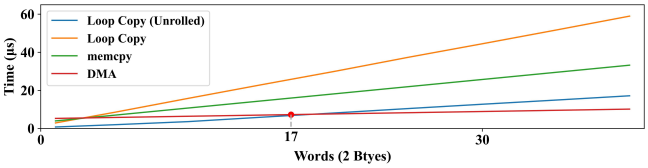


Figure 6: Latencies under different data movement methods. We should use a unrolled loop copy for small sizes and DMA copy for other cases.

3.1.4 Summary of Lupe's DNN Instantiations. Table 1 lists instantiations of supported DNN layers in Lupe. Lupe provides two instantiations for 2D convolutions, including depthwise and pointwise convolutions [10], and fully connected layers. Shortcut connections have a single instantiation. Pooling and activations, computed on CPU, also have a single instantiation. We name each instantiation after the main LEA function it uses. Many other LEA functions are also used; e.g., Lupe uses `msp_deinterleave_q15` to extract the stride results for convolutions with stride sizes greater than one. All LEA functions are reorganized to reduce preparation costs as described in Section 3.1.1.

3.2 Calibrating Layer Instantiations

Based on the insight from Section 2.3, Lupe **generates DNN layers adaptively**. The previous section described how to implement part of that idea: Lupe will generate multiple candidate instantiations for a layer, some using FIR and some using MAC, for example. The final step of adaptive generation, then, is to select the best performing instantiation for a particular layer.

3.2.1 Select Best-Performing Accelerations. Lupe times each generated instantiation individually on the MSP430, from the Construction phase, listed in Table 1, to select the most efficient one in every layer[4]. A Lupe generated DNN program consists of sequential calls of DNN layers with no parallelization between DNN layers. Therefore, an optimal implementation of every layer (in this program) leads to the best-performing DNN program. Lupe aggregates the fastest instantiation for each layer to build the entire model. Particularly, Lupe can be easily extended to newly added instantiations when new acceleration methods are available.

[4]Performance of DNN inferences on our device is stable, less than 0.001% fluctuation, so real-time benchmarking can better reflect the layer performance comparing to static analysis.

3.3 Checkpoint Insertion

To ensure correctness of DNN programs in intermittent environments, Lupe must save execution results without data corruption and recover runtime states after power outages. As discussed in Section 2.5, **Lupe uses loop continuation (LC) [16] with atomic logging to achieve intermittent safety while using a single task for an entire DNN program, eliminating task maintenance during inference**. Lupe augments continuous programs from the Calibration phase with support for intermittent safety. Section 3.3.1 describes how Lupe applies LC to the whole DNN program, while Section 3.3.2 explains how atomic logging works.

3.3.1 Preserve DNN States Seamlessly. Lupe relies on two variations of LC, a `loop`-LC [16] and a `switch`-LC [70], differentiated by types of control flows. Control flows of these two variations are deterministic if the control variables (CVs) are properly saved. `Loop`-LC constitutes the backbone of Lupe's DNN programs as it implements DNN layers in nested loops. Lupe's implementation of intermittent DNN programs are stateful for both inter-layer and intra-layer support. Therefore, a `switch`-LC helps automatically preserve program stages. We illustrate Lupe's inter-layer and intra-layer intermittent support in Figure 7.

For inter-layer support, Lupe uses a `switch`-LC to record the DNN inference progress assuming both execution results and runtime states are well-preserved within DNN layers.

Within layers, Lupe uses a mix of `loop`-LC and `switch`-LC. DNN instantiations consist of four stages, *preprocess*, in which we preprocess inputs for padding and set outputs to zeros if needed, *main*, in which we execute the computation, *bias*, in which we add bias offsets, and *exit*, in which we reset CVs for the next layer. A `recovery` function is used to restore the runtime states, e.g. CVs and execution results, when power outages happen during inference.

Logging multiple pieces of information atomically is an essential component of Lupe's intermittent runtime, whereas MSP430 has no hardware support for it. Therefore, we implement a software atomic logging scheme, i.e. `atomic_logging`, that saves CVs and execution results atomically, explained in Section 3.3.2.

Lupe's LC scheme obviates the need for a task system during DNN inference and greatly reduce the intermittent runtime overhead. Importantly, LC guarantees correctness of runtime only if information is logged atomically.

3.3.2 Log Intermittent Information Atomically. Atomic logging is necessary to prevent data corruption. While Tails [16] does it through a task system, we propose a much cheaper solution. Lupe's atomic logging scheme is built on four assumptions:

(1) Writing single variables to non-volatile memory is an atomic, operation, that is guaranteed by our hardware.
(2) Instructions are executed *in-order*, that is preserved intact by our hardware.

Layer Type	Instantiation
2D (Depthwise) Conv	FIR MAC
Pointwise Conv	MAC MPY
Fully Connected	MAC Matrix MPY
Shortcut Connection	Vector ADD
Pooling Activation	No LEA

Table 1: Acceleration methods for different DNN instantiations.

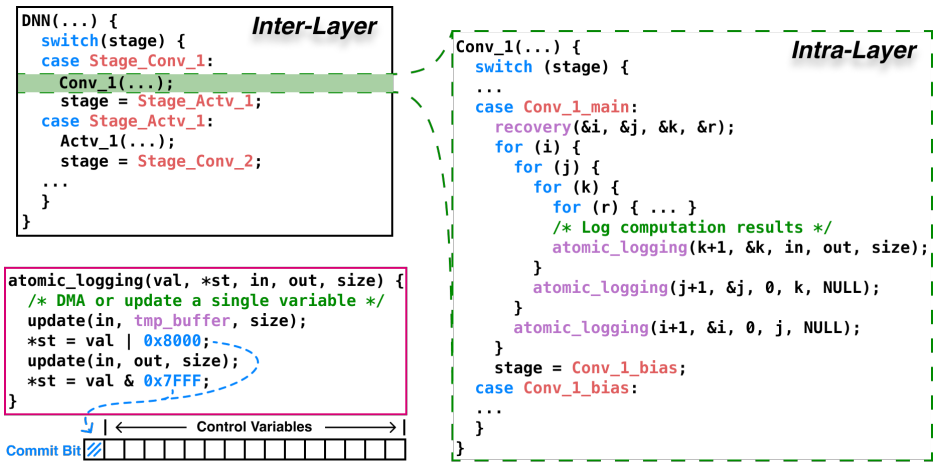


Figure 7: We use `loop`-LC and `switch`-LC to recover runtime states after power outages, while atomic logging saves information atomically avoiding data corruption. We implement atomic logging by using the MSB of CVs as a commit bit to mark if data is successfully saved to a temporary buffer.

(3) Only one CV is changed from one control flow block to the next. This is true as the control flow in Lupe instantiations is continuous, namely that Lupe does not use `loop-break` or `goto` paradigms. We achieve this by simply preventing Lupe from generating such code.
(4) Values of CVs will not exceed 2^{15} because the most significant bit, MSB, of CVs is preserved for a commit bit. This commit bit is the key insight for achieving atomic logging. Fortunately, this upper bound is far greater than any CV values we observe. Our implementation has less than a thousand states, i.e. `switch`-CVs, in the entire DNN program and `loop`-CVs are used to record input/output channels, rows/columns, that are smaller than 100.

The first two assumptions enforce that Lupe's atomic logging scheme will always be executed in the correct order. The third assumption guarantees that the tuple of saved CVs has a one-to-one mapping to control flow blocks, ergo data addresses. The last assumption ensures that Lupe will never corrupt the commit bit.

`atomic_logging` function, in Figure 7, explains how Lupe logs multiple pieces of information atomically. In this example, the goal is to save data from `in` to `out` and write `val` to a CV, `st`. We first write data from `in` to a temporary buffer. Then, we update the commit bit and `st` simultaneously by using the MSB of `st` as the commit bit to indicate data is successfully saved. Finally, we write data to `out` and clear the commit bit sequentially.

During recovery, if the commit bit is set, we will recovery runtime states using CVs and copy data to `out` from the temporary buffer. The commit bit is cleared after the recovery. On the contrary, if the commit bit is not set, which means we either just finished saving data from `in` to `out` or the data is not fully saved to the temporary buffer yet. In both cases, we take no actions.

The benefit of having atomic logging is twofold. First, it preserves runtime information with little data movement. Second, it only demands a small buffer to hold the temporary data, namely the particular row we are operating on. In our case, we need at most a 4KB buffer, to hold the LEA's SRAM. For a simple FIR-instantiation, we can achieve intermittent safety with only 80 additional bytes[5] for all our benchmarked models.

With our LC scheme and atomic logging, we can greatly reduce the data manipulation overhead of preserving intermittent runtime.

4 Evaluation Setup

We discuss our modifications to relevant implementations in Section 4.1. Our hardware setup and benchmarked models are described in Section 4.2.

4.1 Points of Comparison

We compare Lupe generated programs with three relevant implementations under both continuous and intermittent power. Particularly, no intermittent runtime is included when measuring with continuous power.

Tails [16] is the backend of the LEA-accelerated DNN library for a task-based intermittent-safe computing framework (its CPU-counterpart is called Sonic). Tails implements all convolutions by FIR-instantiations from a bottom-up view, using a task system with loop continuation, to guarantee atomic information updates.

Hawaii [27] is an intermittent inference library with limited DNN support. It is designed primarily to produce low-overhead intermittent safety over general efficient DNN execution. Hawaii is relevant for comparison since its support for intermittent safety uses a top-down approach, but it does not appear to explicitly apply other top-down optimization opportunities (such as minimizing data movement, batching accelerator calls, and adaptively generating layers). More specifically, Hawaii reduces intermittent safety overhead by preserving SRAM state, and thus tries to use the LEA SRAM as much as possible. This design indicates that

[5]The largest output width in our benchmarks is 32 and we need 8 CVs in total.

Hawaii achieves some of the benefits of the top-down approach (high SRAM utilization) as a side effect.

Notably, Hawaii also uses LC for intermittent support. However, it does not add support for atomic logging and therefore requires a very large buffer to hold temporary data across multiple input rows to recover data after power outages. In addition, Hawaii does not support convolutions with padding or strides greater than 1. Nor does it support depthwise convolutions and shortcut connections. Moreover, Hawaii assumes that for one channel, both entire input and output can fit into the LEA's 4k SRAM, which is an invalid assumption except for our smallest model, MLPClassifier. Given these limitations, to benchmark Hawaii, we divide the computation into multiple chunks while using the entire SRAM. Still, we fail to fit MobileNetV2 into the on-chip FRAM for Hawaii as it requires a larger buffer than we can fit. We add all missing modules (e.g., shortcut connection, etc.) to Hawaii from Lupe to have a fair comparison.

Lupe-BT is implemented from a bottom-up view but adds intermittent runtime using our LC schemes. Similar to Tails, Lupe-BT uses FIR-instantiations for all convolutions and MAC-instantiations for all fully connected layers. Lupe-BT ought to have performance comparable to Tails in continuous power as both simply wrap vendor supplied libraries and operate in a bottom up approach.

4.2 Evaluation Methodology

We evaluate Lupe and prior works on an MSP430FR5994 with its LEA and DMA enabled. All experiments are conducted under 16MHz CPU frequency and 8MHz FRAM with -O2 optimization. A software rebooter, based on the timer, is implemented to simulate intermittent behavior. Our software rebooter will trigger an interrupt and restart the device when it reaches the set time. We wrap the model function with the software rebooter.

We evaluate performance on a variety of models and datasets. The model architectures in our benchmarks can be found in Table 2. No model compression or decomposition are used in any of our experiments because we purely focus on computational efficiency. Graph optimizations, e.g. Layer fusion [33] for batch normalizations, are enabled through the ONNX library for all models.

5 Evaluation Results

We evaluate the following research questions (RQs):

- **RQ1:** How fast is Lupe in continuous power? (Section 5.1)
- **RQ2:** How fast is Lupe in intermittent power? (Section 5.2)
- **RQ3:** How much energy can Lupe save? (Section 5.3)
- **RQ4:** How does each of Lupe's optimizations contribute to total performance? (Section 5.4)
- **RQ5:** How do Lupe generated programs improve LEA utilization? (Section 5.5)

5.1 How fast is Lupe in continuous power?

Figure 8 shows inference latencies of all 5 models. On average, Lupe achieves 12.36×, 11.11×, 2.22× speedup over Tails, Lupe-BT and Hawaii respectively. Also, as noted in Table 2, Lupe's speedup comes without loss of accuracy.

Lupe is significantly more efficient than the bottom-up implementations, i.e. Tails and Lupe-BT. Similar to our batched-acceleration,

Model FLOPs	Architecture	Dataset Input Size	Accuracy Exp. / Act.
ResNet3 [17]	Conv $10 \times 3 \times 3 \times 3$ $\mathcal{B}$ (Channels: 10, Stride: 1) $\mathcal{B}$ (Channels: 20, Stride: 2)	CIFAR10 [31]	80.42%
5.256 M	$\mathcal{B}$ (Channels: 40, Stride: 2) FC 10×40	$3 \times 32 \times 32$	80.48%
DS-CNN [4]	Conv $64 \times 1 \times 3 \times 3$ (Stride: 2) $\mathcal{DS}$ (Channels: 64, Stride: 1) $\mathcal{DS}$ (Channels: 64, Stride: 1)	SC [63]	94.39%
3.049 M	$\mathcal{DS}$ (Channels: 64, Stride: 1) $\mathcal{DS}$ (Channels: 64, Stride: 1) FC 11×64	$1 \times 49 \times 12$	95.18%
MobileNetV2 [58]	$\mathcal{IR}$ (Channels: 16, Stride: 2) $\mathcal{IR}$ (Channels: 24, Stride: 2) $\mathcal{IR}$ (Channels: 32, Stride: 2) $\mathcal{IR}$ (Channels: 64, Stride: 2)	VWW [11]	80.69%
1.115 M	$\mathcal{IR}$ (Channels: 64, Stride: 1) $\mathcal{IR}$ (Channels: 96, Stride: 1) Conv $24 \times 160 \times 1 \times 1$ FC 2×160	$3 \times 80 \times 80$	79.56%
LeNet [34]	Conv $6 \times 1 \times 5 \times 5$ Conv $16 \times 6 \times 5 \times 5$ Conv $120 \times 16 \times 5 \times 5$	MNIST [6]	98.49%
0.417 M	FC 84×120 FC 10×84	$1 \times 28 \times 28$	98.83%
MLPClassifier	Conv $6 \times 1 \times 5 \times 5$ FC 200×216 FC 120×200	F-MNIST [66]	90.57%
0.165 M	FC 84×120 FC 10×84	$1 \times 28 \times 28$	90.55%

Table 2: DNN model details in our benchmarks. $\mathcal{B}$ denotes a basic block in ResNet, and $\mathcal{DS}$ is a block that has a depthwise convolution followed by a pointwise convolution. The depthwise convolution has padding size of 1. $\mathcal{IR}$ is an inverted residual block in MobileNetV2. We use 0.25 as the width multiplier for MobileNetV2. Additionally, the expansion factor is set to be 6 except for the first block, which is set to 1. We also include the test set accuracy from PyTorch (Exp.) and from Lupe generated programs (Act.). Although computing in fixed points on an MSP430FR5994, our optimizations do not introduce any accuracy decrement, except for MobileNetV2 that we have about 1% accuracy loss.

Hawaii also tries to utilize the SRAM as much as possible, which brings them a tremendous speedup in DS-CNN. Moreover, Hawaii partially restructures programs for cheaper logging cost in intermittent computing. Hawaii also optimizes the DMA calls in the same way as Lupe does. However, although extra performance benefits are introduced, Hawaii fails to recognize the possibility of systematically reorganizing the program to reduce LEA preparation overhead (as discussed in Section 3.1.1). Additionally, Hawaii does not implement DNN layers adaptively, contrary to our optimization in Section 3.2.1.

Overall, these results show that Lupe's top-down approach achieves large performance improvements by enabling optimizations that are not possible otherwise, especially through bottom-up methods.

5.2 How fast is Lupe in intermittent power?

Figure. 9 shows latency per inference, including device initialization time, under different reboot rates. During each reboot cycle, we uniformly pick a random reboot time in the given time intervals. As shown in the figure, the total execution time tends to be stable

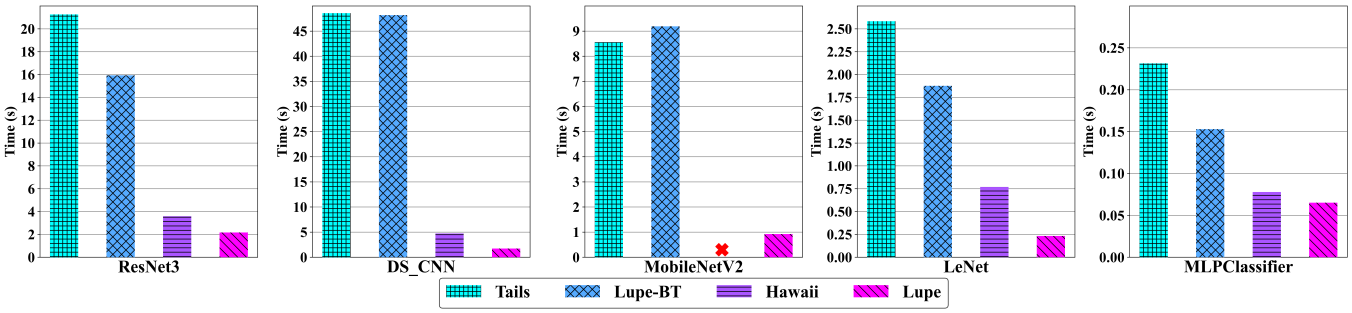


Figure 8: Latencies of the four studied methods in continuous power. MobileNetV2 does not fit in FRAM for Hawaii.

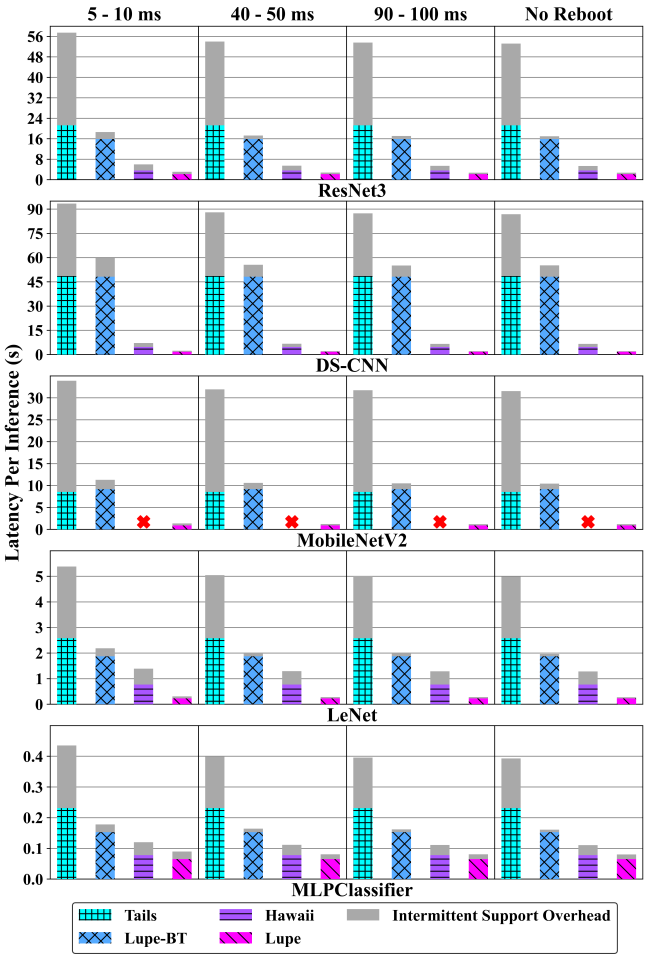


Figure 9: We benchmark all approaches under different reboot rates. We also show the overhead of adding intermittent-safe support to each method. Tails and Lupe-BT have comparable latencies under continuous environments, yet adding intermittent runtime in a top-down way for Lupe-BT leads to much lower overhead for logging information. Hawaii does not support MobileNetV2.

regardless of the reboot rate. Even in the worst case, where Tails restarts 11740.64 times per inference for DS-CNN, it only adds 7.5% execution time.

For the total execution time, Lupe achieves 21.65×, 9.91×, 2.75× average speedup over Tails, Lupe-BT and Hawaii respectively. These higher speedups comparing to Tails and Hawaii show that Lupe provides a much more lightweight intermittent-safe support.

Furthermore, we analyze the overhead of adding such support by highlighting the differences between continuous and intermittent-safe versions. The intermittent runtime of Lupe reduces the average intermittent-safe overhead by 96.65%, 47.49%, 71.15% over Tails, Lupe-BT and Hawaii respectively.

We augment Lupe-BT's bottom-up continuous implementation with a lightweight top-down intermittent-safe support. This results an extremely cheap logging overhead. Particularly, on average, Lupe-BT reduces intermittent-safe overhead by 91.05% over Tails.

Overall, these results show that Lupe maintains its speed advantages under intermittent power. In addition, the comparison of Tails to Lupe-BT highlights the efficiency of Lupe's top-down application of loop continuation compared to a bottom-up approach.

5.3 How much energy can Lupe save?

Figure 10 shows the energy consumption of a single DNN inference for both continuous and intermittent-safe implementations, benchmarked using TI's energy trace software[14]. On average, Lupe reduces energy consumption by 88.06%, 83.9%, 54.13% in continuous conditions over Tails, Lupe-BT and Hawaii, respectively. Additionally, we measure the energy consumption of intermittent-safe programs with no reboots. Lupe provides 92.3%, 79.44%, 56.92% average reduction over Tails, Lupe-BT and Hawaii, respectively.

Additionally, Figure 11 shows power snapshots of executing ResNet3 for continuous and intermittent-safe implementations. We use continuous power and let the MSP430 spin for 2 seconds between each inference to differentiate compute and idle time. Lupe has similar peak power as prior works, but because Lupe dramatically improves LEA utilization—as motivated in Section 2.2 and evaluated in Subsection 5.5—it achieves energy efficiency gains from latency reductions in both continuous and intermittent-safe implementations.

Overall, these results show that Lupe achieves not just significantly lower latency, but that it has no significant energy overhead.

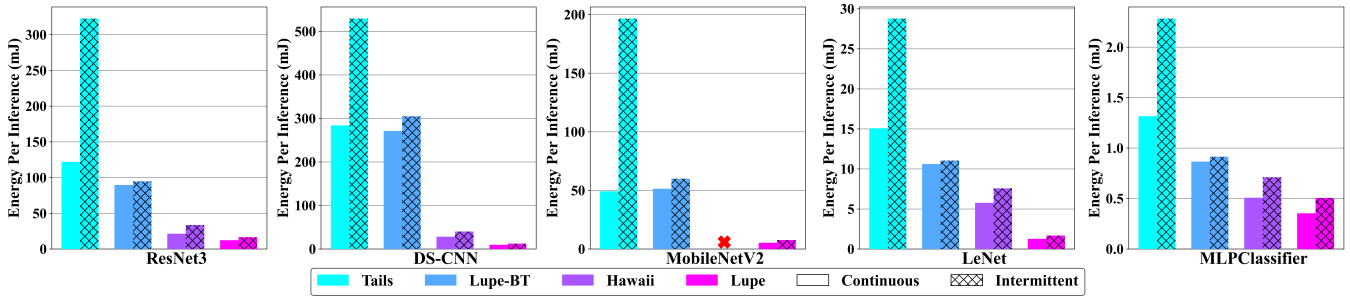
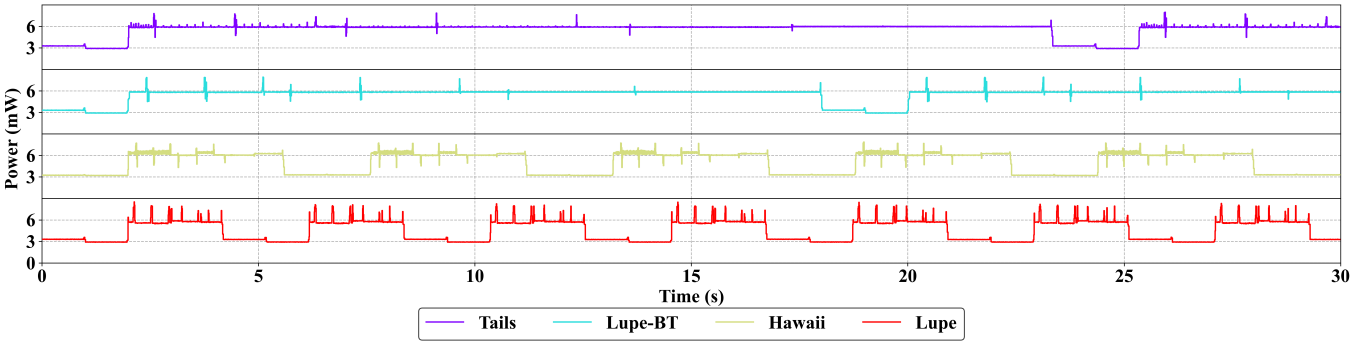
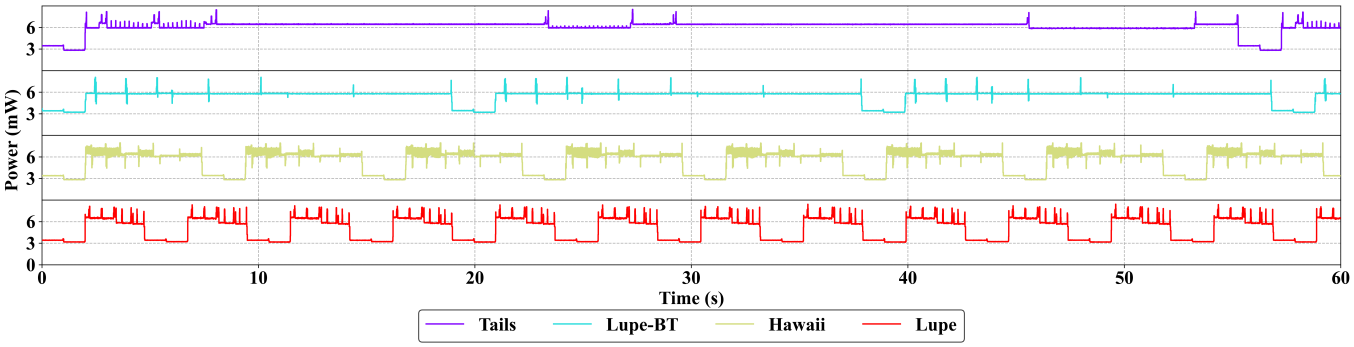


Figure 10: Energy consumption of a single DNN inference for continuous and intermittent-safe implementations. Hawaii does not support MobileNetV2.



(a) Power snapshots of continuous implementations.



(b) Power snapshots of intermittent-safe implementations.

Figure 11: Power snapshots of executing ResNet3 for (a) continuous and (b) intermittent-safe implementations, running in continuous power. CPU spins for 2 seconds between each inference, which consumes around $3mW$.

5.4 How does each of Lupe's optimizations contribute to total performance?

We now explain how each optimization technique improves the overall performance. We use FIR-instantiations for all convolutions unless adaptive layer generation, motivated in Section 2.3, is enabled. Each optimization is added on top of the previous ones. Overall, Lupe provides 11.11× speedup over our non-optimized version (Lupe-BT), as shown in Figure 12.

Section 3.1.1 and Section 3.1.3 demonstrate how to minimize DMA and LEA preparation costs by reorganizing programs. On average, optimizing DMA and LEA function calls improves latency by 1.24× and 2.06×, respectively. Generating data movement methods adaptively is discussed in Section 3.1.3. It only adds 1.04× speedup because DMA operations are already efficient from previous optimizations. Lupe uses batched LEA operations to reduce invocation overhead as described in Section 3.1.2. Changing FIR-instantiations to batched-FIR-instantiations brings 2.12× speedup. Finally, having the possibility of using other instantiations, namely adding the Calibration phase, gives us a speedup of 1.74×. This speedup mainly comes from small input sizes of last few convolution layers.

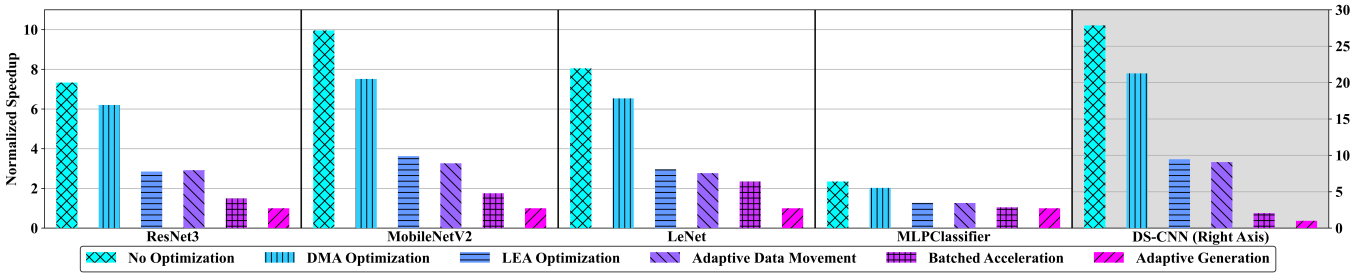


Figure 12: Breakdown of optimizations that contribute to the overall speedup in Lupe, starting with a FIR only instantiation and adding other optimizations step by step. DS-CNN uses a different y2-axis scale.

We achieve a massive 27.85× speedup on DS-CNN, that comes from batched acceleration, improving performance by 4.4×. DS-CNN has large input and output channels for every layer, which provides a perfect execution scenario for batched instantiations.

Overall, these results show that each of Lupe's optimizations described in Section 3 are meaningful to the overall result. Even the adaptive data movement, that has the smallest impact, provides 4% performance improvement on average.

5.5 How do Lupe generated programs improve LEA utilization?

In several places, the paper has argued that the top-down approach will improve LEA utilization, and therefore overall performance. We measure the LEA utilization by comparing LEA execution time of all LEA functions, i.e. time of `msp_lea_invokeCommand` in Figure 5, with the total inference time for one input. In other words, we compute LEA utilization as the percentage of time that the LEA is computing during execution; i.e. the ratio of computation time using the LEA to total time for inferences. We compare the utilization of two instantiations, FIR-instantiations and MAC-instantiations, throughout every DNN layer precluding the Calibration phase. As shown in Figure 13, Lupe greatly improves LEA utilization: on average from 12.70% to 41.44% for FIR-instantiations and from 7.22% to 32.88% for MAC-instantiations.

Overall, these results show that our initial hypothesis is correct: the top-down approach leads to better LEA utilization and better overall performance.

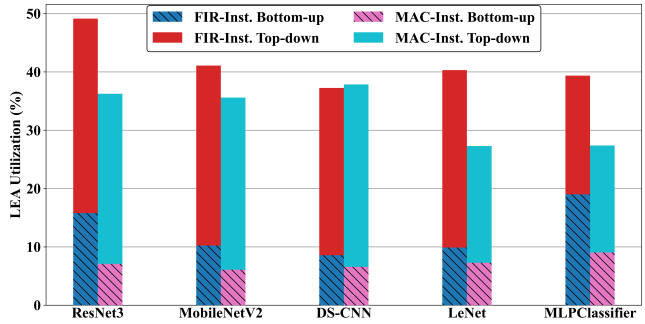


Figure 13: Lupe improves LEA utilization for all models in FIR-instantiations and MAC-instantiations.

6 Related Work

Top-down optimizations: A rich set of prior works have explored top-down optimizations in high power devices, i.e. GPUs, [2, 37, 42, 59, 65, 67, 71]. However, none has explicitly explored the challenges of the top-down optimization on ULP devices, with one exception. Hawaii [27] employs top-down checkpoint insertion for intermittent-safe DNNs on ULP. Lupe is the first work that attempts to tackle the unique challenges of applying a top-down approach to overall DNN optimizations on ULPs. Specifically, programming models and optimization goals differ significantly from a GPU to the LEA. The GPU optimization requires minimizing costs of running parallel computation across different execution units. The LEA, on the other hand, has high costs for SRAM preparation and function invocation within a single execution unit—as discussed in Section 2—and hence requires novel solutions to reduce these costs and increase its utilization.

DNN inferences on ULP devices: To support such efficient inference, several prior works have proposed schemes to deploy DNN models on ULP devices, for MSP430 [5, 7, 13, 16, 24, 27–29, 35, 48, 68, 72] and other hardware [3, 60]. Some use CPU-based implementations [13, 35], while others accelerate DNN programs through on-chip accelerators [3, 5, 7, 16, 24, 28, 29, 48, 60, 68, 72]. However, they all fail to systematically structure programs that utilize on-chip accelerators from a top-down view, and thus suffer from the limitations we analyze in this paper. Particularly, [3, 48] optimize the model architecture itself through quantization and neural architecture search. Lupe, in contrast, does not change the DNN architecture and optimizes it for the LEA by minimizing data movement and adaptively generating DNN layers.

Intermittent DNN systems: Many works support efficient, timely DNN applications in intermittent power. Some focus on DNN architectures [15, 25, 26, 36, 50, 64], while others use advanced scheduling algorithms [8, 22, 23, 47, 49, 57]. Nevertheless, these systems require the underlying DNN inference systems to be efficient.

7 Future Work and Conclusion

Lupe's top-down approach can be applied to other microcontrollers with similar accelerators because it only uses common linear algebra operations. Moreover, other DNN operations, such as self-attention [62], can be added to Lupe using the same top-down methodology. For example, self-attention can be accelerated by matrix-matrix multiplication or MAC functions on the LEA.

This paper introduces Lupe, a code generation framework for DNN inference on ULP devices. Lupe uses a top-down approach to achieve three specific advantages. First, Lupe greatly reduces the data manipulation cost for accelerator use. Second, it adaptively generates the best performing instantiations of DNN layers. Finally, Lupe implements an extremely low cost intermittent-safe computing runtime. For two prior works, Lupe achieves an average speedup of 12.36× and 2.22× in continuous power, reducing the intermittent overhead by 96.65% and 71.15%, respectively.

Acknowledgments

We thank Anwesha Das from the University of Chicago for her valuable advice on this project. We also thank the anonymous reviewers for their feedback and our shepherd for guiding the paper to acceptance. Funding for this work comes from the National Science Foundation (CCF-2119184, CNS-2313190, CCF-1822949 and CNS-1956180).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283.

[2] Ahmed F. AbouElhamayed, Susanne Balle, Deshanand P. Singh, and Mohamed S. Abdelfattah. 2024. Beyond Inference: Performance Analysis of DNN Server Overheads for Computer Vision. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024*, Vivek De (Ed.). ACM, 268:1–268:6.

[3] ADI MAX78000/MAX78002 Model Training and Synthesis [n. d.]. https://github.com/analogdevicesinc/ai8x-synthesis.

[4] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. 2021. MLPerf Tiny Benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (2021).

[5] Rei Barjami, Antonio Miele, and Luca Mottola. 2024. Intermittent Inference: Trading a 1% Accuracy Loss for a 1.9x Throughput Speedup. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems, SenSys 2024, Hangzhou, China, November 4-7, 2024*, Jie Liu, Yuanchao Shu, Jiming Chen, Yuan He, and Rui Tan (Eds.). ACM, 647–660.

[6] Christopher J.C. Burges, Yann LeCun, and Corinna Cortes. [n. d.]. The MNIST Database of Handwritten Digits. ([n. d.]). http://yann.lecun.com/exdb/mnist/

[7] Luca Caronti, Khakim Akhunov, Matteo Nardello, Kasim Sinan Yildirim, and Davide Brunelli. 2023. Fine-grained Hardware Acceleration for Efficient Batteryless Intermittent Inference on the Edge. *ACM Trans. Embed. Comput. Syst.* 22, 5 (2023), 82:1–82:19.

[8] Shu-Ting Cheng, Wen Sheng Lim, Chia-Heng Tu, and Yuan-Hao Chang. 2023. TRAIN: A Reinforcement Learning Based Timing-Aware Neural Inference on Intermittent Systems. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*. IEEE, 1–9.

[9] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler Directed Speculative Intermittent Computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 399–412.

[10] François Chollet. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 1800–1807.

[11] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. 2019. Visual Wake Words Dataset. *CoRR* abs/1906.05721 (2019).

[12] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 514–530.

[13] Rory Conlin, Keith Erickson, Joseph Abbate, and Egemen Kolemen. 2021. Keras2c: A library for converting Keras neural networks to real-time compatible C. *Eng. Appl. Artif. Intell.* 100 (2021), 104182.

[14] ENERGYTRACE: EnergyTrace Technology [n. d.]. https://www.ti.com/tool/ENERGYTRACE.

[15] Pietro Farina, Subrata Biswas, Eren Yildiz, Khakim Akhunov, Saad Ahmed, Bashima Islam, and Kasim Sinan Yildirim. 2024. Memory-efficient Energy-adaptive Inference of Pre-Trained Models on Batteryless Embedded Systems. *CoRR* abs/2405.10426 (2024).

[16] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 199–213.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778.

[18] Josiah D. Hester, Kevin M. Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, SenSys 2017, Delft, Netherlands, November 06-08, 2017*, M. Rasit Eskicioglu (Ed.). ACM, 17:1–17:13.

[19] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 228–240. https://doi.org/10.1145/3079856.3080238

[20] 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. *CoRR* abs/1704.04861 (2017).

[21] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016).

[22] Bashima Islam and Shahriar Nirjon. 2020. Scheduling Computational and Energy Harvesting Tasks in Deadline-Aware Intermittent Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*. IEEE, 95–109.

[23] Bashima Islam and Shahriar Nirjon. 2020. Zygarde: Time-Sensitive On-Device Deep Inference and Adaptation on Intermittently-Powered Systems. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3 (2020), 82:1–82:29.

[24] 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 *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022*, Cristiana Bolchini, Ingrid Verbauwhede, and Elena-Ioana Vatajelu (Eds.). IEEE, 921–926.

[25] Sahidul Islam, Shanglin Zhou, Ran Ran, Yufang Jin, Wujie Wen, Caiwen Ding, and Mimi Xie. 2022. EVE: Environmental Adaptive Neural Network Models for Low-Power Energy Harvesting System. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2022, San Diego, California, USA, 30 October 2022 - 3 November 2022*, Tulika Mitra, Evangeline F. Y. Young, and Jinjun Xiong (Eds.). ACM, 35:1–35:9.

[26] 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 *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services, MobiSys 2023, Helsinki, Finland, June 18-22, 2023*, Petteri Nurmi, Pan Hui, Ardalan Amiri Sani, and Yunxin Liu (Eds.). ACM, 42–55.

[27] 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. Comput. Aided Des. Integr. Circuits Syst.* 39, 11 (2020), 3479–3491.

[28] 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. Embed. Comput. Syst.* 21, 5 (2022), 49:1–49:26.

[29] Tejas Kannan and Henry Hoffmann. 2021. Budget RNNs: Multi-Capacity Neural Networks to Improve In-Sensor Inference Under Energy Budgets. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*. IEEE, 143–156.

[30] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah D. Hester, and Przemyslaw Pawelczak. 2020. Time-sensitive Intermittent Computing Meets Legacy Software. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 85–99.

[31] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. (2009). https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[32] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1935–1944.

[33] Tung D. Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E. Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. 2020. Compiling ONNX Neural Network Models Using MLIR. *CoRR* abs/2008.08272 (2020).

[34] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Handwritten Digit Recognition with a Back-Propagation Network. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, David S. Touretzky (Ed.). Morgan Kaufmann, 396–404.

[35] Seulki Lee and Shahriar Nirjon. 2019. Neuro.ZERO: a zero-energy neural network accelerator for embedded sensing and inference systems. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems, SenSys 2019, New York, NY, USA, November 10-13, 2019*, Raghu K. Ganti, Xiaofan Fred Jiang, Gian Pietro Picco, and Xia Zhou (Eds.). ACM, 138–152.

[36] Chih-Chia Lin, Chia-Yin Liu, Chih-Hsuan Yen, Tei-Wei Kuo, and Pi-Cheng Hsiu. 2023. Intermittent-Aware Neural Network Pruning. In *60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco, CA, USA, July 9-13, 2023*. IEEE, 1–6.

[37] Jun Liu, Nishkam Ravi, Srimat T. Chakradhar, and Mahmut T. Kandemir. 2012. Panacea: towards holistic optimization of MapReduce applications. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, Carol Eidt, Anne M. Holler, Uma Srinivasan, and Saman P. Amarasinghe (Eds.). ACM, 33–43.

[38] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. 2004. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services, MobiSys 2004, Hyatt Harborside, Boston, Massachusetts, USA, June 6-9, 2004*, Guruduth S. Banavar, Willy Zwaenepoel, Doug Terry, and Roy Want (Eds.). ACM / USENIX.

[39] Low-Energy Accelerator (LEA) Frequently Asked Questions (FAQ) [n. d.]. https://www.ti.com/lit/an/slaa720/slaa720.pdf.

[40] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:14.

[41] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 575–585. https://doi.org/10.1145/2737924.2737978

[42] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 881–897.

[43] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 96:1–96:30. https://doi.org/10.1145/3133920

[44] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 129–144.

[45] Kiwan Maeng and Brandon Lucia. 2019. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1101–1116.

[46] Kiwan Maeng and Brandon Lucia. 2020. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1005–1021.

[47] Kiwan Maeng and Brandon Lucia. 2020. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1005–1021.

[48] Hashan Roshantha Mendis, Chih-Kai Kang, and Pi-Cheng Hsiu. 2021. Intermittent-Aware Neural Architecture Search. *ACM Trans. Embed. Comput. Syst.* 20, 5s (2021), 64:1–64:27.

[49] Cyan Subhra Mishra, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. 2021. Origin: Enabling On-Device Intelligence for Human Activity Recognition Using Energy Harvesting Wireless Sensor Networks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*. IEEE, 1414–1419.

[50] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. 2020. ePerceptive: energy reactive embedded intelligence for batteryless sensors. In *SenSys '20: The 18th ACM Conference on Embedded Networked Sensor Systems, Virtual Event, Japan, November 16-19, 2020*, Jin Nakazawa and Polly Huang (Eds.). ACM, 382–394.

[51] MSP Digital Signal Processing library (DSPLib) Library Documentation [n. d.]. https://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/DSPLib/1_30_00_02/exports/html/index.html.

[52] MSP430-GCC-OPENSOURCE: Open Source Compiler for MSP Microcontrollers [n. d.]. https://www.ti.com/tool/MSP430-GCC-OPENSOURCE.

[53] MSP430 microcontrollers [n. d.]. https://www.ti.com/microcontrollers-mcus-processors/msp430-microcontrollers/overview.html.

[54] MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers [n. d.]. https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf.

[55] ONNX: Open Neural Network Exchange [n. d.]. https://onnx.ai.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035.

[57] Emily Ruppel, Milijana Surbatovich, Harsh Desai, Kiwan Maeng, and Brandon Lucia. 2022. An Architectural Charge Management Interface for Energy-Harvesting Systems. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 318–335. https://doi.org/10.1109/MICRO56248.2022.00034

[58] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 4510–4520.

[59] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. 2014. MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs. *Proc. VLDB Endow.* 7, 13 (2014), 1319–1330.

[60] STM32Cube.AI: Free AI model optimizer for STM32 [n. d.]. https://stm32ai.st.com/stm32-cube-ai/.

[61] Joel van der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 17–32.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[63] Pete Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *CoRR* abs/1804.03209 (2018).

[64] 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 *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 1–6.

[65] Chunwei Xia, Jiacheng Zhao, Qianqi Sun, Zheng Wang, Yuan Wen, Teng Yu, Xiaobing Feng, and Huimin Cui. 2024. Optimizing Deep Learning Inference via Global Analysis and Tensor Expressions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024-1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 286–301.

[66] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017).

[67] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (2018), 446–460.

[68] Chih-Hsuan Yen, Hashan Roshantha Mendis, Tei-Wei Kuo, and Pi-Cheng Hsiu. 2022. Stateful Neural Networks for Intermittent Systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41, 11 (2022), 4229–4240.

[69] Kasim Sinan Yildirim, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah D. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems, SenSys 2018, Shenzhen, China, November 4-7, 2018*, Gowri Sankar Ramachandran and Bhaskar Krishnamachari (Eds.). ACM, 41–53.

[70] Eren Yildiz, Lijun Chen, and Kasim Sinan Yildirim. 2022. Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 339–355.

[71] Cody Hao Yu, Haozheng Fan, Guangtai Huang, Zhen Jia, Yizhi Liu, Jie Wang, Zach Zheng, Yuan Zhou, Haichen Shen, Junru Shao, Mu Li, and Yida Wang. 2023. RAF: Holistic Compilation for Deep Learning Model Training. *CoRR* abs/2303.04759 (2023).

[72] Le Zhang, Yubo Luo, and Shahriar Nirjon. 2022. Demo Abstract: Capuchin: A Neural Network Model Generator for 16-bit Microcontrollers. In *21st ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN 2022, Milano, Italy, May 4-6, 2022*. IEEE, 497–498.

[73] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 6848–6856.