Stateful Neural Networks for Intermittent Systems

Chih-Hsuan Yen, *Graduate Student Member, IEEE,* Hashan Roshantha Mendis, *Member, IEEE,*
Tei-Wei Kuo, *Fellow, IEEE* and Pi-Cheng Hsiu, *Senior Member, IEEE*

***Abstract*—Deep neural network (DNN) inference on intermittently-powered battery-less devices has the potential to unlock new possibilities for sustainable and intelligent edge applications. Existing intermittent inference approaches preserve progress information separate from the computed output features during inference. However, we observe that even in highly specialized approaches, the additional overhead incurred for inference progress preservation still accounts for a significant portion of the inference latency.**

This work proposes the concept of stateful neural networks, which enables a DNN to indicate the inference progress itself. Our runtime middleware embeds state information into the DNN such that the computed and preserved output features intrinsically contain progress indicators, avoiding the need to preserve them separately. The specific position and representation of the embedded states jointly ensure both output features and states are not corrupted while maintaining model accuracy, and the embedded states allow the latest output feature to be determined, enabling correct inference recovery upon power resumption. Evaluations were conducted on different Texas Instruments devices under varied intermittent power strengths and network models. Compared to the state of the art, our approach can speed up intermittent inference by 1.3 to 5 times, achieving higher performance when executing modern convolutional networks with weaker power.

***Index Terms*—Deep neural networks, intermittent systems, stateful neural networks, energy harvesting, battery-less devices**

I. Introduction

Deep neural networks (DNNs) are widely deployed on edge devices, where edge DNN inference offers several major advantages, including efficient communication bandwidth usage, improved application responsiveness and better user data privacy. Energy harvesting has increasingly replaced the use of batteries, powering edge devices via ambient energy for maintenance-free operation and extended operational lifetimes at low cost [1]. However, ambient energy (e.g., solar, thermal, wireless or vibration) is inherently unstable and weak. As a result, battery-less devices experience frequent power failures and execute applications *intermittently* [2], [3], where the rate of intermittency increases inversely with the strength of the power source. This has sparked innovative applications such as RF-powered cellphones [4] and cameras [5], as well as game consoles powered by solar and mechanical energy [6]. Several previous works have enabled DNN inference to be executed on intermittent systems [7]–[11], but their execution performance is hindered by the additional overhead incurred to accumulate the inference progress across power cycles. Therefore, *intermittent DNN inference* is challenging, particularly for battery-less devices to achieve an inference speed comparable to that on battery-powered devices.

Edge inference on resource-constrained devices has been made possible by model compression techniques [12], [13], efficient network designs [14], [15], neural architecture search methods [16]–[18] and optimized inference execution approaches [19]. They aim to reduce the size of a DNN model, as well as its memory and computational requirements, while minimizing model accuracy loss. Modern edge devices may also contain on-chip hardware accelerators [20]–[23] or special CPU instructions [24] to efficiently perform inference computations. However, even the most highly optimized and heavily accelerated inference process may consume more energy than a typical lightweight energy harvester can provide. Therefore, in most cases, a single end-to-end inference would still require multiple power cycles to complete, when executed on an intermittent system.

Intermittent inference has been made possible by directly adopting *general* intermittent execution approaches [7], [10], [11], [25], [26]. For example, *checkpointing-based* execution [10], [25], [27], suspends the intermittent system during inference to back up volatile data into non-volatile memory (NVM). However, DNN inference requires large checkpoints due to its high memory usage, incurring a high runtime overhead, and therefore recent attempts such as FLEX [10] strive to reduce the checkpoint size. Alternatively, *task-based* execution requires the application to be decomposed into a chain of tasks, each atomically executed within the available energy in a power cycle, with task indices and modified data preserved in NVM, after each task completes. DNN inference comprises loop-heavy computations, and for task-based intermittent inference, like in SONIC/TAILS [7] and iNAS [11], multiple loop indices need to be preserved to indicate the inference progress [7], [11], along with the computed operation output, thus hindering the inference progress. The inference latency is further degraded as interrupted tasks are re-executed upon power resumption, where the re-execution cost scales with the task size. In short, the overhead of general intermittent

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2022 and appeared as part of the ESWEEK-TCAD special issue. This work was supported in part by the Ministry of Science and Technology, Taiwan, under Grant MOST 110-2222-E-001-003-MY3. (Corresponding author: Pi-Cheng Hsiu.)

Chih-Hsuan Yen is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 10617, Taiwan, and also with the Research Center for Information Technology Innovation (CITI), Academia Sinica, Taipei 11529, Taiwan (e-mail: d06922023@csie.ntu.edu.tw).

Hashan Roshantha Mendis is with the Research Center for Information Technology Innovation (CITI), Academia Sinica, Taipei 11529, Taiwan. (e-mail: rosh.mendis@citi.sinica.edu.tw).

Tei-Wei Kuo is with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, and also with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 10617, Taiwan (Email: ktw@csie.ntu.edu.tw).

Pi-Cheng Hsiu is with the Research Center for Information Technology Innovation (CITI), Academia Sinica, Taipei 11529, Taiwan, also with the Department of Computer Science and Engineering, National Chi Nan University, Nantou 54561, Taiwan, and also with the College of Electrical Engineering and Computer Science, National Taiwan University, Taipei 10617, Taiwan. (e-mail: pchsiu@citi.sinica.edu.tw).

execution would dominate the inference latency if more data transfers are incurred to preserve progress indicators than to preserve computation outputs, making it particularly difficult to make forward progress when the available energy is scarce.

Recently, approaches *specifically* designed for intermittent inference have been proposed, such as *footprint-based* inference [8], [9], aimed at reducing the runtime overhead of intermittent inference. These approaches preserve each intermediate computation output (referred to as a *job* output) paired with a footprint (i.e., a progress indicator) during inference, and use the latest footprint to resume the interrupted job, upon power resumption. This allows accelerated inference operations to be executed *accumulatively*, where an operation may consist of multiple atomic sub-operations (i.e., jobs). Therefore, footprint-based approaches enable job outputs to be preserved in *parallel* to job computations and require only the re-execution of the interrupted job, not the entire task [7]. Initially, in HAWAII [9], progress indicators were preserved *independently* of the job outputs, incurring a high progress preservation overhead, which was considerably reduced in the most recent attempt, JAPARI [8], by augmenting the DNN model to allow progress indicators to be preserved *simultaneously* with job outputs. Despite these efforts, additional computation and data transfer overheads are still incurred to *separately* generate and preserve progress indicators, limiting the latency improvement achievable by footprint-based intermittent inference.

This work proposes the concept of *stateful neural networks* to enable a DNN to inherently contain the inference progress. Although progress indicators are fundamentally essential for intermittent inference, we observed that even in the state of the art, the additional overhead incurred to generate and preserve progress indicators still accounts for a significant portion of the inference latency. Therefore, a substantial latency improvement can be achieved by *eliminating* the progress preservation overhead, motivating the need for a DNN that can indicate its inference progress itself. A stateful DNN embeds inference state information within the DNN model, such that the computed job outputs *intrinsically* contain the progress indicators, avoiding the need to separately preserve them with additional data transfers. Upon power resumption, the state information in the preserved job outputs allows inference to be correctly resumed.

To realize our concept, we develop a footprint-based middleware stack, which addresses two key design challenges. The first challenge is to determine *where* to embed the inference states, without interfering with the original inference computations, thus maintaining model accuracy. This challenge is addressed by *state information embedding*, where states are embedded into specific network components and represented appropriately, to jointly ensure the preserved job outputs are uncorrupted and contain the required states. The second challenge is to determine *what* states to embed, to make it possible to find the last preserved job output in NVM, thereby allowing for correct inference recovery. This challenge is addressed by *state information assignment*, which ensures the specific states assigned to the job outputs allow the latest job output to be uniquely distinguishable from others preserved in the same NVM region.

We prototyped and evaluated our stateful DNN design on the Texas Instruments (TI) MSP-EXP430FR5994 [28] and MSP-EXP432P401R [29], where the latter is more computationally efficient than the former. Experiments were conducted under different power strengths and on three DNN models with varied levels of complexity, representative of those typically used in tiny machine learning applications [30]. Compared to recent intermittent inference approaches, HAWAII [9] and JAPARI [8], our approach reduces the progress preservation overhead between 14% and 83% of the inference latency down to between 1% and 21%, thereby achieving an inference speed-up of 1.3 to 5 times, while maintaining the model accuracy. The improvements are more obvious for modern convolutional network architectures with highly intermittent power supplies. We also conduct a breakdown analysis on the runtime overhead of our approach to identify areas for further study.

The remainder of this paper is organized as follows. Section II provides background information, and Section III explains the motivation for this work. Section IV presents our proposed stateful DNN design with implementation details given in Section V. Experimental results are reported in Section VI with limitations and future improvements discussed in Section VII. Section VIII presents some concluding remarks.

II. Background

A. Embedded Deep Inference

A DNN consists of multiple sequentially processed layers with loop-heavy computations. Each layer performs linear or non-linear operations on *input feature maps* (IFMs) to produce *output feature maps* (OFMs), which are fed into the following layers in the network [31]. The IFM and OFM dimensions of a layer may differ based on factors such as the dimensions of the preceding layers and the characteristics of the filters of the current layer. Non-linear layers, such as activation or pooling layers directly process the IFMs to generate OFMs. Linear layers, such as convolution and fully-connected layers, multiply and accumulate (MAC) filter weights with IFMs and add biases to produce partial sums, which are subsequently accumulated with previously computed partial sums in the OFMs. Here, the weights and biases of the filters form the learnable model parameters. Typically, the model parameters and the first layer's IFMs are quantized into a low-precision fixed-point format, to allow for deployment on resource-constrained embedded devices.

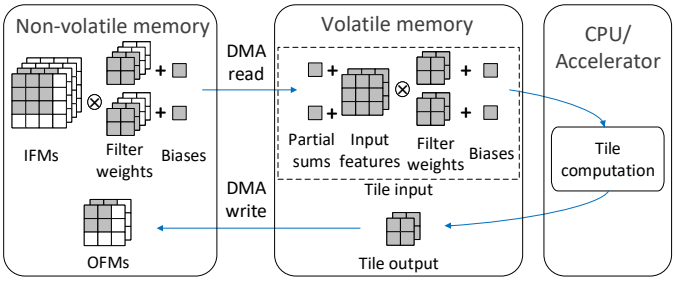


Fig. 1. Inference flow of a DNN layer on a tiled basis.

Due to the limited VM capacity of embedded devices, IFMs, OFMs and model parameters are stored in NVM and are

processed as *tiles* in VM [11], [32], where VM has lower data access energy and latency costs than NVM. As shown in Fig. 1, to compute an OFM tile, the components related to the *tile input*, specifically the required input features, filter weights and biases as well as the previously computed partial sums of the output features are fetched from NVM to VM. The computed *tile output* (i.e., accumulated partial sums or fully completed output features) is written back from VM to NVM. To reduce data access costs between NVM and VM, the inference system usually *reuses* the previously fetched tile input data when processing adjacent tiles in a layer [11], [16].

DNN inference typically requires an *NVM buffer* to store the IFMs and OFMs of the executed layers. As one layer's OFMs form the IFMs of the subsequent connected layers, the same NVM buffer region may be reused for different layers in the network, resulting in the buffer data being overwritten. For performance considerations, a direct memory access (DMA) controller is used for data transfers between NVM and VM, and a hardware accelerator [20]–[22], if available on the MCU, is used to compute the MAC operations of linear layers. An accelerator operation generally contains multiple *sub-operations*, each of which produces the minimum intermediate output of an accelerator, and during the operation execution, an on-chip accelerator would place each *sub-operation output* in a VM region shared with the CPU. Here, a sub-operation output may represent a partially accumulated or fully completed output feature. Non-linear layers are generally computed in software, by the CPU because hardware accelerators on lightweight systems may not support non-linear operations.

B. Intermittent Deep Inference

Intermittently-powered inference differs significantly from inference under continuous power. An intermittent system harvests and buffers ambient energy into an energy buffer (e.g., a capacitor), and the system is powered on when the buffered energy level reaches a preset threshold and powered off when the energy buffer is depleted [33]. The system power on duration, which is typically in the order of tens or hundreds of milliseconds, and the frequency of power failures depends on the strength of the power source, buffered energy and energy consumption in a power cycle [2], [3], [34]. Power loss interrupts execution, and all volatile data in the system are lost, including data in the VM, the peripheral state and the inference progress. Hence, intermittent systems use NVM to accumulate inference progress across power cycles. Upon power resumption, the interrupted inference process is resumed, unlike in continuously-powered systems that restart the process from the outset, which may lead to non-termination.

To accumulate inference progress across power cycles, intermittent inference approaches perform *progress preservation* during inference, where *progress indicators* are generated and preserved along with the computed operation outputs from VM to NVM. Upon power resumption, *progress recovery* is performed where, after system reboot, the preserved progress indicators are used to re-fetch the required input data and correctly *resume* the interrupted inference process. In addition, operations related to unpreserved progress in VM, which were lost in the previous power cycle, are *re-executed* in the next power cycle. Existing intermittent inference approaches may differ in the preservation granularity and what progress indicators they preserve, which directly affects the progress preservation overhead and the amount of re-execution performed upon power resumption.

Initial efforts to realize intermittent inference (e.g., SONIC/TAILS [7]) used task-based execution, developed for *general* intermittent applications. Here, a DNN model is partitioned and executed as a set of sequential, atomic tasks, each consisting of one or more accelerator operations, and computation loop indices are tracked as progress indicators directly in NVM. However, this leads to a high progress preservation overhead as multiple variables have to be updated in NVM and a large re-execution overhead as the entire interrupted task (i.e., operation) has to be re-executed upon power resumption. Recently, footprint-based approaches such as HAWAII [9] and JAPARI [8] were proposed *specifically* for intermittent inference. They generate footprints (i.e., progress indicators) during operation execution, where each is paired with a sub-operation (i.e., a *job*) and preserved during inference. Upon power resumption, the last preserved footprint is used to resume the interrupted operation. Such fine-grained progress preservation allows for accumulative accelerator operations, which is necessary under frequent power failures.

In HAWAII [9], job outputs and progress indicators are *independently* generated and preserved. Accordingly, job computation, job output preservation and progress indicator preservation can be pipelined and performed in parallel, thus reducing the runtime overhead. However, HAWAII requires an individual data transfer command to preserve each progress indicator. By contrast, JAPARI augments the DNN with additional network elements, enabling the accelerator to alternately produce job outputs and progress indicators in VM. This allows all job output and progress indicator pairs to be preserved *simultaneously* with only a single data transfer command, greatly reducing the data transfer overhead of progress preservation, but at the cost of extra accelerator computations.

III. Progress Preservation Overhead: Observations and Motivation

Progress preservation is essential for accumulative inference execution across power cycles, but may significantly degrade inference latency. We conduct an experiment in which we measure the runtime overheads introduced by two recent approaches, HAWAII [9] and JAPARI [8]. These approaches are compared to an *ideal* baseline that computes and preserves job outputs similar to the other two approaches, but the ideal neither generates nor preserves progress indicators, so it cannot accumulate progress across power cycles. This experiment allows us to study the progress preservation overhead incurred by the aforementioned approaches and understand the underlying causes of their overheads.

For our experiment, we use two MCUs from Texas Instruments, namely the MSP430FR5994 [28] and the MSP432P401R [29], respectively referred to hereafter as the MSP430 and MSP432. Unlike the MSP432, the MSP430 has a vector math hardware accelerator, but the MSP432 is computationally superior due to its faster CPU supporting specialized vector math instructions. The MSP430 and MSP432 respectively have 8KB and 64 KB internal VM (SRAM), and a 512KB external NVM (FRAM) module [35] was integrated into each device. The widely used image classifier SqueezeNet

CNN is executed under continuous power, allowing us to isolate the overhead of the approaches under study.

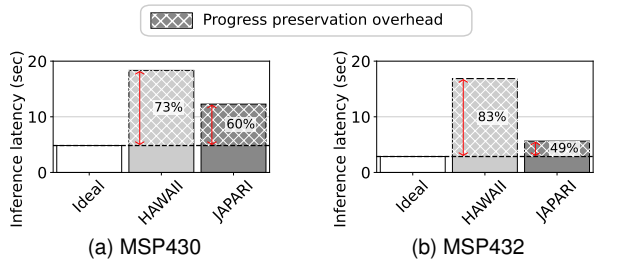


Fig. 2. Progress preservation overhead of HAWAII and JAPARI.

Fig. 2 shows the progress preservation overheads of HAWAII and JAPARI as a proportion of an end-to-end inference latency, obtained by comparing against the ideal baseline. Here, the *progress preservation overhead* includes all extra costs required to accumulate the inference progress across power cycles. HAWAII suffers from excessive progress preservation overhead, respectively accounting for 73% and 83% of the inference latency, on the MSP430 and MSP432. This is because HAWAII requires an individual data transfer command to preserve a progress indicator independent of a job output (Section II-B), leading to high data transfer overhead, particularly on the MSP432, where a data transfer is more costly than a job computation. In comparison, JAPARI considerably reduces the progress preservation overhead, as it requires fewer data transfer commands due to simultaneous preservation of job outputs and progress indicators (Section II-B). However, as JAPARI augments the DNN model and computes the extra network elements to generate the progress indicators, JAPARI suffers from additional computation overhead, which becomes more obvious on the MSP430, where the cost of computation is higher than on the MSP432. Therefore, JAPARI's progress preservation overhead on the two devices still respectively accounts for 60% and 49% of the inference latency.

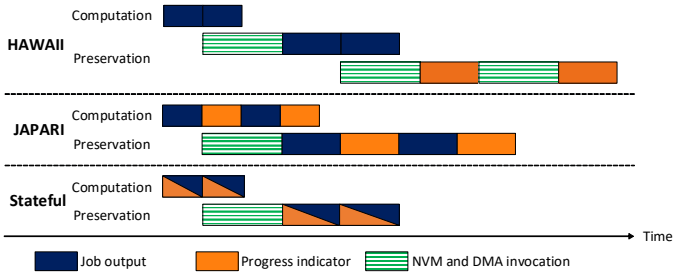


Fig. 3. Illustration of progress preservation across different approaches.

The following implication drawn from our initial investigation inspires this work. As illustrated in Fig. 3, HAWAII incurs additional data transfer overhead due to independent preservation, and although JAPARI can alleviate the data transfer overhead via simultaneous preservation, it incurs extra job computations to generate progress indicators. Regardless of independent or simultaneous progress preservation, both these state-of-the-art approaches still preserve progress indicators *separate* from job outputs. Therefore, progress preservation overhead still accounts for a large portion of the inference latency, and this overhead increases with the gap between the computation and data transfer costs. This observation indicates that the intermittent inference latency can be substantially reduced, which motivates the need for progress indicator preservation to be intrinsically incorporated within job output preservation, as indicated by *Stateful* in Fig. 3.

IV. Stateful Deep Neural Network Inference

A. Design Rationale and Challenges

We propose the concept of a *stateful neural network*, where inference progress information is embedded within the network model, in the form of inference *states*[1]. By making a neural network stateful, we circumvent the need to explicitly track progress indicators separately, thus minimizing the impact of progress preservation overhead on the inference latency. Essentially, to embed states, the values of specific network components are transformed during inference, allowing progress indicator preservation to be *intrinsically* carried out as part of job output preservation. Subsequently, the embedded states are used for correct inference recovery. However, realizing stateful neural networks raises two major design challenges.

The first challenge is determining *where* to embed states in the network model, such that both the embedded states and output features are isolated and not corrupted during inference. This is particularly challenging as embedding the state information at invalid positions in the model may result in the loss of the states after computation, or more seriously, may significantly change the computed output features, compromising model accuracy (Section II-A). The second challenge is determining *what* states to embed, such that the interrupted inference process can be correctly recovered. This is non-trivial as the same NVM region is reused to preserve the job outputs (i.e., partial sums or output features) of different layers in the DNN due to limited NVM space (Section II-A), thus making it difficult to identify the latest preserved job output.

We address the first challenge of where to embed, by *state information embedding* (Section IV-C), which determines the position and representation of the inference states in the DNN model, while ensuring the output features and embedded states are uncorrupted during inference. The key idea is to split the value range of specific network components to represent different states, in such a way that they do not interfere with the output feature computation. This allows intrinsic progress indicator preservation, which eliminates the associated data transfers. We address the second challenge regarding what to embed, by *state information assignment* (Section IV-D), which ensures the specific states assigned and contained within the job outputs allow the latest preserved job output to be uniquely identified from previously preserved job outputs. The key idea is to ensure that the states assigned to the selected network components of the current layer are different from those assigned for previous layers that use the same NVM locations for job output preservation. This allows the interrupted job to be determined, making correct inference recovery possible.

B. System Architecture

We make a given DNN *stateful*, by embedding state information into the network model at runtime. Fig. 4 shows the layered system architecture of a typical lightweight embedded system. The application layer includes the DNN model

[1]Here, *states* are a type of progress indicator, analogous to loop indices used by SONIC/TAILS [7] or layer/job counters used by HAWAII [9].

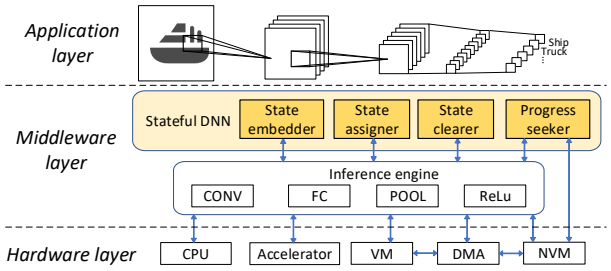


Fig. 4. System overview.

structure and the user program, the middleware layer includes our proposed stateful DNN design that is built on top of a lightweight inference engine, and the hardware layer consists of the CPU, NVM, VM and DMA controller, as well as other on-chip peripherals such as a hardware accelerator, if available on the device. The inference engine can process different types of DNN layers and performs job output preservation in parallel to job computation, similar to that in HAWAII [9] and JAPARI [8]. Our stateful DNN middleware contains four key components: (1) *state embedder*, (2) *state assigner*, (3) *state clearer* and (4) *progress seeker*, which are invoked by the inference engine during inference, to make a DNN stateful. We explain the role of each component below.

The inference engine executes layer by layer, where each layer is processed as tiles. To compute a tile, the inference engine first fetches the tile input from NVM to VM (Section II-A), then the tile output is computed using the CPU or the accelerator. When processing a tile, the inference engine invokes the state embedder to determine where in the tile input or output to embed inference states, and the specific state to embed is determined by the state assigner. Subsequently, the state embedder transforms the values of the selected network components to represent the required states. The inference engine preserves the computed job outputs into NVM, each containing an inference state. However, the inference states of the previous layer's job outputs need to be first cleared to derive the original output features, allowing them to be used as valid input features for the current layer and avoiding corruption to the tile computations. Therefore, from the second layer onwards, immediately after the input features are fetched into VM, the inference engine invokes the state clearer to remove the inference states contained in the input features.

Layer processing may be interrupted by a power failure. Upon power resumption, the inference engine invokes the progress seeker to search for the last preserved job output in NVM. The assigned states in the preserved job outputs enable the latest job output to be distinctly identified, allowing the interrupted job in the current layer to be derived. Next, the tile input related to the remaining jobs of the interrupted tile are re-fetched and the system is re-configured to resume the inference process from the interrupted job.

Our stateful DNN middleware currently supports the most common linear (e.g., CONV and FC) and non-linear (e.g., ReLu and POOL) layers. Our middleware also supports networks with *multiple paths* (i.e., skip connections) typically used in modern network architectures to improve their model accuracy [14], [36]. Section IV-F further discusses how our approach can be easily extended to support different types of neural networks and hardware devices.

C. State Information Embedding

The specific position and representation of the state information embedded in the network model jointly ensure the job outputs preserved into NVM contain the required inference states without corrupting the output features. We first discuss the selection of the network components into which states will be embedded, where they differ based on the type of layer, and subsequently we discuss how the states are represented within the selected network components. In additional to state correctness, factors such as the runtime overhead and sensitivity to model accuracy degradation are considered when we determine both state position and representation.

1) State Position: State information can be embedded into the network components related to the tile input or the tile output. In the case of *tile output*, the state needs to be directly embedded into each job output of each tile in a layer, incurring a higher runtime overhead. By contrast, embedding states into the *tile input* is preferable, because the process of state information embedding need not to be individually performed for each tile, as the tile input data are typically *reused* across multiple tiles in a layer (Section II-A). Therefore, the states embedded into a tile's input also become intrinsically embedded into the input of subsequent tiles that reuse the same tile input data.

Within the tile input, states can be embedded into the input features, filter weights, partial sums or biases, to allow the computed job outputs to contain the required states. As weights, input features and partial sums are typically multiplied and accumulated during inference, they may be more sensitive to accuracy degradation. By contrast, embedding states into the biases is preferable because they are added to a computed tile output. In addition, biases have higher data reuse and there are fewer biases compared to the other tile input components. Thus, embedding states into the tile input, specifically into the biases, minimizes the number of times the process of state information embedding is carried out, thus reducing the runtime overhead.

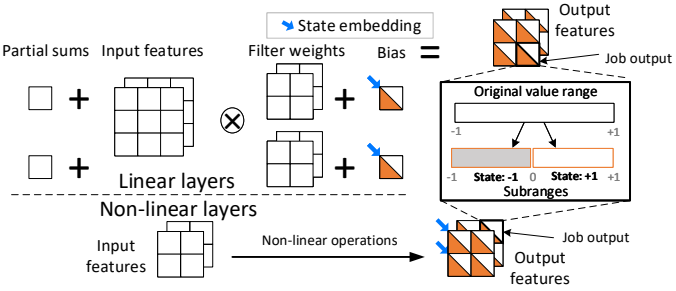


Fig. 5. State embedding position based on layer types.

The selection of the specific network component for embedding state information differs for linear and non-linear layers, as shown in Fig. 5. For *non-linear* layers, the only option is to embed the states into the tile output, as embedding the states into the tile input would result in a loss of state during computation. For example, non-linear operations such as ReLu and POOL may set certain input features to zero depending on their values, or discard them due to downsampling [31]. Note that although less efficient, directly embedding a state into each computed job output, may minimally impact the inference latency, non-linear layers typically account for only a small

portion of inference computations and generate a minority of the job outputs in a network. For *linear layers*, states can be embedded into either the tile input or output, but we chose to embed states into the biases of the tile input, because linear layers dominate the inference computations. To ensure a bias is present for each tile computation, the original bias values are equally divided across all tiles of a layer. The control flow of the state embedder is illustrated in Fig. 6.

2) State Representation: We embed state information into the values of the selected network components. Representing state information *explicitly* using certain bits (either unused or freed up via model compression) raises several issues. On off-the-shelf MCUs with general purpose accelerators, the state bits used in an accelerator operand may be corrupted or lost in the accelerator output, as those bits are also subject to change during computation if not isolated in the operand. If states and network component values are separated into different operands, they may be multiplied during convolution and interfere with each other, as the accelerator does not distinguish between the two types of operands. Thus, explicit state representation constrains a state to be embedded solely into every job output in the network after tile computation, increasing the runtime overhead. Instead, as shown in Fig. 5, we *implicitly* represent inference states by splitting and scaling down the original value range of the selected network components, into multiple smaller subranges, each representing a different state. However, the number of subranges has to be minimized, to reduce the impact on model accuracy.

Taking the example of a network model that has been quantized to the Q15.1 fixed-point number format during deployment, we split its original value range of [-1, +1) into *two*, non-overlapping *subranges*, namely, [-1, 0) and [0, +1) to respectively represent the states -1 and +1. As two subranges are used, the precision of the selected network components is reduced by only one bit. Accordingly, state embedding for non-linear layers involves dividing the original value of a job output by 2 to scale down to the [-0.5, +0.5) range, and then moving the scaled down value to the required subrange by adding or subtracting 0.5, immediately before the job outputs are preserved into NVM. For linear layers, state embedding involves scaling down and moving the values of the biases into the required subrange, as well as scaling down the values of the input features to the [-0.5, +0.5) range. This ensures the product of the input features and weights is at the same scale as the biases, so that the computed output features, which are in the subrange representing a state, can be transformed back to the original range, by simply adding or subtracting 0.5. As the OFMs of one layer become the IFMs of the following layers, from the second layer onwards, the states contained in the input features need to be *cleared* before tile computation, allowing the input features corresponding to the original output features of the previous layer to be obtained. The corresponding control flow of the state clearer is illustrated in Fig. 6. To clear the states, the values of the input features are moved back to the [-0.5, +0.5) range from the subrange and scaled up to the original range.

During tile computation, the multiply and accumulate operations may result in the values of certain job outputs being large and *overflowing* into a different value range, thereby corrupting

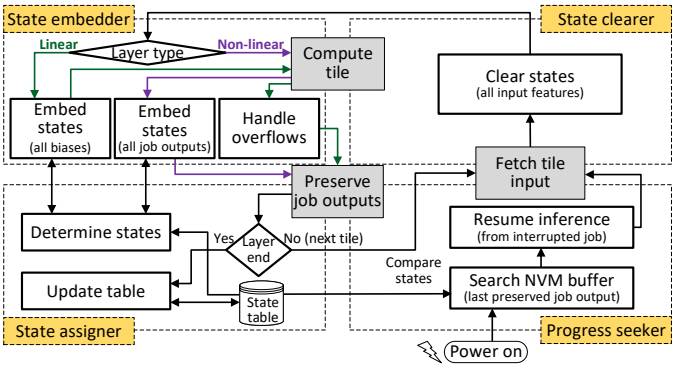


Fig. 6. Stateful control flow of key components.

the states that were embedded prior to tile computation. To resolve this overflow issue, prior to job output preservation, we ensure the sign bit of the computed job output is correctly set to allow the value to be in the required subrange, where for states -1 and +1, the sign bit of a job output should respectively be 1 and 0. Note that we cannot directly change the sign bit of a value to embed state information, as this may corrupt the original values of a large number of selected network components, severely degrading model accuracy. In contrast, changing the sign bit to handle overflows does not significantly impact accuracy because an inference normally encounters only a few overflows. Scaling down the values during state embedding may slightly increase the number of overflows, particularly for values on the two boundaries of the original value range [37], but a highly accurate DNN model would have only a few values on those boundaries [13].

D. State Information Assignment

The specific state information assigned during inference, ensures the latest job output preserved into NVM can be distinctly identified, allowing for correct progress recovery. Recall that each job output contains an inference state, either embedded directly or via the biases, depending on the layer type (Section IV-C). As the NVM buffer is reused during inference (Section II-A), job output preservation may overwrite previously preserved job outputs in the same NVM locations. Therefore, the state assigner ensures the states of the current and overwritten job outputs are different.

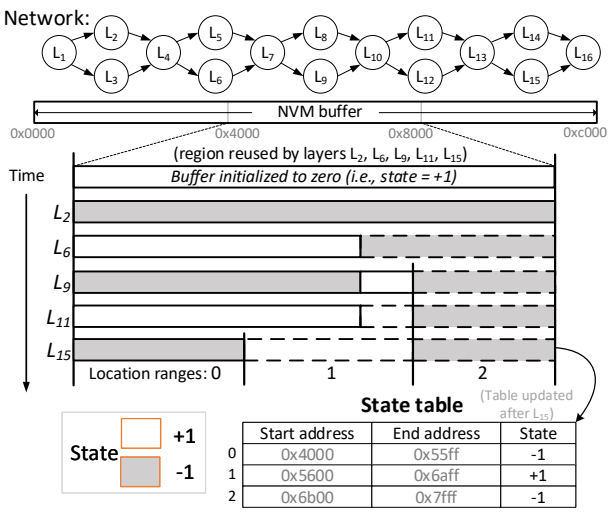


Fig. 7. State information assignment using a state table.

1) State Determination: To determine the inference states assigned to the job outputs of the current tile, the state assigner uses a *state table* maintained in NVM, as shown in Fig. 6. The state table is a custom data structure used to track the states of the job outputs preserved in the NVM buffer, allowing the state assigner to efficiently determine the state at any specific NVM location without directly scanning the NVM buffer. As shown in Fig. 7, each entry in the state table represents a location range in the NVM buffer that contains consecutive locations with the same state[2]. The first and second table fields respectively indicate the start and end NVM addresses, and the third field indicates the state associated with that location range. To derive the state assignments for the current tile's job outputs, the state assigner first queries the state table to determine the states of the job outputs that will be overwritten, and then inverts those states, to ensure the states assigned to the current tile's job outputs are different. The process of state assignment is performed before and after tile computation respectively for linear and non-linear layers. To improve efficiency, only the table entries related to the current layer are fetched into VM at the start of each power cycle.

The preservation location in NVM of every job output is known by the inference engine, as it manages the NVM buffer allocation for each layer, ensuring the job outputs of the current layer do not overwrite any IFMs required by the current or subsequent layers in the network. As shown in Fig. 7, the NVM buffer is divided into multiple regions, where a region is reused by different layers for job output preservation. Furthermore, the NVM buffer is sufficiently large to contain the IFMs of all preceding layers connected to the current layer and the IFMs required by any subsequent layers, as well as to individually preserve all job outputs of a layer, including partial sums and completed output features. Hence, a specific layer's job output in NVM can only be overwritten by a job output of a different layer.

Accordingly, the state table is updated only at the end of each layer (Fig. 6), where table entries are appropriately split or merged, or the states are inverted to reflect the state assignments made for this completed layer. In detail, entries are split if the states in any two adjacent locations within the same range are different, while two adjacent entries are merged if they have the same state. In the case where two successive layers contain the same number of job outputs and reuse the same NVM buffer locations, the corresponding state table entries will simply be inverted, without being split or merged. For example, given the network in Fig. 7, the table will have two and three entries after the system has respectively completed processing layers L_6 and L_{15}. Note that because the size of a layer (i.e., number of jobs) and the NVM locations for job output preservation are known by the inference engine, the state assigner can update the table without scanning the NVM buffer. Moreover, to improve efficiency, the state table entries related to the current layer are updated in VM and written back into NVM, while protecting idempotency despite frequent power failures (detailed in Section V).

2) State Recovery: Upon power resumption, the progress seeker correctly recovers the interrupted inference process. First, the interrupted layer is identified by its index that is independently tracked in NVM by the inference engine. Then, to determine the index of the interrupted job in the layer, the states of the preserved job outputs in the corresponding NVM region are compared against the entries in the state table (see Fig. 6). Here, the state table is essential as it provides a view of the NVM buffer locations *before* they are overwritten by the current layer's job outputs. If the state of a job output in NVM is different from the state recorded in the state table at the corresponding address range, then that job output is regarded as successfully computed and preserved. Therefore, the NVM buffer is searched to find the first job output whose state *matches* the entry in the state table. The search result is the index of the last preserved job, which is subsequently used to derive the loop indices corresponding to the interrupted tile (similar to HAWAII [9] and JAPARI [8]) using the layer's execution settings known by the inference engine. Lastly, the tile inputs required for the remaining jobs in the current tile are fetched into VM, and the system is configured and initialized to resume the inference execution from the interrupted job.

[2]The NVM buffer is initialized to zero (i.e., representing the state +1) after the stateful DNN is deployed onto the target device.

E. Runtime Overhead

The state embedder takes only a few clock cycles to scale down and move the value of a job output or bias, to scale down an input feature value, or to handle an overflow. Its overhead will be lower on hardware devices with a larger VM, which can accommodate a larger tile size, resulting in more tile input data reuse. The state clearer also takes only a few clock cycles to move the value of an input feature back to the original value range. Section V-B provides further implementation details.

The overhead of the state assigner is related to the update frequency and size of the state table. To reduce the update frequency, the state assigner updates the table only after completing a layer. The size of the state table (i.e., number of entries) is bounded by the product of the number of regions in the NVM buffer and the number of layers that may reuse the same region. However, in practice, the table may only have a couple of dozen entries, as DNNs typically have several consecutive layers with the same number of job outputs [14], [36], as those layers have the same dimensions and parameters. In such cases, the related entries are simply inverted but not split. Furthermore, the state assigner's overhead may be lower on devices that can accommodate a larger NVM buffer, allowing all the job outputs of several network layers to be preserved without being overwritten, thus reducing the state table's update frequency. To minimize the progress recovery overhead, the progress seeker uses binary search to find the latest job output in the NVM buffer. The search latency increases logarithmically with the number of jobs in a layer (typically, a few thousand), and the search is constrained to the NVM buffer region allocated to the current layer. The measured runtime overheads of each key design component is provided in Section VI-E.

F. Generality

1) Support for various neural networks: Our stateful DNN design currently supports the most common DNN layers. Linear layers such as CONV and FC are natively supported by embedding states into the biases (Section IV-C). The depthwise and pointwise variants of CONV layers [15] can

also be made stateful, as they use the same form of MAC operations as CONV. Non-linear layers such as POOL and ReLu are also inherently supported, as the states are directly embedded into the job outputs, unlike in JAPARI [8], where progress indicators are generated as individual values that may become corrupted or lost due to a non-linear operation.

Multiple path networks (i.e., with skip connections) are the architecture of choice in modern DNNs, due to their improved accuracy and training efficiency [14], [36]. Unlike JAPARI [8], our stateful design is compatible with both single and multiple path networks, as the NVM buffer is partitioned into flexible regions to accommodate the OFMs of the current layer and the IFMs of the connected preceding layers (Section IV-D). Our approach can also be extended to support recurrent neural networks used for time series datasets, as the network feedback paths are generally unfolded and sequentially executed as multiple path networks.

2) Support for different hardware devices: Our design can be deployed on widely available off-the-shelf MCUs containing hardware accelerators [21], [38], [39] or special CPU instructions [24] to accelerate inference computations. This is because state embedding does not change the network structure or corrupt its computations, so a stateful DNN can be executed as intended. Also, in contrast to HAWAII/JAPARI [8], [9], the performance of a stateful DNN does not degrade even if the MCU does not have the hardware capability to support parallel job computation and job output preservation.

Our stateful DNN design can also be realized on FPGA-based custom accelerators. On most FPGA-based systems, the CPU controls the I/O to the accelerator, and the CPU and accelerator may share a memory region, so the states can be embedded to the tile input/output as per our design. In contrast, if the system does not contain a shared memory region and the accelerator has direct access to the NVM, then Stateful needs to fetch the tile input/output data to VM, embed the states and write the modified data back to NVM, in parallel to tile computation. However, this would impose an additional overhead. Our design can also be easily extended to support accelerators that are optimized for highly quantized, right-sized networks. In such accelerators, it is common to pack multiple low-bitwidth values corresponding to either weights, biases or input/output features, into a long bit vector for increased parallelism [40]. Therefore, the inference state can be embedded into one of the values that are packed together without significantly degrading accuracy.

V. Stateful DNN Implementation

Our design, as shown in Fig. 4, was realized on the MSP430 [28] and MSP432 [29] devices. The MSP430 is a 16-bit MCU, which includes a Low Energy Accelerator (LEA) [22] and 8KB SRAM (4KB VM shared between the CPU and LEA). In contrast, the MSP432 is a 32-bit MCU with 64KB SRAM, and although the MCU does not contain a hardware accelerator, the CPU (ARM Cortex-M4F) supports single instruction multiple data (SIMD) based instructions [24]. The vendor-provided driver libraries were used to invoke the required accelerated operations. An external 512KB NVM module (Cypress CY15B104Q serial FRAM [35]) is used to deploy reasonably large DNN models. The application layer consists of the DNN model, which is quantized to 16 bits (Q15.1 format). Our stateful DNN middleware was implemented on top of a lightweight inference engine, similar to that found in HAWAII [9] and JAPARI [8].

A. Inference Engine

The inference engine processes each layer as tiles (Section II-A), with the tile size optimized to improve data reuse, while constrained by the VM capacity of the device [11], [16]. Linear layers are implemented as accelerated vector-matrix multiplication and vector addition computations [41], while non-linear layers are executed using the CPU. When the inference engine fetches input features and weights from NVM to VM, they are transformed into a column-wise matrix suitable for vector-matrix multiplication using an *im2col* technique that improves data reuse [41]. For non-linear layers, all job outputs of a tile are preserved together after they are all computed and embedded with states. Differently, for linear layers, job output preservation, job computation and overflow handling are pipelined and performed in parallel, using timers for the required synchronization, thus ensuring any computed job output that has overflowed is corrected before it is preserved.

A job output is 2 bytes on both devices and is preserved into NVM byte by byte. As power may fail during the byte-wise data transfer, the byte containing the sign bit of the job output is preserved last, ensuring an incompletely preserved job output will not be incorrectly identified as the latest. As the NVM buffer is large enough to contain all job outputs of a layer (Section IV-D1), partial sums of a tile input are effectively read from a different NVM location to where the tile's job outputs will be written, thus avoiding write-after-read dependencies to protect idempotence (i.e., ensuring the same outcome despite repeated execution) [3], [33].

B. Stateful DNN Middleware

State embedding involves scaling down specific network component values and moving them into the required subrange (Section IV-C2), which we respectively implement by right shifting the values by 1 bit (i.e., division by 2) and by subtracting or adding 0.5 to the values. To handle value range overflows, we use only a pair of bitwise AND and OR instructions per job output, in order to set the sign bit to represent the required state. During state clearing, the values of the input features need to be moved from the subrange and scaled up to the original range, but we omit the redundant step of scaling up, because the input feature values need to be scaled down again before tile computation.

The state table used for state assignment (Section IV-D1) is implemented as an array of custom data types, maintained in a contiguous NVM region, thus allowing the table to be transferred between NVM and VM using a single DMA command. For efficiency, the state assigner fetches only the table entries related to the NVM buffer region associated with the current layer into VM. To protect idempotence, when writing back the updated state table to NVM, the state assigner implements a double buffering mechanism similar to that in [7], [33], where two state tables are used in an alternating manner. During progress recovery, the progress seeker fetches the job output inspected in each iteration of the binary search into VM and compares the job output's state against that recorded in the state table. Note that the aforementioned key

TABLE I
DNN MODELS USED FOR PERFORMANCE EVALUATION

Model	Layers	# of parameters	# of MACs	# of Jobs
KWS: DNN for speech keyword spotting [42] Dataset: Speech Commands [43] (test set size: 4890)	FC × 4 ReLu × 4	Weights: 79 K Biases: 0.4 K	80 K	L: 1 K NL: 0.4 K
HAR: CNN for human activity detection [44] Dataset: Accelerometer sensor data [45] (test set size: 2947)	CONV × 3 ReLu × 3 POOL × 3 FC × 1	Weights: 14 K Biases: 0.1 K	316 K	L: 23 K NL: 10 K
SQN: Multiple path CNN for image classification [14] Dataset: CIFAR-10 [46] (test set size: 10000)	CONV × 11 ReLu × 11 POOL × 2	Weights: 73 K Biases: 0.7 K	4.1 M	L: 94 K NL: 46 K

L: linear layers, NL: non-linear layers

processes of the stateful middleware are short compared to the inference computations, and they need to be executed atomically (i.e. power uninterruptible), where an interrupted process would be re-executed upon power resumption.

VI. Performance Evaluation

A. Experiment Setup

We evaluate our stateful DNN design on the MSP430 and MSP432 devices as introduced in Section V, with the clock frequencies respectively set at 16 and 48 MHz. Intermittent execution was emulated using a BK Precision 9171B power supply and a TI-BQ25504 energy harvesting and management unit, containing a 100 µF capacitor. Experiments were conducted under continuous power (1.65 W), as well as under intermittent power, with weak and strong power sources (respectively 4 mW and 10 mW), which are representative of solar energy under different environmental conditions [47]. Neither intermittent power source was sufficient to continuously operate the devices, leading to repeated yet unpredictable power failures during runtime. These experimental settings are similar to those in prior work [8], [9].

As shown in TABLE I, our evaluation uses DNN models typically used for TinyML applications [30] and fit within our 512 KB external NVM. Namely, we use a fully connected DNN [42] developed for speech keyword spotting (denoted as KWS), a single path CNN developed for human activity detection (denoted as HAR) [44] and a multiple path CNN (denoted as SQN) similar to SqueezeNet [14], which was developed for image classification. These network models vary in terms of their structures, sizes, numbers of MAC computations and jobs. Each model is repeatedly run for 100 inferences, which is sufficient to mitigate experimental variances while making the experiments reproducible.

We compare our approach (referred to as *Stateful*, for brevity) against two state-of-the-art intermittent inference approaches, namely JAPARI [8] and HAWAII [9]. To rigorously evaluate the extra overhead of Stateful, we also compare against an *ideal* baseline, which computes and preserves job outputs similar to the other approaches but it neither generates nor preserves progress indicators, leaving it unable to accumulate inference progress across power cycles. For fair comparison with Stateful, we adapted the implementations of HAWAII [48] and JAPARI [49] to use the implementation optimizations described in Section V-A. Primarily, these adaptations include the use of accelerated vector-matrix multiplications and related tile input data transformations, the tile size and processing order adapted to fully utilize the VM and maximize data reuse, as well as integration with the external NVM and adding support for multiple path networks. Note that the ideal baseline, HAWAII and Stateful were configured to use the same tile size per layer for a DNN, and a relatively smaller tile size was selected for JAPARI, as it requires additional VM capacity to augment a DNN.

We evaluated Stateful in terms of its *inference accuracy*, *progress preservation overhead* and *inference latency*. First, we report the impact on accuracy due to state embedding by Stateful, compared with the original 16-bit quantized model. Next, we measure the progress preservation overheads of all evaluated approaches, compared with the ideal baseline under continuous power. We then evaluate the inference latency under intermittent power for all evaluated approaches, where we define the inference latency as the average time required to complete one end-to-end inference. Lastly, we conduct a breakdown analysis of the runtime overhead of Stateful, in terms of the cost of each key design component, allowing us to explore opportunities for further improvements.

B. Inference Accuracy

TABLE II
INFERENCE ACCURACY

Methodology	KWS	HAR	SQN
16-bit quantized model	92.60%	75.55%	80.08%
Stateful DNN	92.60%	75.44%	80.14%

TABLE II shows the inference accuracy achieved by Stateful compared with the original 16-bit quantized model. We report the inference accuracy of each evaluated model, measured based on the test set of each respective dataset (TABLE I). Although the downscaling step of state embedding causes the values of the IFMs and OFMs to be different from those of the original model, the results indicate there is no significant impact on the accuracy of the evaluated models. Specifically, Stateful shows an accuracy loss of 0.11% for the SQN model, no accuracy change for the HAR model and even an accuracy gain of 0.06% for the KWS model, although, by design, Stateful does not improve the inference accuracy. As discussed in Section IV-C2, the accuracy loss is minimal because: (1) state embedding scales down the network component values to one of only two subranges to represent the inference state, (2) state clearing removes the embedded states to produce the original output features and (3) infrequent overflows reduce the impact of overflow handling.

C. Progress Preservation Overhead

Fig. 8 shows the progress preservation overhead incurred by each approach as a proportion of an end-to-end inference

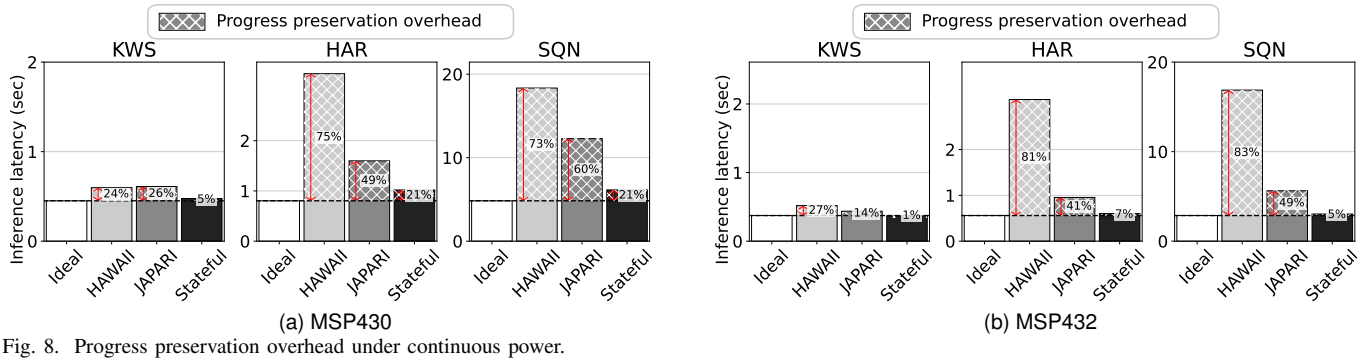


Fig. 8. Progress preservation overhead under continuous power.

latency, obtained by comparing against the ideal baseline, under continuous power, across all three DNNs and the two prototype devices. HAWAII has the largest overhead (>70% in many cases) as it requires an individual data transfer command to preserve each progress indicator (Section III). HAWAII's overhead is larger for convolutional networks (e.g., HAR and SQN) that have more jobs, each paired with a progress indicator, and also on the MSP432, where a data transfer is more costly than a job computation. By augmenting the DNN, JAPARI allows progress indicators and job outputs to be simultaneously preserved with one data transfer command (Section II-B), reducing the data transfer overhead compared to HAWAII, but effectively doubling the number of job computations. JAPARI's additional computational overhead is more obvious on the MSP430, which is computationally inefficient relative to the MSP432. Interestingly, unlike Stateful, JAPARI also imposes additional data transfers, related to extra NVM reads, because the augmentations made on one layer increases the number of input features fetched into VM for subsequent layers. Hence, JAPARI incurs higher overhead for DNN models with more input features per layer, such as convolutional networks (e.g., SQN and HAR), particularly wide networks (i.e., more channels per layer) with multiple paths (e.g., SQN).

As Stateful does not separately preserve progress indicators and job outputs, it does not incur any additional data transfers, thus considerably reducing the runtime overhead compared to HAWAII and JAPARI, across all evaluation conditions. Stateful incurs a lower overhead for networks with more FC layers that have fewer jobs and input/output features than CONV layers, resulting in lower state embedding and clearing overheads. Importantly, Stateful's runtime overhead does not dominate the inference latency, even for convolutional networks with predominantly CONV layers (e.g., HAR and SQN). Furthermore, as the runtime overhead of Stateful is mainly computational (Section VI-E), the proportion of its overhead is lower on the MSP432 than on the MSP430.

Overall, the progress preservation overheads incurred by HAWAII and JAPARI, which respectively account for 24% to 83% and 14% to 60% of the inference latency, are reduced down to 1% to 21% by Stateful, depending on the DNN model and hardware device.

D. Inference Latency

Fig. 9 shows the end-to-end inference latency under intermittent power for each approach[3], under different DNN models, hardware devices and power strengths. The numbers above the bar plots indicate how many times Stateful's intermittent inference is faster than that of HAWAII and JAPARI, which correlates well with their overheads as discussed in Section VI-C. Stateful drastically reduces the inference latency compared to HAWAII because the latter incurs a progress preservation overhead that is much higher than that of the former. Note that under weak power (i.e., $4\,\mathrm{mW}$), all approaches experience frequent power failures, thus losing more unpreserved progress and as a result incurring a larger number of job re-executions. However, approaches with a high progress preservation overhead, incur the most progress loss, as most of the available energy is spent on progress preservation instead on inference. Thus, Stateful shows significant improvements over HAWAII when the latter's overhead is at its highest, specifically under weak power, when executing convolutional networks on the MSP432.

Stateful also achieves large inference latency improvements over JAPARI, across all evaluation conditions. Stateful more obviously outperforms JAPARI when executing wider, multiple path convolutional networks such as SQN, because JAPARI's progress preservation overhead is more dominant for such models (Section VI-C). Furthermore, the improvements of Stateful over JAPARI increase under weak power. This is because JAPARI incurs a higher overhead than Stateful and also doubles the number of job computations during inference. Therefore under weak power, Stateful suffers from much less progress loss and as a result incurs a significantly lower number of job re-executions than JAPARI.

Overall, Stateful shows the lowest inference latency among all evaluated approaches, in all scenarios. Stateful can speed up intermittent DNN inference by 1.3 to 5 times and 1.3 to 2 times respectively compared to HAWAII and JAPARI. The improvements are generally higher for neural networks typically used by edge applications [30], especially modern convolutional networks that have a wide and multiple path architecture, when executed on computationally lightweight devices, under weak power.

E. Runtime Overhead Breakdown

Stateful incurs a runtime overhead to allow a DNN to intrinsically contain inference state information (Section IV-E). During inference, the state embedder transforms the values of specific network components to represent the inference states, the state assigner maintains a state table to determine the specific states to embed, the state clearer removes the states from the input features to avoid computation corruption, and

[3]The ideal baseline is excluded as it cannot make forward progress under intermittent power.

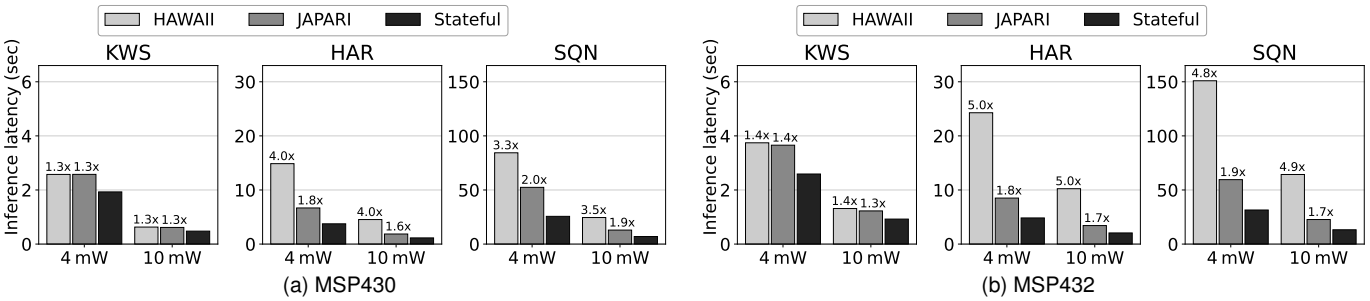


Fig. 9. Inference latency under intermittent power.

upon recovery, the progress seeker searches the NVM for the latest preserved job output. Fig. 10 shows a breakdown of Stateful's overhead in terms of its key design components, for the evaluated DNNs, under continuous and intermittent power, on the MSP430. The numbers inside the stacked bars indicate the absolute latency of each design component in milliseconds. Note that the overhead is more noticeable under weak power, due to a higher job re-execution and progress recovery cost, and we expect the costs of the state embedder, assigner and clearer to be lower on the MSP432. We also ensured the cost of code instrumentation required to capture the overhead breakdown was kept low.

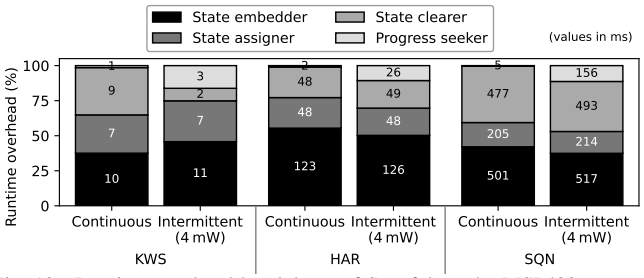


Fig. 10. Runtime overhead breakdown of Stateful on the MSP430.

In all cases, the state embedder is the most costly component. As discussed in Section IV-C1, for linear layers, it embeds states into all biases and also transforms the input features. For non-linear layers, it embeds states into all job outputs, so its overhead is higher for networks with more non-linear layers (e.g., HAR). Hence, the amount of embedded state information varies depending on the structure of the network model, where states are embedded into 0.6%, 1.0% and 0.9% of the total model parameters in KWS, HAR and SQN, respectively, and into 29%, 30% and 33% of all job outputs in the respective networks. By contrast, the state clearer only reverts values of the input features, the state assigner reduces its overhead by fetching only a few entries of the state table to VM, and the progress seeker incurs only a few binary search iterations per power cycle. However, the state clearer's overhead increases for networks with more CONV layers (e.g., SQN), as they have more input features.

Importantly, power instability does not significantly impact the overhead of Stateful where, under intermittent power, the proportion of the runtime overhead is 5% to 22% of the total time the device is powered on, which is only a marginal increase compared to the overhead under continuous power (Section VI-C). The overheads of the components responsible for progress preservation (i.e., the state embedder, clearer and assigner) do not grow drastically moving from continuous to intermittent power. However, although still not dominant, the overhead of the progress seeker increases under weak intermittent power, as the latest job output in NVM has to be searched in each power cycle.

VII. Discussion

There are still several possibilities to further improve Stateful. As observed in Section VI-C, the overhead of Stateful grows with the numbers of jobs and input/output features in the network. However, to further reduce the overhead, instead of pairing each job output with a progress indicator, multiple job outputs can be paired with one progress indicator, to decrease the number of times state embedding and state clearing are carried out, thus trading a higher re-execution overhead for a lower progress preservation overhead. Although for portability, Stateful has been implemented solely in software using only the CPU, further overhead reductions can also be achieved if Stateful is implemented using the hardware accelerator, which may also reduce the additional overhead for systems without a shared memory region between the CPU and accelerator.

Similar to existing intermittent inference approaches, Stateful also inevitably incurs additional overheads related to system reboot, data re-fetching due to power loss and progress recovery in each power cycle. The cost of system reboot is typically small and constant, the cost of data re-fetching is already minimized in Stateful, by fetching only the tile input data required for the remaining jobs in the tile, and the cost of progress recovery is minimized by restricting the search region in the NVM buffer (Section IV-E). Although progress preservation is currently the major overhead factor in existing approaches, these aforementioned costs may dominate the inference latency given extremely frequent power failures. Hence, instant progress recovery for inference resumption remains an interesting avenue for further investigation.

VIII. Concluding Remarks

This paper presents the concept of stateful neural networks, to allow a network to inherently indicate the inference progress itself. Existing intermittent inference approaches suffer from an excessive progress preservation overhead because they require progress indicators to be preserved separately from the computed output features. In contrast, our stateful DNN middleware embeds inference state information into the network model during inference, allowing progress indicators to be intrinsically contained within output features, thereby significantly alleviating the overhead of progress preservation. We realize and evaluate our proposed approach on two Texas Instruments devices, with different computational efficiency.

By eliminating additional data transfers for progress indicator preservation and incurring only a moderate runtime overhead, our stateful DNN approach can significantly speed up inference, without noticeable accuracy degradation, compared to two recent attempts specifically developed for intermittent DNN inference [8], [9]. Our approach is highly suitable for convolutional networks deployed on tiny, severely resource-constrained devices, executed under intermittent power.

Our stateful DNN middleware stack, which transparently embeds inference state information into neural networks, has been made open [50], simplifying the development of low latency, intelligent applications on battery-less devices.

REFERENCES

[1] J. Hester and J. Sorber, "New directions: The future of sensing is batteryless, intermittent, and awesome," in *Proc. of ACM SenSys*, 2017, pp. 1–6.

[2] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE TCAD*, vol. 35, no. 12, pp. 1968–1980, 2016.

[3] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," in *Proc. of ACM OOPSLA*, 2017, pp. 1–30.

[4] V. Talla, B. Kellogg, S. Gollakota, and J. R. Smith, "Battery-Free Cellphone," *ACM IMWUT*, vol. 1, no. 2, pp. 25:1–20, 2017.

[5] S. Naderiparizi, A. N. Parks, Z. Kapetanovic, B. Ransford, and J. R. Smith, "WISPCam: A battery-free RFID camera," in *Proc. IEEE RFID*, 2015, pp. 166–173.

[6] J. De Winkel, V. Kortbeek, J. Hester, and P. Pawełczak, "Battery-free Game Boy," *ACM IMWUT*, vol. 4, no. 3, pp. 1–34, 2020.

[7] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *Proc. of ACM ASPLOS*, 2019, pp. 199–213.

[8] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu, "More is less: Model augmentation for intermittent deep inference," *ACM TECS*, 2022.

[9] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu, "Everything leaves footprints: Hardware accelerated intermittent deep inference," *IEEE TCAD*, vol. 39, no. 11, pp. 3479–3491, 2020.

[10] S. Islam, J. Deng, S. Zhou, C. Pan, C. Ding, and M. Xie, "Enabling fast deep learning on tiny energy-harvesting IoT devices," in *Proc. of IEEE/ACM DATE*, 2022.

[11] H. R. Mendis, C.-K. Kang, and P.-C. Hsiu, "Intermittent-aware neural architecture search," *ACM TECS*, vol. 20, no. 5s, pp. 64:1–27, 2021.

[12] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *Proc. of ICLR*, 2019, pp. 1–26.

[13] F. Zhu, R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang, and J. Yan, "Towards unified INT8 training for convolutional neural network," in *Proc. of IEEE CVPR*, 2020, pp. 1969–1979.

[14] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," in *Proc. of ICLR*, 2017.

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. of IEEE CVPR*, 2018, pp. 4510–4520.

[16] W. Jiang, L. Yang, S. Dasgupta, J. Hu, and Y. Shi, "Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start," *IEEE TCAD*, vol. 39, no. 11, pp. 4154–4165, 2020.

[17] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," in *Proc. of ICLR*, 2019.

[18] W. Jiang, L. Yang, E. H.-M. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, and J. Hu, "Hardware/software co-exploration of neural architectures," *IEEE TCAD*, vol. 39, no. 12, pp. 4805–4815, 2020.

[19] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Memory-efficient patch-based inference for tiny deep learning," in *Proc. of NeurIPS*, 2021, pp. 1–13.

[20] Kendryte, "K210 AI chip datasheet," https://github.com/kendryte/kendryte-doc-datasheet, 2018.

[21] Maxim Integrated, "MAX78000 ultra-low-power MCU with Arm Cortex-M4 and a CNN accelerator," https://datasheets.maximintegrated.com/en/ds/MAX78000.pdf, 2021.

[22] Texas Instruments, "Low-energy accelerator," https://www.ti.com/lit/an/slaa720/slaa720.pdf, 2016.

[23] K. Qiu, N. Jao, M. Zhao, C. S. Mishra, G. Gudukbay, S. Jose, J. Sampson, M. T. Kandemir, and V. Narayanan, "ResiRCA: A resilient energy harvesting ReRAM crossbar-based accelerator for intelligent embedded processors," in *Proc. of IEEE HPCA*, 2020, pp. 315–327.

[24] ARM, "Cortex-M4 instructions," https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC, 2010.

[25] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with Just-in-Time checkpoints," in *Proc. of ACM PLDI*, 2019, pp. 1101–1116.

[26] Y. Wu, Z. Wang, Z. Jia, Y. Shi, and J. Hu, "Intermittent inference with nonuniformly compressed multi-exit neural network for energy harvesting powered devices," in *Proc. of IEEE/ACM DAC*, 2020, pp. 1–6.

[27] F. Li, K. Qiu, M. Zhao, J. Hu, Y. Liu, Y. Guan, and C. J. Xue, "Checkpointing-aware loop tiling for energy harvesting powered non-volatile processors," *IEEE TCAD*, vol. 38, no. 1, pp. 15–28, 2019.

[28] Texas Instruments, "MSP430FR5994 MCU," https://www.ti.com/product/MSP430FR5994, 2018.

[29] Texas Instruments, "MSP432P401R MCU," https://www.ti.com/product/MSP432P401R, 2019.

[30] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. Patterson, D. Pau, J. sun Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav, "Benchmarking TinyML systems: Challenges and direction," *arXiv:2003.04821*, 2020.

[31] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[32] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. of ACM FPGA*, 2015, pp. 161–170.

[33] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support," in *Proc. of USENIX OSDI*, 2016, pp. 17–32.

[34] K. Qiu, N. Jao, K. Zhou, Y. Liu, J. Sampson, M. T. Kandemir, and V. Narayanan, "MaxTracker: Continuously tracking the maximum computation progress for energy harvesting ReRAM-based CNN accelerators," *ACM TECS*, vol. 20, no. 5s, pp. 78:1–23, 2021.

[35] Cypress, "4-Mbit SPI FRAM," https://www.cypress.com/file/209146/download, 2019.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of IEEE CVPR*, 2016, pp. 770–778.

[37] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv:2004.09602*, 2020.

[38] Greenwaves Technologies, "GAP-8 IoT application processor with a hardware convolution engine," https://greenwaves-technologies.com/gap8_gap9/, 2018.

[39] STMicroelectronics, "STM32G4 series mixed-signal MCUs with DSP and FPU instructions," https://www.st.com/en/microcontrollers-microprocessors/stm32g4-series.html, 2021.

[40] H. Peng, S. Zhou, S. Weitze, J. Li, S. Islam, T. Geng, A. Li, W. Zhang, M. Song, M. Xie, H. Liu, and C. Ding, "Binary complex neural network acceleration on FPGA," in *Proc. of IEEE ASAP*, 2021, pp. 85–92.

[41] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *Proc. of ICML*, 2017, pp. 815–824.

[42] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv:1711.07128*, 2017.

[43] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv:1804.03209*, 2018.

[44] B. Himmetoglu, "CNN model for human activity recognition," https://github.com/healthDataScience/deep-learning-HAR, 2017.

[45] D. Anguita, A. Ghio, L. Oneto, X. Parra Perez, and J. L. Reyes Ortiz, "A public domain dataset for human activity recognition using smartphones," in *Proc. of ESANN*, 2013, pp. 437–442.

[46] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

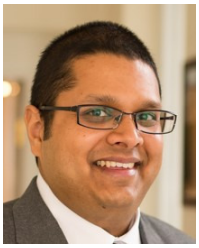
[47] I. Corporation, "IXOLAR high efficiency SolarMD," https://ixapps.ixys.com/DataSheet/SM111K04L.pdf, 2010.

[48] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu, "HAWAII open source project," https://github.com/EMCLab-Sinica/HAWAII_Project, 2020.

[49] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu, "JAPARI open source project," https://github.com/EMCLab-Sinica/JAPARI, 2022.

[50] C.-H. Yen, H. R. Mendis, T.-W. Kuo, and P.-C. Hsiu, "Stateful open source project," https://github.com/EMCLab-Sinica/stateful-cnn, 2022.

Chih-Hsuan Yen (Graduate Student Member, IEEE) received the B.S. degree from the Department of Electrical Engineering and M.S. degree from the Department of Computer Science and Information Engineering, National Taiwan University, Taiwan, in 2015 and 2017, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Information Engineering, National Taiwan University, Taiwan. He is also a Research Assistant with the Research Center for Information Technology Innovation, Academia Sinica, Taiwan. His research interests include embedded systems and intermittent computing.



Hashan Roshantha Mendis (Member, IEEE) received his MSc. and Eng.D degrees respectively from the Department of Electronics and Department of Computer Science in the University of York, UK, in 2011 and 2017. He is currently a postdoctoral research fellow at the Research Center for Information Technology Innovation (CITI), Academia Sinica, Taiwan. His research interests include intermittently-powered embedded systems, embedded deep learning, as well as the design and optimization of multi/many-core systems.

Tei-Wei Kuo (Fellow, IEEE) received the B.S.E. degree in computer science from National Taiwan University, Taiwan, in 1986, and the Ph.D. degree in computer science from the University of Texas at Austin, USA, in 1994.

He is currently a Distinguished Professor of the Department of Computer Science and Information Engineering, National Taiwan University, where he was an Interim President (2017-2019) and an Executive Vice President for Academics and Research (2016-2019). He was the Lee Shau Kee Chair Professor of information engineering, an Advisor to President of information technology and the Founding Dean of the College of Engineering, City University of Hong Kong, Hong Kong (2019-2022). His research interest includes embedded systems, non-volatile-memory software designs, neuromorphic computing and real-time systems. Prof. Kuo received numerous awards and recognition, including Humboldt Research Award in 2021, Outstanding Technical Achievement and Leadership Award from IEEE TC on Real-Time Systems and the Distinguished Leadership Award from IEEE TC on Cyber-Physical Systems both in 2017. He is an executive committee member of IEEE TC on Real-Time Systems (TCRTS), the Vice Chair of ACM SIGAPP and the Chair of ACM SIGBED Award Committee. He was the founding Editor-in-Chief of ACM Transactions on Cyber-Physical Systems (2015-2021) and serves in the Program Committees of many top conferences. Prof. Kuo has over 300 technical papers and received many best paper awards, including the Best Paper Award from IEEE/ACM CODES+ISSS 2019 and ACM HotStorage 2021. Dr. Kuo is a Fellow of ACM and US National Academy of Inventors and also a Member of European Academy of Sciences and Arts.

Pi-Cheng Hsiu (Senior Member, IEEE) received the B.S. degree in computer information science from National Chiao Tung University, Taiwan, in 2002, and the M.S. and Ph.D. degrees in computer science and information engineering from National Taiwan University, Taiwan, in 2004 and 2009, respectively.

He is currently a Research Fellow (Professor) and Deputy Director of the Research Center for Information Technology Innovation (CITI), where he leads the Embedded and Mobile Computing Laboratory, and is also a Joint Research Fellow with the Institute of Information Science, Academia Sinica, a Jointly Appointed Professor with the Department of Computer Science and Engineering, National Chi Nan University, and a Jointly Appointed Professor with the College of Electrical Engineering and Computer Science, National Taiwan University, Taiwan. He joined CITI as an Assistant Research Fellow in 2009, and was promoted to an Associate Research Fellow in 2013 and to a Research Fellow in 2018. He was a Visiting Scholar with the Department of Computer Science, University of Illinois at Urbana-Champaign, USA, in 2007, and with the Department of Electrical and Computer Engineering, University of Pittsburgh, USA, in 2019. His research interests include embedded systems, mobile systems and real-time systems. Dr. Hsiu's work has been awarded the Best Paper Award at IEEE/ACM CODES+ISSS 2020 and 2021 in a row, and nominated for the Best Paper Award in 2019. He serves as an Associate Editor for the ACM Transactions on Cyber-Physical Systems and in the Program Committees of many international conferences in his field, such as CODES+ISSS, DAC, ISLPED, RTSS and RTAS. He is a Senior Member of ACM.