More is Less: Model Augmentation for Intermittent Deep Inference

CHIH-KAI KANG, National Taiwan University and Academia Sinica, Taiwan
HASHAN ROSHANTHA MENDIS, Academia Sinica, Taiwan
CHUN-HAN LIN, National Taiwan Normal University, Taiwan
MING-SYAN CHEN, National Taiwan University and Academia Sinica, Taiwan
PI-CHENG HSIU, Academia Sinica, National Taiwan University and National Chi Nan University, Taiwan

Energy harvesting creates an emerging *intermittent computing* paradigm, but poses new challenges for sophisticated applications such as intermittent *deep neural network* (DNN) inference. Although *model compression* has adapted DNNs to resource constrained devices, under intermittent power, compressed models will still experience multiple power failures during a single inference. Footprint-based approaches enable hardware accelerated intermittent DNN inference by tracking *footprints*, independent of model computations, to indicate accelerator progress across power cycles. However, we observe that the extra overhead required to preserve progress indicators can severely offset the computation progress accumulated by intermittent DNN inference.

This work proposes the concept of *model augmentation* to adapt DNNs to intermittent devices. Our middleware stack, JAPARI, appends extra neural network components into a given DNN, to enable the accelerator to intrinsically integrate progress indicators into the inference process, without affecting model accuracy. Their specific positions allow progress indicator preservation to be piggybacked onto output feature preservation to amortize the extra overhead, and their assigned values ensure uniquely distinguishable progress indicators for correct inference recovery upon power resumption. Evaluations on a Texas Instruments device under various DNN models, capacitor sizes, and progress preservation granularities, show that JAPARI can speed up intermittent DNN inference by 3x over the state of the art, for common convolutional neural architectures that require heavy acceleration.

CCS Concepts: • **Computer systems organization → Embedded software**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Deep neural networks, intermittent systems, model adaptation, energy harvesting, edge computing.

Authors' addresses: Chih-Kai Kang, Graduate Institute of Electrical Engineering, National Taiwan University and Research Center for Information Technology Innovation (CITI), Academia Sinica, Taiwan, ckkang@arbor.ee.ntu.edu.tw; Hashan Roshantha Mendis, Research Center for Information Technology Innovation (CITI), Academia Sinica, Taiwan, rosh.mendis@citi.sinica.edu.tw; Chun-Han Lin, Department of Computer Science and Information Engineering (CSIE), National Taiwan Normal University, chlin@ntnu.edu.tw; Ming-Syan Chen, Graduate Institute of Electrical Engineering, National Taiwan University and Research Center for Information Technology Innovation (CITI), Academia Sinica, Taiwan, mschen@ntu.edu.tw; Pi-Cheng Hsiu (corresponding author), Research Center for Information Technology Innovation (CITI), Academia Sinica and Department of Computer Science and Information Engineering (CSIE), National Chi Nan University, and the Data Science Degree Program, National Taiwan University, Taiwan, pchsiu@citi.sinica.edu.tw.
This work was supported in part by the Ministry of Science and Technology, Taiwan, under grant MOST 110-2222-E-001-003-MY3.

1539-9087/2022/0-ARTX $15.00
https://doi.org/10.1145/3506732

ACM Reference Format:
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. Embedd. Comput. Syst.* X, X, Article X (2022), 26 pages. https://doi.org/10.1145/3506732

1 INTRODUCTION

Energy harvesting enables the development of battery-less, sustainable and cost-effective Internet of Things (IoT) devices, thus creating innovative applications like solar-powered cameras [57], wireless-powered cellphones [66], and battery-free game consoles [20]. However, such devices encounter unique challenges as applications on these devices suffer from frequent power failures and thus are executed *intermittently*, i.e., when energy is available [15, 30, 49]. Given the scarcity of ambient energy, it is prohibitive to send raw sensor data and offload computations to remote servers. Therefore, to provide responsive applications with efficient communication bandwidth usage, intermittent systems frequently make decisions *locally* and execute lightweight versions of intelligent algorithms such as *deep neural networks* (DNNs) [65]. Although computationally expensive, DNN inference on lightweight devices becomes a possibility with hardware acceleration available on modern ultra-low power microcontrollers (MCUs) [16, 35, 39, 67]. Therefore, *hardware accelerated intermittent DNN inference* has emerged as a crucial challenge as future IoT devices become self-powered and rely more on local, on-device inference.

The shift towards local inference has prompted significant research on *model adaptation* approaches, to make DNN inference feasible on resource and energy constrained embedded IoT devices [65]. Specifically, *model compression* techniques such as network pruning [51, 58, 71], weight sharing [29, 68], and non-uniform quantization [46, 69] substantially reduce the computational complexity of DNN models in order to speed up DNN inference, while minimizing the degradation to model accuracy. Depending on the application requirements, some techniques are designed to provide high inference performance but require custom hardware support (e.g., FPGAs [28]), while other techniques may target commercial off-the-shelf platforms (e.g., MCUs [22]) as they are widely available and easily programmable [25].

Existing model adaptations are well suited for battery-powered systems, but on an intermittent system, where the harvested power is usually weak and unstable [40, 52], in most cases, a compressed model would still require multiple power cycles to complete a single end-to-end inference [24, 72]. Some techniques have been proposed to allow intermittent execution across power cycles. *Checkpointing-based* approaches backup volatile system state in non-volatile memory (NVM) [3, 8, 41, 60], but the high memory demand of DNN inference results in large checkpoints, and therefore significant runtime overhead, which scales with the backup size and frequency. Moreover, checkpointing cannot correctly preserve the progress of peripherals with inaccessible internal state, such as accelerators [10, 55]. Alternatively, *task-based* approaches [12, 14, 54] partition the application into multiple *atomic* (i.e., power uninterrupted) tasks that fit within an energy budget (capacitor), and only backup the task indices and modified data at task boundaries. However, energy-aware task partitioning is non-trivial for DNN inference even with tool assistance [24], and minor changes to the DNN model may require energy re-profiling.

In light of intrinsic peripheral behavior, *footprint-based* approaches [43, 56] track *footprints* to indicate the progress of a peripheral operation, and thus allow for *accumulative* peripheral execution, without requiring access to the peripheral internal state and application-level energy estimation. Specifically, *footprint-based accelerated DNN inference* [43] preserves the accelerator outputs (i.e., weighted sums) and footprints (i.e., progress indicators) in NVM during DNN inference, and upon recovery from power failure, uses footprints to *resume* the interrupted operation. One accelerator operation may comprise multiple sub-operations, where each performs an *atomic* computation

(called a *job*). Therefore, footprint-based approaches require re-executing only the last incomplete job, not the entire task [24], and also enables sub-operation (job) outputs to be preserved in *parallel* to operation computation.

This work proposes the novel concept of *model augmentation* to adapt DNN models to intermittently-powered edge devices. Model augmentation is motivated by an observation that as progress indicators are generated *independent* of the job outputs computed by the accelerator, they incur extra progress preservation overhead, which can significantly offset the benefit of *accumulative* execution enabled by intermittent DNN inference. In contrast to state-of-the-art intermittent inference approaches [24, 43], model augmentation appends extra neural network components to a DNN model to intrinsically integrate progress indicators into the inference process. This allows for job outputs and progress indicators to be generated in an alternate manner, and therefore progress indicator preservation can be *piggybacked* onto job output preservation, thereby amortizing the extra overhead of progress preservation. The additional computation overhead brought by model augmentation is compensated by a reduction in data transfer overhead, because the accelerator computation cost is typically much lower than the NVM access cost. While seemingly mutually contrary, model compression and augmentation can be combined to speed up intermittent edge DNN inference.

To realize model augmentation, we develop *JAPARI* (Job and Progress Alternate Inference), a footprint-based middleware stack, which addresses two key design challenges of model augmentation. The first challenge is to determine *where* in the model architecture to augment in order for the accelerator to alternately generate job outputs and footprints, to enable simultaneous preservation. This challenge is addressed by *footprint appending*, where footprint kernels and channels are inserted into specific *positions* without corrupting the model architecture, to generate the required interleaved layout. The second challenge is to determine *how* to identify the interrupted job based on the preserved footprints, such that an inference can correctly resume upon power resumption. This challenge is addressed by *footprint representation*, where footprint kernels and channels are assigned specific *values* without affecting the model accuracy, to ensure the latest footprint is uniquely distinguishable from prior footprints that reuse the same preservation buffer.

We prototyped JAPARI on the Texas Instruments (TI) MSP430FR5994 LaunchPad [36] with an internal low-energy accelerator (LEA), 8KB SRAM (VM), and external 1MB FRAM (NVM) [1]. Experiments were conducted under intermittent power with different energy buffer capacities, as well as under continuous power. Three DNN models were used for evaluation, with varied levels of complexity, representative of those typically used in tiny machine learning applications [9]. Compared to a task-based intermittent inference approach similar in concept to TAILS [24], and a state-of-the-art footprint-based approach called HAWAII [43], JAPARI respectively reduces the *energy consumption* of an end-to-end inference by 52% and 26% on average, thereby reducing the *inference time* by 42% and 25%. The reduction is more evident when data transfers required for progress preservation in a DNN model can be largely converted into computations.

The remainder of this paper is organized as follows. Section 2 provides background information and Section 3 explains the motivation for this work. Section 4 presents our JAPARI design, with implementation details given in Section 5. Experimental results are reported in Section 6, and Section 7 discusses the limitations of JAPARI and future extensions. Section 8 presents some concluding remarks.

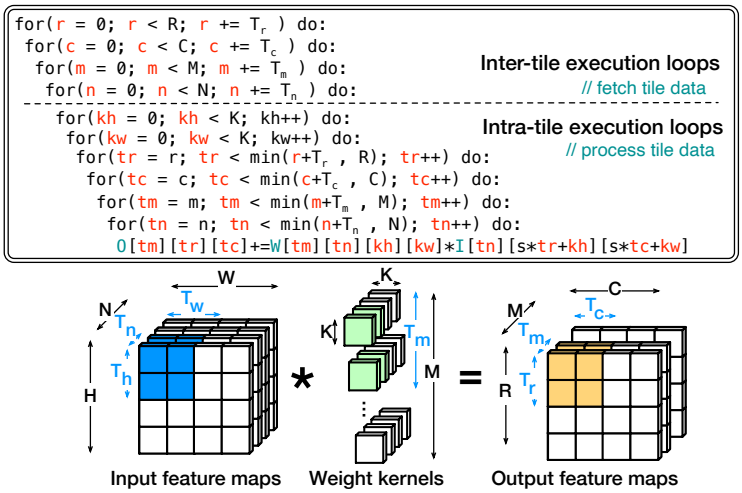


Fig. 1. Tiled convolution

2 BACKGROUND

2.1 Deep Neural Network Acceleration

DNNs consist of multiple connected layers (L) such as convolutional (CONV), fully connected (FC) and pooling (POOL) layers [65]. A CONV layer, for example, takes input feature maps (IFMs) with N channels and $W \times H$ spatial dimensions and convolves them with M weight kernels of dimension $N \times K \times K$ to produce M output feature maps (OFMs) with $C \times R$ spatial dimensions. Due to their large size, the IFMs, weights, and OFMs of a layer are stored in NVM and logically partitioned into *tiles* during inference computation on edge devices. As shown in Figure 1, to compute an OFM tile of size $T_m \times T_c \times T_r$, *tiled input data* need to be fetched from NVM to VM, specifically an *IFM tile* of size $T_n \times T_w \times T_h$, a *weight kernel tile* of T_m kernels with size $T_n \times K \times K$ and stride s, and also the previously computed partial sums of the *OFM tile*. The size of a tile computation is determined by the $< T_n, T_m, T_c, T_r >$ parameters.

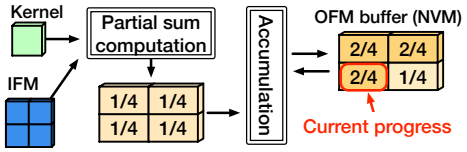
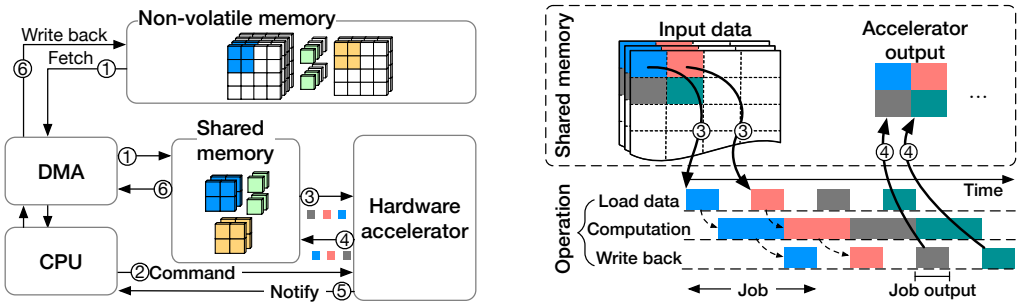


Fig. 2. Partial sum accumulation in the OFM buffer

To complete a layer, OFM tiles are sequentially computed in a certain order. The loop order and associated dataflow may vary across different accelerator designs, depending on factors such as VM size and data reuse strategy. Figure 1 (top) shows a tiled convolution computation order, commonly used in accelerator designs [42, 63, 74]. The *intra-tile* and *inter-tile* execution loops respectively determine the computation order within an OFM tile and the computation order among OFM tiles. Due to limited VM space and accelerator capability, the output features within an OFM tile are typically not computed using one accelerator operation; instead, each output feature is accumulated from those corresponding *partial sums* computed individually in the channel dimension. To this

end, an *OFM buffer* (of size $M \times C \times R$) in NVM is used to store the computed OFM tiles of a layer. As illustrated in Figure 2, previously computed partial sums in the OFM buffer are *accumulated* with the partial sums of the currently computed OFM tile (e.g., given $\frac{N}{T_n} = 4$, four inter-tile accumulation iterations are required) to fully complete the output features.



(a) High-level architecture (b) Dataflow inside the accelerator

Fig. 3. Hardware accelerated DNN inference

Recent ultra-low power microcontrollers (MCUs) contain computation peripherals such as general vector math accelerators [16, 35] and dedicated convolution accelerators [39, 67]. As shown in Figure 3(a), an on-chip accelerator typically shares a VM region with the CPU, and tiled input data are fetched from NVM into the shared VM before accelerator *operations* are invoked. By an operation, we indicate any command available on the accelerator (e.g., matrix multiplication or smaller vector multiply-accumulate operation). One operation can contain multiple sub-operations which produce the *minimum* intermediate outputs *intrinsically* written back by the accelerator to the shared VM. One sub-operation computation is referred to as a *job*. A lightweight accelerator with very limited internal memory typically has a *fine-grained* job, which can be a partial sum, a partially accumulated output feature, or a fully completed output feature, depending on the sub-operation. As shown in Figure 3(b), to produce job outputs, the accelerator loads the input data into its internal memory, computes the jobs and writes back the job outputs to the shared VM, in a pipelined fashion.

2.2 Intermittent Deep Model Inference

Intermittent systems rely on ambient power sources. Harvested energy is accumulated into an energy buffer (e.g., a capacitor) [41, 49, 52], and the intermittent system is powered ON when the buffered energy level reaches a preset threshold and powered OFF when the energy buffer is depleted. Therefore, the power ON duration in a power cycle is determined by the incoming power, buffered energy, and energy consumption. NVM is leveraged for state persistence across power cycles, but NVM typically has higher data access energy and latency costs than VM, so for performance and energy concerns, the application is run from VM. The direct memory access (DMA) controller is utilized for efficient data transfer between NVM and VM, while the hardware accelerator is used to speed up specific computations.

DNN inference on intermittent systems has recently been made possible [24, 43], by accumulating inference *progress* across multiple power cycles. Existing intermittent inference approaches take a pre-trained DNN model and execute it intermittently without any modifications on the model. During inference, *progress indicators* are preserved along with accelerator outputs from VM to

NVM. The unpreserved progress in VM will be lost upon power failure and has to be *re-executed* in the next power cycle. After power resumption, the progress indicators are used to identify the interrupted progress, re-fetch the required input data, and correctly resume the inference process. Intermittent inference approaches are different in the progress indicators used (e.g., loop indices [24] or layer/job counters [43]), resulting in different preservation granularities (at the operation [24] or sub-operation level [43]) and overheads.

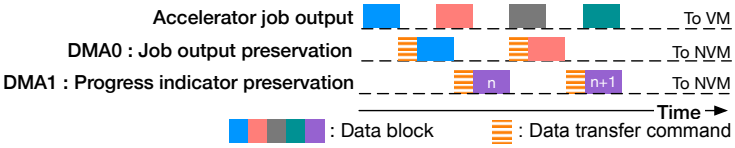


Fig. 4. Inference progress preservation

Figure 4 illustrates progress preservation in HAWAII [43], a state-of-the-art hardware accelerated intermittent inference approach. The job output preservation and progress indicator preservation (i.e., transferring job outputs and progress indicators to NVM) are carried out separately via two DMA channels. Accordingly, the previous job output and its corresponding indicator (called *footprint*) are preserved in a *pipelined* fashion, and in *parallel* with the current job computation, to reduce the runtime overhead. The job outputs and progress indicators are preserved in *non-contiguous* NVM locations, respectively in the OFM buffer and in a persistent variable. Hence, *two* separate DMA channels and data transfer commands are required for both job output and progress indicator preservation. The overhead of a *data transfer command* includes DMA invocation, NVM invocation, and the NVM write. If this overhead required for progress indicator preservation is significant, it would offset the benefit brought by accumulating inference progress across power cycles, thereby impeding intermittent DNN inference.

3 PROGRESS PRESERVATION: OBSERVATION AND MOTIVATION

Progress preservation is essential for intermittent systems. To explore a new means for progress preservation, we conducted an experiment on a Texas Instruments (TI) MSP430FR5994 device [36], with the TI low energy accelerator (LEA), 8KB internal VM and 1MB external NVM [1]. Vector math operations in LEA were used to compute a convolution tile (i.e., partial sum computation and accumulation as described in Section 2.1). The tile size was set as $T_n = T_m = 8$ and $T_c = T_r = 4$, and the kernel size was set as $K = 3$, allowing the tiled input data to fit in VM. An accelerator operation consists of T_m sub-operations (jobs). The DMA controller was used for job output and progress indicator preservation. We repeatedly executed the convolution tile, and the *inference time* was measured as the average time required to compute the tile, under a power source (3 mW) and two respective capacitor sizes (100 μF and 1 mF). The power source was insufficient for the device to operate continuously, leading to repeated yet unpredictable power failures.

We compare three progress preservation configurations. The first (baseline), similar to HAWAII [43], uses a *footprint* to indicate each computed job during inference. As a job output and footprint pair is preserved in *non-contiguous* NVM locations, two *separate* DMA channels and data transfer commands are required for progress preservation, resulting in $2T_m$ total data transfer commands, each writing one data block of 2 bytes to NVM. The second configuration (denoted 1DMA-2) uses a *single* DMA channel to *simultaneously* preserve a job output and footprint pair. In 1DMA-2, we double the tile size (i.e., $2T_m = 16$) to essentially double the number of accelerator computations, in order to emulate the behavior of the accelerator that computes an augmented tile and writes

contiguous job output and footprint pairs to shared VM (details provided in Section 4). Accordingly, 1DMA-2 writes an equal number of total data blocks from VM to NVM as the baseline, but using only *half* the number of transfer commands. The third configuration (denoted 1DMA-ALL) also uses one DMA channel, but only a *single* transfer command is used to preserve *all* accelerator outputs of an operation, with the equal number of data blocks written to NVM. 1DMA-ALL exploits the slower NVM write compared to an accelerator operation, where the current accelerator output is available in the shared VM before the previous accelerator output preservation has completed. All configurations transfer the same number of data blocks to NVM, but they differ in terms of the number of transfer commands invoked for an accelerator operation, respectively $2T_m$, T_m, and 1 for the baseline, 1DMA-2, and 1DMA-ALL.

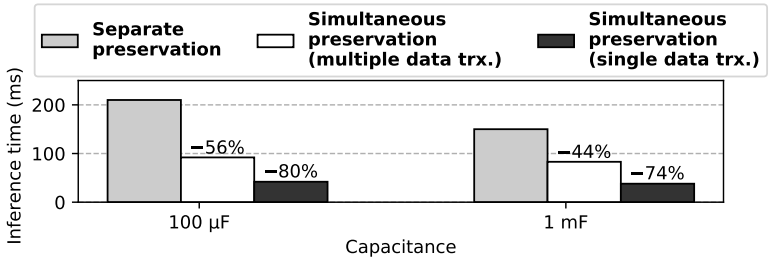


Fig. 5. Inference time under different progress preservation configurations

The experiment result (Figure 5) shows that different progress preservation configurations have a significant impact on the inference time, and the impact is more evident when power failures occur more frequently under smaller capacitance. In the baseline, separate preservation of job outputs and footprints can lead to significant overhead which prolongs the inference time. By contrast, simultaneous preservation can reduce the overhead. In 1DMA-2, using a single transfer command to preserve an adjacent job output and footprint pair reduces the inference time by 56% and 44% against the baseline, respectively under 100μF and 1mF capacitance. The reductions are respectively 80% and 74% in 1DMA-ALL, which uses a single transfer command to preserve all job outputs and footprints in *contiguous* NVM locations.

Although the accelerator computations in 1DMA-2 and 1DMA-ALL have doubled against the baseline, the additional computation overhead can be traded for a reduction in data transfer overhead to significantly reduce the inference time, due to two key reasons. Firstly, the latency required by the accelerator to compute a job is typically much shorter than the latency required to preserve a job output into NVM, and the latency reduction is more apparent when such a latency gap between the accelerator and NVM is larger[1]. Secondly, multiple data blocks per transfer command efficiently utilize the NVM bandwidth and reduce energy consumption. Note that, unlike continuously-powered inference, intermittently-powered inference typically incurs a high NVM utilization due to progress preservation. From the experiment, we draw a useful insight that inspires our approach. Progress preservation overhead can be amortized if progress indicator (footprint) preservation is *piggybacked* onto job output preservation, provided that they are placed contiguously in the shared VM.

4 JAPARI: JOB AND PROGRESS ALTERNATE INFERENCE

Motivated by the observation in Section 3, we propose *model augmentation*, which adapts a standard, pre-trained DNN model to intermittent power by appending additional computation units such

[1]One job computed by a vector MAC of length $T_n = 8$ incurs 12 clock cycles on the LEA, while using DMA to preserve a 2byte job output to the external NVM incurs 58 cycles.

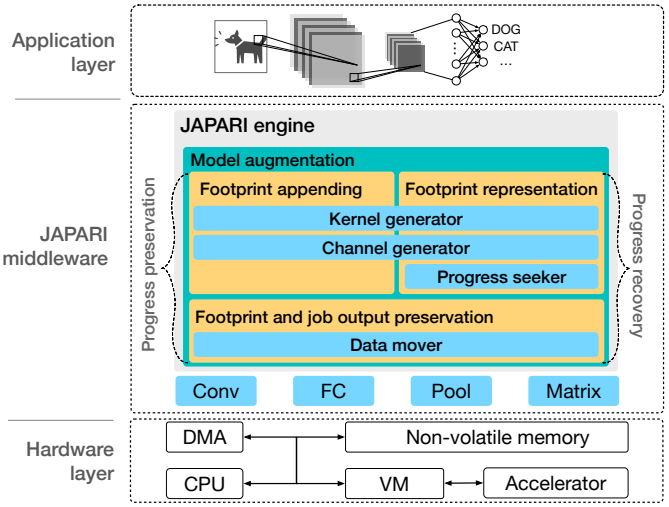


Fig. 6. System architecture with JAPARI middleware

as extra channels and kernels onto the DNN model, thereby intrinsically preserving progress information during inference. This allows data transfers required for progress indicator preservation to be converted into accelerator computations, and thus progress indicator preservation can be piggybacked onto job output preservation. While model augmentation could be generic to different progress indicators [24, 43], we use a footprint to indicate each computed job output because a single footprint is relatively minimal compared with other indicators such as a set of loop indices. As shown in Figure 6, to realize model augmentation, we propose *JAPARI*, a runtime middleware that sits between the application and hardware layers of a system. JAPARI transparently appends additional computation units onto the DNN during inference, manages simultaneous preservation of job outputs and footprints, and ensures correct progress recovery upon power resumption, without involvement from the application. Note that although we carry out model augmentation online, it can also be applied in an offline fashion, before deployment.

4.1 Design Challenges

Two key design challenges need to be addressed to realize model augmentation. The first challenge is to determine the *positions* in the original model architecture to augment in order for the accelerator to generate job outputs and footprints in contiguous VM locations, with the model architecture *uncorrupted*. Here, we mean the original model architecture is unchanged and contained within the augmented model. This is non-trivial to achieve, as the accelerator may not generate the required layout, where job outputs and footprints are placed in an alternate manner, if new computation units are invalidly added into the model. Also, incorrect model augmentation can affect the model architecture. The second challenge is to assign the *values* of the computation units so that the latest progress can be derived based on the footprints preserved in NVM, with the model accuracy *unaffected*. Here, we mean the OFMs of the original model are unchanged and contained within the OFMs generated by the augmented model. Due to model augmentation, job outputs and footprints are no longer preserved separately, but in the same contiguous NVM region which may be reused by prior operations and layers on resource constrained devices. Therefore, identifying the latest footprint becomes particularly difficult, as old footprints preserved in the same NVM region may obfuscate the new ones. Also, incorrect value assignment can affect the model accuracy.

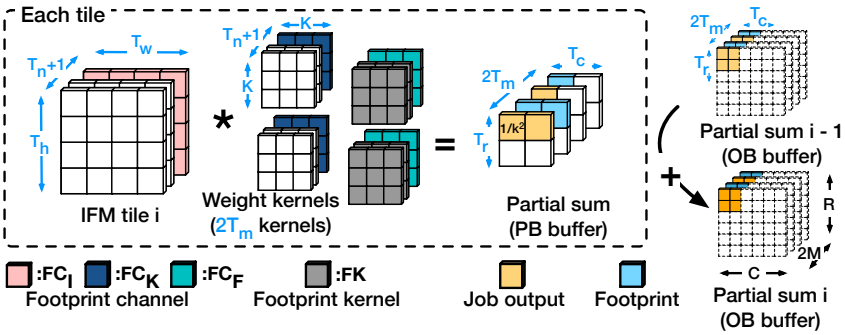


Fig. 7. Footprint kernels and channels appended onto a tile

The first design challenge of where in the model to augment is addressed by *footprint appending*. As shown in Figure 7, footprint kernels and footprint channels are appended into specific positions in the tiled input data, without corrupting the model architecture. This allows the accelerator to intrinsically generate and place job outputs and footprints in VM in an alternate manner during the convolution process (Section 2.1), making simultaneous preservation possible. Footprint appending increases the number of accelerator computations, but the additional computation overhead is compensated by the reduced transfer overhead (Section 3).

The second challenge of how to identify the interrupted job is addressed by *footprint representation*. Footprint kernels and channels are assigned specific values, without affecting the model accuracy. This ensures that the latest footprint can be distinctively identified from previously preserved footprints and used to derive the interrupted job, making correct recovery possible. Note that as currently computed partial sums are accumulated onto those previously computed and preserved in the OFM buffer (Section 2.1), we cannot reset the buffer to zero before a new operation, nor can we implicitly determine the interrupted job by the position of the last non-zero value in the buffer.

4.2 JAPARI Architecture

Our *JAPARI* middleware performs model augmentation at runtime. Figure 6 shows the typical software architecture of an embedded system, where the application layer defines the DNN model structure, and the hardware layer contains the CPU, hardware accelerator, DMA controller, VM and NVM. The accelerator uses a shared VM region with the CPU. The inference input data (obtained via sensors at runtime), model weight parameters, and preservation buffer are located in NVM. Our JAPARI middleware transparently manages progress preservation and recovery, without involvement from the application layer. The JAPARI engine includes four key components: (1) *progress seeker*, (2) *kernel generator*, (3) *channel generator*, and (4) *data mover*. These components are invoked by the JAPARI library functions during the respective layer computation (e.g., CONV, FC and POOL) to integrate footprints into the inference process. The role of each key component is explained as follows.

An end-to-end inference is computed layer by layer on a tiled basis, where an IFM tile, a weight kernel tile and a partially computed OFM tile are fetched from NVM to VM by the data mover to accumulatively compute an OFM tile. Subsequently, the kernel and channel generators automatically append footprints kernels and channels onto the IFM and weight kernel tiles, before accelerator operations are invoked. To accommodate the VM space required for footprint kernels and channels, when fetching tiled input data, the data mover ensures appropriately sized memory regions are unoccupied after the fetched IFM tile and between the weight kernels. The specific positions of

the appended footprint kernels and channels (Section 4.3) ensure job output and footprint pairs, produced by the accelerator, are placed contiguously in the shared VM. The data mover then preserves the job outputs and footprints of the same accelerator operation to the preservation buffer in NVM using one single DMA transfer command, in *parallel* to the accelerator computing the subsequent jobs in a pipelined fashion. Layers are processed sequentially, where one layer's output is the next layer's input. Therefore, during the computation of the current layer, footprints generated for the previous layer are *excluded* when tiled input data are fetched.

A tile computation may be interrupted due to power failure. Upon power resumption, the progress seeker searches for the latest footprint in the current layer from all preserved footprints in the preservation buffer in NVM. Footprint kernels and channels are assigned specific values (Section 4.4), such that the latest footprint in the current layer can be distinctively identified from prior footprints paired with job outputs, even though the preservation buffer is reused for different operations, tiles, and layers. Hence, once the latest footprint has been retrieved, it is used to derive the interrupted job in the current layer. The accelerator is then configured to resume inference from the interrupted job. Only partial IFM, weight kernel, and OFM tiles are re-fetched by the data mover to complete the *remaining* jobs of the interrupted tile.

Currently, JAPARI has incorporated footprint appending and footprint representation to the convolution (CONV), fully connected (FC) and pooling (POOL) layers, which account for a majority of the computation in DNNs. We primarily focus on the CONV layer in Sections 4.3 and 4.4, and then discuss how the proposed footprint appending and representation are applied to FC and POOL layers in Section 4.5.1. The JAPARI engine and library can be extended to support other layer types such as depthwise and pointwise convolution.

4.3 Footprint Appending for Progress Preservation

4.3.1 Footprint-related Positions: The positions of the appended footprint kernels and channels ensure that each job output is paired with a footprint indicating the inference progress. The output granularity varies across accelerators, depending on the size of their internal registers. We first consider generic accelerators with *fine-grained* outputs and basic support for vector/matrix math operations, such as those typically found in lightweight systems. Support for accelerators with coarse-grained outputs will be discussed in a subsequent section.

As shown in Figure 7, for tiled input data fetched into VM, one footprint kernel (denoted FK) is appended to each weight kernel. Then, one footprint channel is appended to the IFM tile, each weight kernel, and each footprint kernel, where the appended channels are respectively denoted as FC_I, FC_K and FC_F. The dimensions of each FK match the original weight kernel (i.e., $|FK| = T_n \times K \times K$), the dimensions of FC_I match one channel of the fetched IFM tile (i.e., $|FC_I| = T_w \times T_h$) and the dimensions of FC_K and FC_F match the kernel spatial dimensions (i.e., $|FC_K| = |FC_F| = K \times K$). Accordingly, an *augmented* OFM tile comprises $2T_m$ channels with spatial dimensions $T_c \times T_r$. Note that each OFM channel will only contain either job outputs or footprints, making the original model architecture uncorrupted.

To allow job outputs and footprints to be produced in the shared VM in an alternate manner, an augmented OFM tile with dimensions $2T_m \times T_c \times T_r$ is computed in *channel-major* order (Figure 7). On generic accelerators, intra-tile computation requires multiple operations of two types. The first type of operation, denoted `psum`, performs $2T_m$ vector MACs (each of which computes the dot product of a kernel vector and an IFM vector in the input channel dimension) to produce $2T_m$ partial sums in the output channel dimension, where each pair of adjacent partial sums represent a job output and a footprint with interleaved positions. To produce all partial sums in the augmented OFM tile, `psum` needs to be invoked $T_c \times T_r$ times. Then, the second operation type, denoted `acum`, performs a vector addition, where the currently computed $2T_m$ partial sums in the channel dimension are

respectively accumulated onto those previously computed counterparts. Symmetrically, acum is also invoked $T_c \times T_r$ times. As each kernel contains $K \times K$ vectors, to complete the augmented OFM tile, the above process needs to be repeated $K \times K$ times. Overall, for fine-grained accelerators, an augmented OFM tile requires up to $2T_c \times T_r \times K \times K$ operations to compute, where their computation order can vary depending on the intra-tile execution loops (Figure 1).

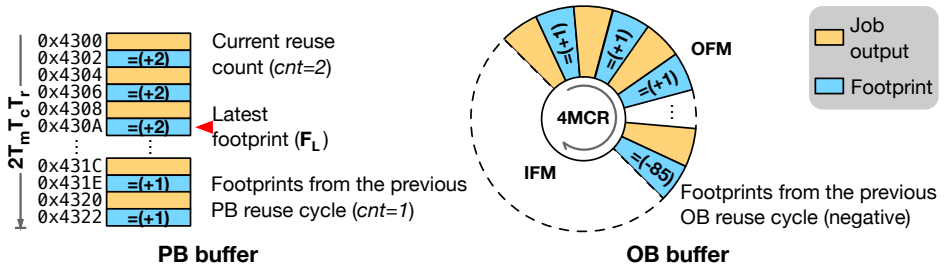


Fig. 8. Layout of preserved job outputs and footprints

To preserve inference progress despite power instability, a partial sum buffer (denoted PB) and an OFM buffer (denoted OB) in NVM are respectively used to preserve the outputs of the psum and acum operations. As illustrated in Figure 8, the size of PB is $2T_m \times T_c \times T_r$, which is sufficient to preserve the currently computed OFM tile, and its layout directly follows the same interleaved layout in the shared VM. As one layer's output is the next layer's input, OB is a *circular* buffer with size $4M \times C \times R$, which is sufficient to accommodate both the augmented OFMs and IFMs of the currently computed layer (Figure 1). In the circular buffer, the OFMs are preserved next to the IFMs, to prevent the IFMs from being overwritten by currently computed OFM tiles. The augmented OFMs are computed by successively computing augmented OFM tiles, where the computation order can vary depending on the inter-tile execution loops selected to maximize data reuse (Figure 1). To reduce memory usage on edge devices, all layers reuse the same PB and OB, and thus their sizes must be applicable to any layer (i.e., with the largest $T_m \times T_c \times T_r$ and $M \times C \times R$) in the DNN. Crucially, they may contain job outputs and footprints from the current layer as well as from a prior layer. Note that for dedicated convolution accelerators, with psum and acum merged into one operation, its job outputs and footprints will directly be preserved in OB, and PB will not be required.

4.3.2 Computation and Preservation Overhead: Compared with HAWAII, JAPARI invokes the same number of accelerator operations but each operation contains more sub-operations. The computation overhead mainly increases with the number of appended footprint kernels (*FK*s). The required number of *FK*s depends on the *granularity* (J_i) of an accelerator output, which is related to the level of computation parallelism and the size of the internal memory of the accelerator. With the maximum number of *FK*s, the number of required sub-operations is doubled, and each fine-grained job output is paired with a footprint. As illustrated in Figure 9, for an accelerator with a *fine-grained* output (i.e., P0 case, where J_i is one partial sum), one *FK* is appended to each weight kernel, resulting in T_m footprint kernels, as described in Section 4.3.1. Conversely, for a *coarse-grained* output (i.e., P1-P5 case, where J_i is a vector/matrix of partial sums), only one *FK* is appended to the last weight kernel to preserve the whole operation progress, and thus the computation overhead is relatively lower. Furthermore, a *batch* of B job outputs can be paired with a footprint, where B is between 1 and the total number of jobs in an invoked operation. Then, only one *FK* is appended to every B weight kernels. Figure 9 (bottom), shows an example where $B = 3$ and J_i is P0, resulting in $\lceil \frac{T_m}{B} \rceil$ appended *FK*s. A larger batch size B can reduce the number of

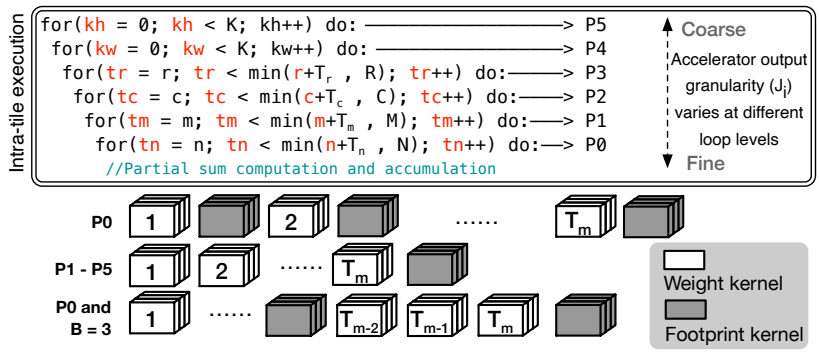


Fig. 9. Different output granularities and batch sizes

appended *FK*s, thereby decreasing the computation overhead, but at the cost of increased progress lost upon power failure.

Compared with HAWAII, JAPRAI invokes fewer transfer commands to preserve an equal number of job output and footprint pairs. The preservation overhead is dominated by the number of invoked DMA transfer commands. To carry out progress preservation in parallel to accelerator computation, the data mover is invoked alongside an accelerator operation to preserve all job outputs and footprints in VM to NVM using a single transfer command. The number of required transfer commands depends on the *latency difference* between the accelerator computation and NVM access on the target platform. As shown in Section 3, if the computation latency is lower than the preservation latency, then only a single transfer command is required to preserve all job outputs and footprints of the same operation, reducing the total number of $2{\times}T_m$ transfer commands required for separate preservation to 1. For the inverse case, a new transfer command is required to preserve every pair of job output and footprint, but the number of transfer commands required is still smaller than if every job output and its associated footprint were preserved separately [43].

4.4 Footprint Representation for Progress Recovery

4.4.1 Footprint-related Values: The values of the appended footprint kernels and channels ensure that the latest footprint (denoted F_L) generated for the current layer can be distinctively identified from previously preserved footprints. There are three requirements for the value assignment. First, despite those appended channels and kernels involved during convolution computation, the model accuracy should be unaffected. Second, as the PB and OB buffers are reused within each layer (Section 4.3), where old footprints will be overwritten by new ones, footprints generated for the same layer must be distinguishable from one another to uniquely identify the latest footprint within the layer. Third, as the two buffers are also reused across layers, where old footprints may obfuscate the new ones, footprints generated for the current layer must be distinguishable from those preserved for prior layers.

Figure 10 illustrates the value assignment scheme used. We assign the first element of the channel appended to each footprint kernel (denoted FC_F^0) to $cnt{\times}b$, the first $T_c{\times}T_r$ elements of the channel appended to the IFM tile to 1, and all remaining elements of the footprint kernels and channels are assigned 0, which results in all generated footprints only taking on the value of $FC_F^0 = cnt{\times}b$ after partial sum computation. Note that because those elements of the appended footprint kernels that convolve on the original IFM tile are assigned 0, the generated job outputs will only take on the

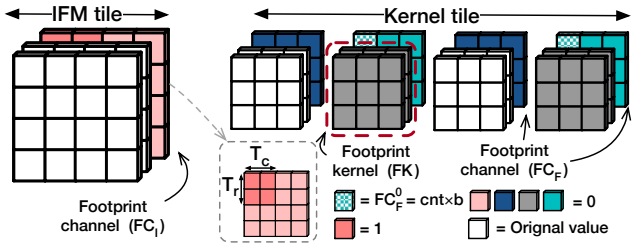


Fig. 10. Values of footprint kernels and footprint channels

values computed based on the original IFM tile and weight kernels, and thus the first requirement of unaffected model accuracy is satisfied.

Here, *cnt* is a counter variable in VM, used to uniquely distinguish footprints generated within the same layer (i.e., the second requirement). At the start of each layer, *cnt* is set to 1 and then incremented before each time a position in PB is reused (i.e., an old footprint is overwritten), which depends on the intra-tile execution loops (Figure 1). For example, in Figure 7, where a psum operation comprises $2T_m$ sub-operations and the PB size is $2T_m \times T_c \times T_r$, the outputs of $T_c \times T_r$ psum invocations can be preserved in a single reuse cycle, and in such a case, each position in PB is reused $K \times K$ times and thus *cnt* is incremented by $K \times K$ after an augmented OFM tile is completed. Because footprints preserved onto the same PB position increase in their absolute values, while those having the same value are preserved onto different positions, they are uniquely distinguishable. The footprints preserved in OB are accumulated by those counterparts in PB (Section 4.3.1), and thus they also satisfy the aforementioned second requirement.

Finally, the control variable $b = \{+1 \text{ or } -1\}$ in VM is used to satisfy the third requirement. At the start of each layer, the PB buffer is reset to 0 (using one single DMA command). Thus, the footprints preserved into PB for the current layer will never be obfuscated by old footprints preserved for prior layers. However, we do not reset the OB buffer to 0 before a new layer, because OB is orders of magnitude larger than PB in size and contains the IFMs required for the current layer. Instead, we use b to indicate two adjacent reuse cycles by inverting its sign whenever OB is fully reused (i.e., all footprints have the same sign or, simplistically, the first and last footprints have the same sign). Consequently, an old footprint can only be overwritten by a new footprint with an inverted sign. Within the NVM region allocated to the OFMs of the current layer, footprints preserved respectively for the current layer and for prior layers (if any exist) must have different signs and thus are distinguishable.

4.4.2 Progress Search and Recovery: Upon power resumption, the progress seeker is invoked to correctly recover the power-interrupted inference. Specifically, the objective of the progress seeker is to determine three data items, namely the interrupted layer, the interrupted operation type, and the interrupted job within the layer. Figure 11 shows the control flow of the progress recovery process. First, the index of the interrupted layer (L_i) is retrieved from NVM, where L_i is directly tracked in NVM as a counter variable and is incremented by 1 before a new layer. Then, the interrupted operation type (*opt*) is determined, where *opt* is indicated by the PB buffer usage status (i.e., fully reused or not) upon power resumption. For each PB reuse cycle, if PB is fully reused (i.e., all footprints increase in their absolute values or, simplistically, the absolute value of the last footprint is not smaller than that of the first footprint), it indicates all psum operations successfully completed but a subsequent acum operation was interrupted (i.e., *opt* = acum) due to power loss. Inversely, if PB is not fully reused, it indicates a psum operation was interrupted (i.e., *opt* = psum).

Next, in order to derive the interrupted job, the latest footprint (F_L) in the current layer is searched and retrieved from NVM, where the buffer to search depends on *opt*. If *opt* = acum, then OB is searched, else PB is searched instead. Furthermore, only the buffer region that contains footprints of the currently computed layer is searched, using binary search, where the search latency increases logarithmically with the region size. As the generated footprints are always increasing in their absolute values (Section 4.4.1), the search criteria is essentially the last largest or last smallest footprint in the searched buffer, depending on the control variable b (+1 or −1) currently used. Note that as the same b is used in each OB reuse cycle, the currently used b is implicitly indicated by the first footprint in OB.

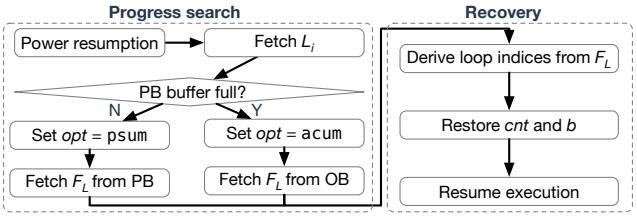


Fig. 11. Progress search and recovery

As shown in Figure 11, once F_L is retrieved, it is used to derive the *loop indices* (i.e., r, c, m, n, kh, kw, tr, tc, tm, and tn in Figure 1) of the interrupted job. As the inference process is unidirectional and each layer is computed by sequential operations, the structure of the inter/intra-tile execution loops, including their bounds and order, are deterministic and known at design time. Furthermore, as each job progress is indicated by a footprint, which is uniquely distinguishable by its absolute value and position in the buffer region (Section 4.4.1), the interrupted job can be derived based on the value and position of the latest footprint F_L. Take the intra-tile computation in Figure 7 for example, where the *cnt* variable is incremented by $K{\times}K$ after the augmented OFM tile, and all the $T_m{\times}T_c{\times}T_r$ footprints in each PB reuse cycle share the same *cnt* value. The loop indices kw and kh can be respectively derived as $kw = |F_L|\%K$ and $kh = \frac{|F_L|\%(K\times K)}{K}$, given the footprint value $|F_L|$. Similarly, based on the footprint position $\hat{F_L}$ in PB, the loop indices tm, tc, and tr can be respectively derived as $tm = \hat{F_L}\%T_m$, $tc = \frac{\hat{F_L}\%(T_m\times T_c)}{T_m}$, and $tr = \frac{\hat{F_L}}{T_m\times T_c}$. Note that the inner-most tn loop is carried out with an *atomic* sub-operation to compute a job in the used psum operation.

Lastly, the inference execution can *resume* from the interrupted job. To ensure that the footprint values of the remaining jobs correctly continue to increase or decrease, the *cnt* and b variables introduced for footprint representation (Section 4.4.1) are respectively restored as the absolute value and the sign of the last footprint (i.e., $cnt = |F_L|$ and $b = \{+1 \text{ or } -1\}$). The tiled input data required for the remaining jobs are then fetched to VM by the data mover (Section 4.2) and, subsequently, the accelerator is appropriately configured and initialized with the derived loop indices to resume the interrupted *opt* operation and proceed to complete the interrupted tile (Section 4.3.1).

4.5 Generality

4.5.1 Support for Other Layer Types: A *Fully-connected* (FC) layer with an $N{\times}W{\times}H$ input and M output neurons can be equivalently formulated as a CONV layer with M kernels each $N{\times}W{\times}H$ (i.e., the kernel dimensions match the IFM dimensions) [50], and therefore footprints are appended in the same way as CONV (Section 4.3). Figure 12 (left) illustrates an FC layer implemented as a matrix multiply operation, and footprint kernels and channels are appended as described in Section 4.3. A

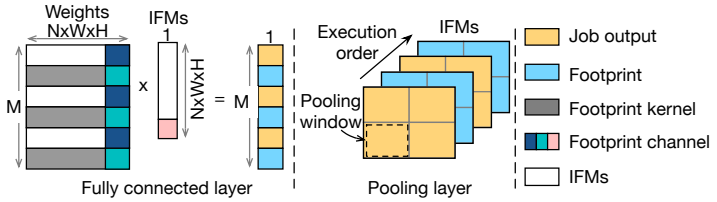


Fig. 12. Footprint appending for FC and POOL layers

pooling (POOL) layer typically obtains input from a CONV layer and already contains footprints as shown in Figure 12(right). Each IFM channel is grouped according to the pooling window size and processed along the channel dimension, where a job is a pooling window computation (executed by the CPU if unsupported by the accelerator) and the job output is preserved onto OB, with PB unused. Upon power resumption, as each pooling window is indicated by a footprint, the position of F_L in OB is used to derive the interrupted job and recover the POOL layer computation.

JAPARI can also be extended to support *depthwise* and *pointwise* convolution, used in models designed for resource constrained systems [32]. Appending footprints for pointwise convolution is the same as the process for CONV with kernel dimensions (N×1×1). For depthwise convolution, a footprint kernel (1×K×K) is appended to each weight kernel and, subsequently, a footprint channel (K×K) is appended to each kernel, as well as a footprint channel (W×H) is appended to each IFM channel. It is worth noting that the aforementioned layer types (and their variants) cover a majority of those typically used in tiny machine learning applications [9]. Model augmentation opens up an avenue for DNN models to adapt to intermittent computing. Future efforts may further simplify JAPARI or propose compatible model augmentation techniques for other newer layer types targeting such lightweight applications.

4.5.2 Support for Other Accelerator Types: An augmented DNN model can be executed in exactly the same manner as a standard DNN model. This is because fundamentally, model augmentation simply increases the number of computation units in a model without corrupting the original network architecture, and also because JAPARI assumes a generic tile-based inference execution model that can be implemented on a majority of lightweight embedded platforms. However, to allow for simultaneous preservation of job outputs and footprints, an important requirement is that the CPU or accelerator, when computing an augmented tile, outputs *interleaved* job outputs and footprints, either directly to persistent NVM or to a VM region accessible by the application layer. Therefore, JAPARI requires the augmented OFM tile to be computed in the *channel-major* order (i.e., with respect to the inter and intra-tile loop orders).

Off-the-shelf MCUs are a popular choice for intelligent edge systems due to their generality and ease of programmability, and to support tiny ML applications, recent MCUs have started to include lightweight computational accelerators [16, 35, 39, 64, 67]. Such accelerators efficiently perform common vector and matrix computations, and place the job outputs into a shared VM region that is accessible by the CPU. Moreover, there are also a wide range of conventional off-the-shelf MCUs that have CPUs with specialized instructions (e.g., ARM Cortex-M4 [7] and RISC-V [23]), which are used to accelerate the multiply accumulate operations required for DNN inference. For such systems, the CPU can preserve the job outputs that may reside in registers or VM to NVM. JAPARI directly supports all aforementioned types of off-the-shelf MCUs, as they can be programmed to process a tile in any computation order.

Early FPGA-based custom accelerators focused on exploiting data reuse [11, 13, 21, 59, 73, 74], resulting in accelerator designs optimized for a particular computation order. In order for JAPARI to support such accelerators that do not follow the channel-major computation order, the footprint appending (Section 4.3) and footprint representation (Section 4.4) schemes need to be appropriately adapted to the used order. Recent custom accelerators have focused on new strategies such as exploiting model sparsity [26, 48] and variable bitwidths [61, 62] to further improve inference performance and energy efficiency. Accelerators that exploit model sparsity rely on a special irregular data format to represent sparse matrices. To support such accelerators, JAPARI can be adapted to use a dynamic footprint representation scheme, where the values assigned to the footprint kernels and channels change based on the sparsity of the weight kernels and IFMs, at runtime. Variable bitwidth accelerators may encounter integer overflow during footprint representation, and to address this, JAPARI divides and processes a layer as multiple sub-layers (Section 5.1).

Using our JAPARI middleware, AI programmers can easily deploy their DNN models on common MCUs or FPGAs supporting flexible computation orders, to achieve high inference performance under intermittent power. However, it is worth noting that custom hardware accelerators could also be designed to explicitly track their computation progress across power cycles, and such solutions would be valuable for intermittent inference provided that the sales volume is sufficient to amortize the associated engineering cost [19].

5 JAPARI IMPLEMENTATION

We realized JAPARI on the Texas Instruments (TI) MSP430FR5994 [36], with a TI Low Energy Accelerator (LEA) and 8KB SRAM (4KB VM shared between the CPU and LEA). To support reasonably large DNN models we use 1MB external NVM (Cypress CY15B108Q serial FRAM [1]). A custom LEA driver was also developed to efficiently use the DMA and operate LEA asynchronously.

5.1 Accelerated Tile Computation

Due to the limited VM space, a $K\times K$ kernel convolution is computed as the sum of K^2 separate 1×1 convolutions [5], with a stride of 1. Accordingly, an IFM tile ($T_n\times T_w\times T_h$), a kernel tile ($T_m\times T_n\times 1\times 1$) and a partially computed OFM tile ($T_m\times T_c\times T_r$) are fetched for the four inner-most intra-tile loops (Figure 1) and passed onto the LEA for computation. This process is repeated $K\times K$ times to compute an (augmented) OFM tile. The absolute values (*cnt*) of footprints continuously increase within a layer and therefore may suffer from an *integer overflow* on the 16-bit MCU. To address this, JAPARI divides a large layer into multiple sub-layers, tracks their indices in NVM, and resets the *cnt* value and *PB* buffer at the start of each sub-layer as if it were a new layer (Section 4.4). The LEA commands `LEACMD__MPYMATRIXROW` (vector-matrix multiply) and `LEACMD__ADDMATRIX` (vector addition) are respectively used to implement the `psum` and `acum` operations (Section 4.3.1).

To enable the tiled input data to fit in VM, the tile parameters that respectively define the spatial size and output channel size were set as $T_h = T_w = T_r = T_c = 4$ and $T_m = 8$. The job computation latency for `psum` on the LEA is calculated as $\frac{3}{2}\times(T_n + 1)$ clock cycles [34], and the job preservation latency on the external NVM is calculated as DMA invocation overhead plus NVM write overhead, i.e., $2+16 = 18$ clock cycles [1, 37]. An accelerator invocation takes 16 clock cycles and an NVM invocation takes 40 clock cycles. These two are omitted in the above calculation as JAPARI incurs them only once at the start of accelerator computation and output preservation, respectively. To preserve all accelerator outputs of the same operation using only a single transfer command, the job computation latency must be less than the job preservation latency (Section 4.3.2); therefore, T_n was set to 8, as $\frac{3}{2}\times(8 + 1) < 18$.

5.2 Accelerator Output Preservation

The external NVM is interfaced via SPI (serial peripheral interface) at the same clock speed as the MCU (16 MHz). The SPI peripheral on the MCU was configured to cooperate with the DMA controller to transmit/receive data to/from the external NVM one *byte* at a time, without CPU intervention. For accelerator output preservation, the DMA controller copies a byte of the output data from VM to the SPI transmit buffer, and the SPI peripheral transmits the byte of data to the external NVM and notifies the DMA controller to copy the next byte. This process is repeated until all accelerator outputs are transferred, whereupon the DMA controller notifies the CPU.

An accelerator output (i.e., a job output or footprint) is 2 bytes on the LEA. As power may fail during the byte-wise data transfer, we preserve the byte containing the sign bit of a footprint last, ensuring the incompletely preserved footprint will not be incorrectly identified as the latest footprint (Section 4.4.2). To protect *idempotence* (i.e., the same outcome will be produced despite repeated execution) during partial sum accumulation and accelerator output preservation (Section 4.3.1), the JAPARI implementation uses a *double buffering* mechanism similar to that in [24, 54], where two `OB` buffers are alternately used. JAPARI ensures an operation invocation reads from and writes to the counterpart positions in different `OB` buffers, thereby avoiding *write-after-read* dependencies.

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

Table 1. Specifications of the experimental platform

Hardware	
MCU	TI MSP430FR5994
Volatile memory	8KB SRAM
Non-volatile memory	Cypress CY15B108QI 1MB FRAM
Accelerator	TI Low-Energy Accelerator
Energy	
Boost converter	TI BQ25504
Switch on/off voltage	2.8 V / 2.4 V
Capacitance	100 μF and 1 mF
Continuous power	1.65 W = 3.3 V $\times$ 0.5 A
Intermittent power	3 mW = 1 V $\times$ 3 mA

Our experimental environment is summarized in Table 1. We evaluated JAPARI on the TI MSP430FR5994 with the LEA, internal VM, and external NVM, as described in Section 5. Intermittent execution was emulated using a Keithley 2280S power supply, a BQ25504 energy management unit, and a capacitor (energy buffer). A power source of 3 mW (1 V, 3 mA) was used, representative of the power output of a 6cm^2 sized solar cell [17] under indoor light, which is insufficient to complete a single inference in one power cycle. Experiments were conducted under intermittent power, using different capacitor sizes (100 μF and 1 mF), and under continuous power (1.65 W). Small capacitors are favorable due to their short recharge time, low energy leakage, and small area [4], but they cause frequent power failures and require more power cycles to complete an inference. A capacitor of 100 μF was selected as it is the smallest size suggested to operate the used energy management unit, as per its specification [38], and the capacitor size of 1 mF is sufficiently large to observe the trade-off between preservation and re-execution overhead under different energy budgets.

Table 2 shows the three DNN models used in our evaluation, namely a convolutional neural network (CNN) used for image classification (denoted ICS) [27], a CNN used for human activity detection (denoted HAR) [31], and a fully connected DNN used for speech keyword spotting

Table 2. DNN models used for performance evaluation

Model	Layers	Job count
CNN for Image Classification (ICS) [27] Dataset: CIFAR-10 [47] Size: 250 KB	CONV × 3 POOL × 2 FC × 2	1.47 M
DNN for Speech Keyword Spotting (SKS) [75] Dataset: Speech commands [70] Size: 400 KB	FC × 4	49.9 K
CNN for Human Activity Recognition (HAR) [31] Dataset: Smartphone sensors [6] Size: 15 KB	CONV × 3 POOL × 3 FC × 1	45 K

(denoted SKS) [75]. These models are representative of those typically used in tiny machine learning applications [9] and fit within our 1MB external NVM. To allow for the models to run on the TI embedded device (without floating-point arithmetic support), they were compressed by quantizing their inputs and weight parameters to a 16-bit fixed point representation (Q15.1 format) from the 32-bit floating point representation used during training, without a significant loss of accuracy. All models use dense convolution with a stride of 1 and no zero-padding. Each model was repeatedly executed for 100 runs (end-to-end inferences), which is sufficient to mitigate experimental variances while reproducing the results.

JAPARI is compared against TL-A and HAWAII[2]. TL-A is a task-based intermittent inference approach with support for acceleration, similar in concept to TAILS [24]. TL-A performs a one-time calibration to find a task size that fits the energy budget. HAWAII [43] is a state-of-the-art footprint-based approach, which separately preserves job outputs and footprints, to enable hardware accelerated intermittent inference. The progress preservation granularity of HAWAII and JAPARI varies based on the batch size B (Section 4.3.2), which ranges between one sub-operation (1 job) and one operation (T_m jobs), and that of TL-A varies based on the task size T, which ranges between one operation and one layer, because task partitioning cannot be performed at the sub-operation granularity (in that peripheral sub-operations cannot be directly invoked in an application).

JAPARI is evaluated in terms of *runtime overhead* and *inference time*. In our first experiment, we evaluate the runtime overhead incurred by JAPARI to preserve inference progress in comparison with the baselines, and therefore we measure the energy consumption to complete one end-to-end inference under continuous power. The TI EnergyTrace tool [33] was used to measure energy consumption. As JAPARI is developed for intermittent inference, in our second experiment, we measure inference time under intermittent power, which is defined as the average time required to compute one end-to-end inference. Lastly, we conduct a breakdown analysis of model augmentation overhead, in terms of computations and data transfers, which allows us to explore opportunities for further improvement.

6.2 Runtime Overhead

Figure 13 shows the runtime overhead of each approach, across all evaluated DNN models and different progress preservation granularities (batch size B in HAWAII and JAPARI or task size T in TL-A), in terms of the energy consumption required to complete a single end-to-end inference under continuous power. Results are shown for the preservation granularity of one job, half an

[2]HAWAII was originally implemented using 1D convolution [44]. For a fair comparison, we implement HAWAII and TL-A using tiled convolution with vector-matrix multiply, similar to JAPARI, as described in Section 5.1. Moreover, to avoid an incompletely preserved progress indicator, as discussed in Section 5.2, they use a *shadowing* mechanism similar to that in [12] to *atomically* preserve every indicator (a set of loop indices in TL-A or a footprint in HAWAII) into NVM.

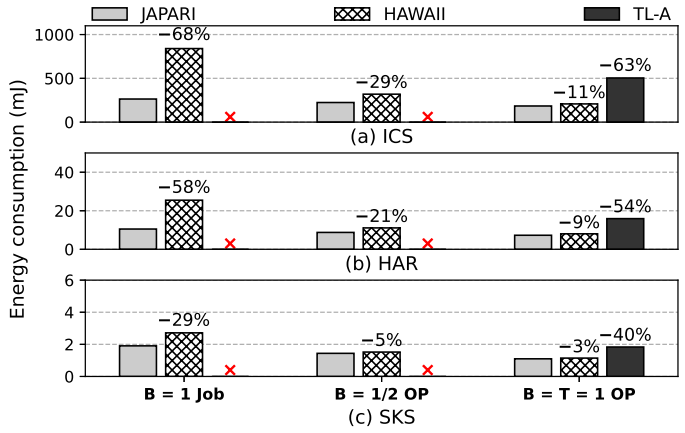


Fig. 13. Runtime overhead under continuous power

operation, and one operation (8 jobs). The relative energy consumption reductions of JAPARI against the baselines are indicated above the bar plots. As TL-A has a minimum T of one operation, TL-A cannot be evaluated when the preservation granularity is finer than one operation. Under the same preservation granularity ($B = T$), TL-A requires higher energy consumption than JAPARI and HAWAII, because the convolution inter/intra-tile loop indices are incremented and preserved in NVM at the task boundary, which is more costly than preserving a single footprint. Moreover, unlike HAWAII and JAPARI which preserve job outputs in parallel to job computations, TL-A preserves all job outputs only after the completion of the entire operation. TL-A also has extra task management overhead (e.g., task creation and transition), where the overhead can increase dramatically as the task size is reduced [54].

HAWAII's energy consumption increases drastically as B decreases, because of the extra data transfer overhead related to separate job output and footprint preservation, whereas JAPARI's energy consumption shows a linear yet moderate increase with respect to B, due to the reduction of data transfer overhead brought by simultaneous preservation (Section 4.3.2). The energy consumption reduced by JAPARI is more evident in the ICS and HAR models, which largely comprise CONV layers (Table 2) and therefore are *heavily accelerated*. Such models incur higher data transfer overhead related to a larger number of jobs, but JAPARI trades additional computations for a reduction in data transfer overhead. Moreover, as CONV layers typically have fewer weight parameters and more weight reuse than FC layers, JAPARI incurs lower data access and footprint appending overhead.

Overall, JAPARI shows the lowest energy consumption among all evaluated approaches, in all scenarios. JAPARI consumes 40% to 63% less energy than TL-A, and 3% to 68% less energy than HAWAII, depending on the DNN model and progress preservation granularity.

6.3 Inference Time

Figure 14 shows the inference time under intermittent power for each evaluated approach, under different DNN models, energy buffer capacities, and progress preservation granularities. The relative inference time reductions of JAPARI against the baselines are indicated above the bar plots. JAPARI shows a shorter inference time than TL-A, which is mainly because the latter incurs task re-execution overhead when the unpreserved progress is lost upon power failure. TL-A dynamically

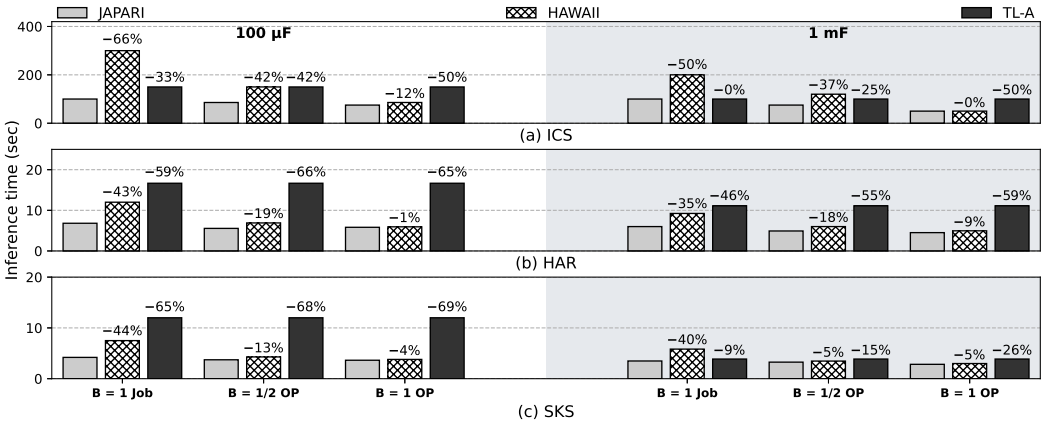


Fig. 14. Inference time under intermittent power

calibrates the task size (T) at the start of each experiment run, based on the energy budget. Re-execution overhead typically increases with T, but factors such as supply voltage variations and current leakage can vary the available energy budget of a capacitor from run to run [2, 53].

JAPARI also shows a shorter inference time than HAWAII. The main reason is that the latter incurs higher preservation overhead than the former (Section 6.2). JAPARI more obviously outperforms HAWAII under the ICS and HAR models than under the SKS model, because HAWAII incurs higher preservation overhead for the former models, as discussed in Section 6.2. The inference time reductions of JAPARI over HAWAII are more obvious when the preservation overhead of HAWAII is considerably higher under a finer preservation granularity and especially dominant under a smaller capacitor. Note that using a fine preservation granularity may incur unnecessary preservation overhead under a relatively large capacitor yet, contrarily, a coarse preservation granularity would suffer from large progress loss if the capacitor is relatively small. There is a tradeoff between preservation overhead and re-execution overhead (Section 4.3.2).

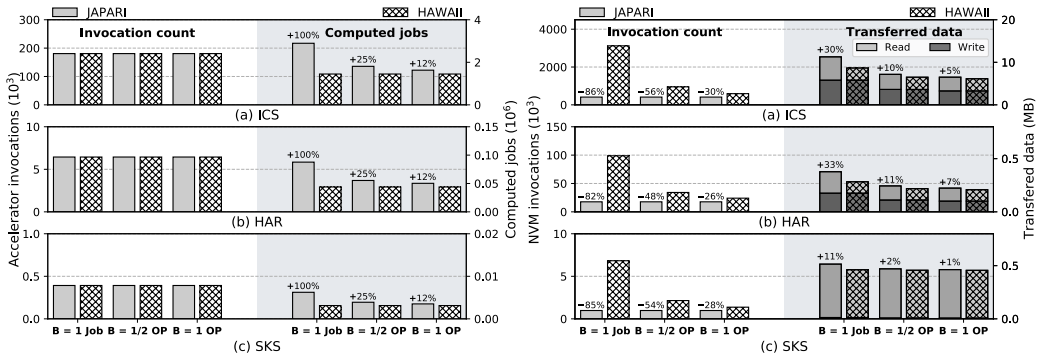
Overall, JAPARI shows the shortest inference time among all evaluated approaches, in all scenarios. Compared with TL-A and HAWAII, JAPARI respectively reduces the inference time by 42% and 25% on average. JAPARI substantially improves intermittent inference for common neural networks used in lightweight IoT applications [9], especially convolutional architectures that typically require heavy acceleration, where a speedup of 3 times is achievable when fine-grained progress preservation is used under a small capacitor.

6.4 Breakdown Analysis

In this section we first breakdown the computation and data transfer overhead of intermittent inference under a frequent power failure condition (i.e., using a 100μF capacitor) for JAPARI and HAWAII, across the different DNN models and progress preservation granularities. Next, we report the hardware resource utilization of the platform, in terms of the accelerator and NVM under the same experimental configurations.

6.4.1 Extra Job Computation and Data Transfer: Figure 15 shows the breakdown of the computation and data transfer overhead of both approaches, where the relative overhead increases in JAPARI compared to HAWAII are indicated above the plots. As shown in Figure 15(a), both JAPARI and HAWAII incur an equal number of accelerator invocations because model augmentation does not

corrupt the original model architecture (Section 4.3). Compared with HAWAII, JAPARI incurs more jobs with each accelerator invocation, in order to compute the footprint kernels and channels appended onto the original DNN model. However, an accelerator invocation takes more clock cycles than a job computation (Section 5.1), and therefore the number of accelerator invocations is the dominant factor that impacts the computation overhead.



(a) Accelerator (b) NVM

Fig. 15. Computation and data transfer overhead breakdown

As shown in Figure 15(b), JAPARI incurs a significantly lower number of NVM invocations than HAWAII, because JAPARI preserves all pairs of job outputs and footprints using only a single data transfer command. Both HAWAII and JAPARI write an equal amount of data to NVM, as they compute and preserve the same number of output features and footprints during inference. Interestingly, JAPARI reads more data from NVM than HAWAII because the augmentation carried out on one layer increases the amount of tile input data fetched for the subsequent layer, particularly for networks with low data reuse, such as SKS with primarily FC layers. However, an NVM invocation takes more clock cycles than an NVM write or read (Section 5.1), and therefore the number of NVM invocations is the dominant factor that impacts the NVM data transfer overhead.

Overall, compared with HAWAII, JAPARI can compute at most double the number of jobs (Section 4.3.2), but the additional job computations are compensated by a reduction in the number of NVM invocations by 20% to 86%, which considerably reduces the NVM data transfer overhead, particularly when fine-grained preservation is used. Therefore, as reported in Section 6.3, compared with HAWAII, JAPARI significantly improves the end-to-end inference time, as the NVM data transfer overhead dominates the active inference time, where the system is powered ON.

6.4.2 Hardware Resource Utilization: Figure 16 shows the accelerator and NVM utilization, for JAPARI and HAWAII, across the same experimental conditions as in Section 6.4.1. We define the accelerator and NVM utilization, as the total active time of the respective hardware component, over the total system power ON duration, across multiple power cycles taken to complete an inference. Note that the combined utilization of both components may not equal to 100%, due to parallel computations and data transfers, and CPU-based operations executed during inference.

In all cases, both HAWAII and JAPARI have a very low accelerator utilization (1% to 7%), while the NVM is heavily utilized (70% to 99%), primarily because NVM access is generally slower than accelerator computation, and therefore NVM data transfers related to job output and footprint preservation typically dominate the active inference time. The NVM utilization increases inversely

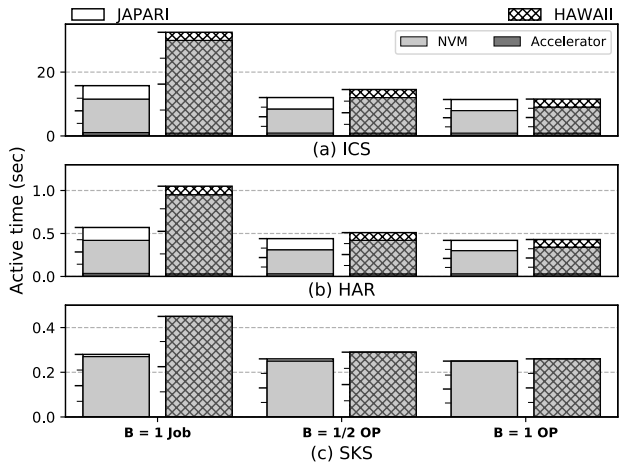


Fig. 16. Hardware resource utilization

with the preservation batch size as more footprints are preserved during inference, but this trend is much less apparent in JAPARI than in HAWAII, as JAPARI significantly reduces the NVM data transfer overhead, as discussed in Sections 6.2 and 6.4.1.

Typically, the NVM is slower than the accelerator as in our experimental platform. However, assuming an uncommon platform, where the NVM data transfer is faster than the accelerator computation, JAPARI may require more data transfer commands for job output and footprint preservation, increasing the NVM utilization and also the inference time. Even in such a case, JAPARI would still have a shorter inference time than HAWAII, because JAPARI preserves a pair of job output and footprint simultaneously using only one data transfer command (Section 4.3.2), thus reducing the progress preservation overhead.

Overall, JAPARI is able to exploit an underutilized accelerator by converting the data transfers required for progress preservation into computations, thereby reducing the NVM utilization. These relative hardware utilization differences, which lead to significant improvements in inference time (Section 6.3), are particularly evident when the preservation granularity is small and under convolutional networks that require heavy acceleration.

7 DISCUSSION

Through extensive experiments, we demonstrate the potential of model augmentation realized via our JAPARI middleware. However, several opportunities remain to further improve the performance of JAPARI. Based on Section 3, the inference time reduction of JAPARI could be even higher than in Section 6.3, but JAPARI incurs additional footprint appending and footprint searching overhead unaccounted for in the former section. VM write overhead is incurred during footprint appending (Section 4.3), which may particularly impact performance for models with large kernel and IFM sizes, and the footprint searching overhead is incurred during system recovery and may increase with the preservation buffer size (Section 4.4).

Our prototype platform (TI MSP430FR5994) also introduces additional overhead when fetching tiled input data. The DMA controller only supports data transfers with a stride of one, and hence footprints from the previous layer can only be excluded after the entire tiled input data is fetched into VM. Due to this reason, JAPARI incurs additional NVM reads to fetch augmented tile input data, as discussed in Section 6.4.1. If the platform supports 2D DMA data transfers (e.g., Cypress

PSoC 62 [18]), then we can fetch only the required output features, excluding footprints, from the previous layer, thus avoiding these additional NVM reads. However, as shown in Section 6.4.2, the inference time is still bounded by NVM data transfers, and therefore there still exists more opportunity to convert more data transfers into accelerator computations.

8 CONCLUDING REMARKS

This paper presents the concept of model augmentation to adapt DNN models to intermittent power, which is orthogonal to existing model compression techniques for adapting DNN models to resource-constrained devices. The JAPARI middleware is developed to realize model augmentation. In contrast to existing intermittent inference approaches, which track progress indicators (loop indices [24] or footprints [43]) independently of the accelerator outputs, JAPARI allows footprints to be intrinsically integrated into the inference process to indicate the accelerator progress, thereby amortizing the overhead required to enable intermittent DNN inference.

Evaluations were conducted on a Texas Instruments device featuring a low-energy accelerator and hybrid memory. By trading extra accelerator computations for a reduction in the invocation of data transfer commands, JAPARI[3] can reduce the energy consumption and significantly shorten the inference time, compared to existing task-based and footprint-based inference approaches [24, 43]. The improvements are more evident for highly accelerated DNN models executed using a finer preservation granularity, under a smaller capacitor.

The JAPARI middleware stack, which transparently performs model augmentation, has been made open [45], facilitating the development of intermittent-aware DNN inference systems that require short inference latency even under frequent power disruptions.

REFERENCES

[1] 2019. Excelon LP 8-Mbit SPI F-RAM. https://www.cypress.com/file/444186/download.

[2] Saad Ahmed, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-powered Computers. In *Proc. of ACM LCTES*. 97–109.

[3] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proc. of ACM LCTES*. 70–81.

[4] Jun Ick Ahn, Daeyong Kim, Rhan Ha, and Hojung Cha. 2021. State-of-Charge Estimation of Supercapacitors in Transiently-powered Sensor Nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), 1–1.

[5] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2017. Low-memory Gemm-based Convolution Algorithms for Deep Neural Networks. *arXiv preprint arXiv:1709.03395* (2017).

[6] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L. Reyes-Ortiz. 2013. A Public Domain Dataset for Human Activity Recognition using Smartphones. https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones.

[7] Arm. 2021. Cortex-M4 processor Mixed-signal MCUs with DSP and FPU instructions. https://www.st.com/en/microcontrollers-microprocessors/stm32g4-series.html.

[8] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE TCAD* 35, 12 (2016), 1968–1980.

[9] Colby R. Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, Danilo Pau, Jae sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav. 2020. Benchmarking TinyML Systems: Challenges and Direction. *arXiv preprint arXiv:2003.04821* (2020).

[10] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2018. Sytare: A Lightweight Kernel for NVRAM-based Transiently-Powered Systems. *IEEE TC* 68, 9 (2018), 1390–1403.

[11] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. In *Proc. of ACM/IEEE ISCA*. 247–257.

[3]Interested readers may refer to a demo at https://youtu.be/Ojsd3NwapQI.

[12] Wei-Ming Chen, Tei-Wei Kuo, and Pi-Cheng Hsiu. 2020. Enabling Failure-Resilient Intermittent Systems Without Runtime Checkpointing. *IEEE TCAD* 39, 12 (2020), 4399–4412.

[13] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *Proc. of IEEE ISSCC*. 262–263.

[14] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. of ACM OOPSLA*. 514–530.

[15] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *Proc. of ACM ASPLOS*. 767–781.

[16] Eta Compute. 2021. ECM3532 Neural Sensor Processor. https://media.digikey.com/pdf/DataSheets/EtaComputePDFs/ECM3532_AI_Sensor_PB_1.0.pdf.

[17] IXYS Corporation. 2010. IXOLAR High Efficiency Solar Cell. http://ixapps.ixys.com/DataSheet/SM111K04L.pdf.

[18] Cypress. 2020. PSoC 62 MCU. https://www.cypress.com/products/32-bit-arm-cortex-m4-cortex-m0-psoc-6.

[19] G De Michell and Rajesh K Gupta. 1997. Hardware/software co-design. *Proc. IEEE* 85, 3 (1997), 349–365.

[20] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-Free Game Boy. *In Proc. of ACM IMWUT* 4, 3, Article 111 (2020), 1-34 pages.

[21] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proc. of ACM/IEEE ISCA*. 92–104.

[22] Igor Fedorov, Ryan P. Adams, Matthew Mattina, and Paul N. Whatmough. 2019. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. In *Proc. of NeurIPS*.

[23] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. 2017. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2700–2713.

[24] Graham Gobieski, Nathan Beckmann, and Brandon Lucia. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proc. of ACM ASPLOS*. 199–213.

[25] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems. In *Proc. of ACM/IEEE MICRO*. 670–684.

[26] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proc, of IEEE/ACM Micro*. 151–165.

[27] Google. 2020. Example CNN to Classify CIFAR-10 using Tensorflow. https://www.tensorflow.org/tutorials/images/cnn.

[28] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-eye: A complete Design Flow for Mapping CNN Onto Embedded Fpga. *IEEE TCAD* 37, 1 (2017), 35–47.

[29] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *Proc. of ICLR*.

[30] Josiah Hester and Jacob Sorber. 2017. New Directions: The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proc. of ACM SenSys*. 1–6.

[31] Burak Himmetoglu. 2017. CNN Model for Human Activity Recognition. https://github.com/healthDataScience/deep-learning-HAR.

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

[33] Texas Instruments. 2014. EnergyTrace Technology for MSP430. http://www.ti.com/tool/ENERGYTRACE.

[34] Texas Instruments. 2016. Benchmarking the Signal Processing Capabilities of the LEA on MSP430 MCUs. http://www.tij.co.jp/jp/lit/an/slaa698b/slaa698b.pdf.

[35] Texas Instruments. 2016. Low-energy Accelerator (LEA). http://www.ti.com/lit/an/slaa720/slaa720.pdf.

[36] Texas Instruments. 2018. MSP430FR5994 MCU. http://www.ti.com/product/MSP430FR5994.

[37] Texas Instruments. 2018. MSP430x5xx and MSP430x6xx - DMA Controller Module. https://www.ti.com/lit/ug/slau395f/slau395f.pdf.

[38] Texas Instruments. 2019. BQ25504 Ultra Low Power Boost Converter with Battery Management for Energy Harvester. http://www.ti.com/product/BQ25504.

[39] Maxim Integrated. 2021. MAX78000 Ultra-Low-Power MCU with Arm Cortex-M4 and a Convolutional Neural Network Accelerator. https://datasheets.maximintegrated.com/en/ds/MAX78000.pdf.

[40] Hrishikesh Jayakumar, Kangwoo Lee, Woo Suk Lee, Arnab Raha, Younghyun Kim, and Vijay Raghunathan. 2014. Powering the Internet of Things. In *Proc. of ACM/IEEE ISLPED*. 375–380.

[41] Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. 2017. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM TECS* 16, 3 (2017), 65:1–65:23.

[42] Weiwen Jiang, Xinyi Zhang, Edwin H.-M. Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Accuracy vs. Efficiency: Achieving Both through FPGA-Implementation Aware Neural Architecture Search. In *Proc. of ACM/IEEE DAC*.
[43] 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 TCAD* 39, 11 (2020), 3479–3491.
[44] Chih-Kai Kang, Hashan Roshantha Mendis, Chun-Han Lin, Ming-Syan Chen, and Pi-Cheng Hsiu. 2020. HAWAII open source project. https://github.com/EMCLab-Sinica/HAWAII_Project.
[45] Chih-Kai Kang, Hashan Roshantha Mendis, Chun-Han Lin, Ming-Syan Chen, and Pi-Cheng Hsiu. 2020. JAPARI open source project. https://github.com/EMCLab-Sinica/JAPARI.
[46] Raghuraman Krishnamoorthi. 2018. Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper. *arXiv preprint arXiv:1806.08342* (2018).
[47] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.
[48] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. 2019. SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules. *IEEE TC* 68 (2019), 1663–1677.
[49] Yongpan Liu, Zewei Li, Hehe Li, Yiqun Wang, Xueqing Li, Kaisheng Ma, Shuangchen Li, Meng-Fan Chang, Sampson John, Yuan Xie, Jiwu Shu, and Huazhong Yang. 2015. Ambient Energy Harvesting Nonvolatile Processors: From Circuit to System. In *Proc. of ACM/IEEE DAC*. 150:1–150:6.
[50] Ziwei Liu, Xiaoxiao Li, Ping Luo, Chen-Change Loy, and Xiaoou Tang. 2015. Semantic Image Segmentation via Deep Parsing Network. In *Proc. of IEEE ICCV*. 1377–1385.
[51] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the Value of Network Pruning. In *Proc. of ICLR*.
[52] Kaisheng Ma, Xueqing Li, Shuangchen Li, Yongpan Liu, John Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Nonvolatile Processor Architecture Exploration for Energy-Harvesting Applications. *IEEE Micro* 35, 5 (2015), 32–40.
[53] Kaisheng Ma, Xueqing Li, Huichu Liu, Xiao Sheng, Yiqun Wang, Karthik Swaminathan, Yongpan Liu, Yuan Xie, John Sampson, and Vijaykrishnan Narayanan. 2017. Dynamic Power and Energy Management for Energy Harvesting Nonvolatile Processor Systems. *ACM TECS* 16, 4 (2017), 1–23.
[54] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. In *Proc. of ACM OOPSLA*. 96:1–96:30.
[55] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proc. of ACM PLDI*. 1101–1116.
[56] Hashan Roshantha Mendis and Pi-Cheng Hsiu. 2019. Accumulative Display Updating for Intermittent Systems. *ACM TECS* 18, 5s, Article 72 (2019), 22 pages.
[57] Shree K. Nayar, Daniel C. Sims, and Mikhail Fridberg. 2015. Towards Self-Powered Cameras. In *Proc. of IEEE ICCP*. 1–10.
[58] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning. In *Proc. of ASPLOS*. 907–922.
[59] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *Proc. of ACM/IEEE ISCA*. 27–40.
[60] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. of ACM ASPLOS*. 159–170.
[61] Sungju Ryu, Hyungjun Kim, Wooseok Yi, and Jae-Joon Kim. 2019. BitBlade: Area and Energy-Efficient Precision-Scalable Neural Network Accelerator with Bitwise Summation. In *Proc. of ACM/IEEE DAC*. 1–6.
[62] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In *Proc. of ISCA*. 764–775.
[63] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency through Resource Partitioning. In *Proc. of ACM/IEEE ISCA*. 535–547.
[64] STMicroelectronics. 2021. STM32G4 Series Mixed-signal MCUs with DSP and FPU instructions. https://www.st.com/en/microcontrollers-microprocessors/stm32g4-series.html.
[65] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *IEEE PIEEE* 105, 12 (2017), 2295–2329.
[66] Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua R. Smith. 2017. Battery-Free Cellphone. *In Proc. of ACM IMWUT* 1, 2 (2017), 1-20.

[67] Greenwaves Technologies. 2018. GAP-8 IoT Application Processor with a Hardware Convolution Engine. https://greenwaves-technologies.com/gap8_gap9/.
[68] Karen Ullrich, Edward Meeds, and Max Welling. 2017. Soft Weight-Sharing for Neural Network Compression. *arXiv preprint arXiv:1702.04008* (2017).
[69] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *Proc. of IEEE CVPR*. 8612–8620.
[70] Pete Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv preprint arXiv:1804.03209* (2018).
[71] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *Proc. of IEEE CVPR*. 5687–5695.
[72] Wu Yawen, Wang Zhepeng, Jia Zhenge, Shi Yiyu, and Hu Jingtong. 2020. Intermittent Inference with Nonuniformly Compressed Multi-Exit Neural Network for Energy Harvesting Powered Devices. In *Proc. of ACM/IEEE DAC*. 1–6.
[73] Shouyi Yin, Peng Ouyang, Shibin Tang, Fengbin Tu, Xiudong Li, Shixuan Zheng, Tianyi Lu, Jiangyuan Gu, Lingling Liu, and Shaojun Wei. 2018. A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications. *IEEE JSSC* 53 (2018), 968–982.
[74] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proc. of ACM FPGA*. 161–170.
[75] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2017. Hello Edge: Keyword Spotting on Microcontrollers. *arXiv preprint arXiv:1711.07128* (2017).