Everything Leaves Footprints: Hardware Accelerated Intermittent Deep Inference

Chih-Kai Kang, *Graduate Student Member, IEEE*, Hashan Roshantha Mendis, *Member, IEEE*, Chun-Han Lin, *Member, IEEE*, Ming-Syan Chen, *Fellow, IEEE*, and Pi-Cheng Hsiu, *Senior Member, IEEE*

***Abstract*—Current peripheral execution approaches for intermittently powered systems require full access to the internal hardware state for checkpointing or rely on application-level energy estimation for task partitioning to make correct forward progress. Both requirements present significant practical challenges for energy-harvesting, intelligent edge Internet-of-Things devices, which perform hardware-accelerated deep neural network (DNN) inference. Sophisticated compute peripherals may have an inaccessible internal state, and the complexity of DNN models makes it difficult for programmers to partition the application into suitably sized tasks that fit within an estimated energy budget. This article presents the concept of *inference footprinting* for intermittent DNN inference, where accelerator *progress* is accumulatively preserved across power cycles. Our middleware stack, HAWAII, tracks and restores inference footprints efficiently and transparently to make inference forward progress, without requiring access to the accelerator internal state and application-level energy estimation. Evaluations were carried out on a Texas Instruments device, under varied energy budgets and network workloads. Compared to a variety of task-based intermittent approaches, HAWAII improves the inference throughput by 5.7%–95.7%, particularly achieving higher performance on heavily accelerated DNNs.**

***Index Terms*—Deep neural networks (DNNs), edge computing, energy harvesting, intermittent systems.**

Manuscript received March 26, 2020; revised June 9, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported in part by the Project for Excellent Junior Research Investigators, Ministry of Science and Technology, Taiwan, under Grant MOST 107-2628-E-001-001-MY3. This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Pi-Cheng Hsiu.)*

Chih-Kai Kang and Ming-Syan Chen are with the Graduate Institute of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan, and also with the Research Center for Information Technology Innovation, Academia Sinica, Taipei 11529, Taiwan (e-mail: ckkang@arbor.ee.ntu.edu.tw; mschen@ntu.edu.tw).

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

Chun-Han Lin is with the Department of Computer Science and Information Engineering, National Taiwan Normal University, Taipei 11677, Taiwan (e-mail: chlin@ntnu.edu.tw).

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

Digital Object Identifier 10.1109/TCAD.2020.3012217

I. Introduction

ADVANCEMENTS in deep neural networks (DNNs) are pushing inference to the edge of Internet of Things (IoT), to allow for responsive applications with lower bandwidth and storage costs compared to typical cloud-based solutions. IoT edge devices are now adopting sustainable battery-less, energy harvesting solutions which offer benefits, such as reduced device size and maintenance cost. However, applications running on these platforms suffer from frequent power failures and thus are executed *intermittently* [35]. Therefore, *intermittent DNN inference* has emerged as a crucial challenge as modern edge devices become self-powered and rely on local intelligence for efficient decision making. Current intermittent solutions are particularly unsuitable for hardware-accelerated DNN inference as they have issues related to performance, hardware supportability, and energy estimation, while also increasing the burden on programmers.

Despite being computationally expensive, local DNN inference on lightweight devices becomes a possibility with *hardware acceleration* [6], [21], [22]. However, even the most power-optimized DNN hardware accelerators can still consume tens to hundreds of mW of power [18], [23], which is several orders of magnitude higher than that typical ambient energy harvesters can provide [25], [34]. Therefore, a standard DNN model running on an energy-harvesting intermittent system would need to be executed across several power cycles to complete a single inference operation.

Many attempts have been made to enable correct *forward progress* of intermittently executed programs by *checkpointing* volatile system state into nonvolatile memory (NVM) [26], [29], [42], [45]. However, checkpointing incurs high runtime overhead to guarantee correct execution [26], [30], impeding concurrent task execution [15]–[17], and is prone to data inconsistency issues [39], [46], [47]. Alternatively, *task-based* programming models decompose an application into multiple tasks while ensuring each task's *estimated* energy cost can be satisfied by an energy budget [19], [36], [38]. Despite their benefits, both approaches can be expensive in terms of performance and can increase the burden on programmers, especially when implementing DNN algorithms due to their complexity and the volume of computations performed.

While checkpointing and task-based methods address intermittent *program execution*, they do not directly support *intermittent peripheral operation*. In fact, only a few attempts have been made to study the intermittent operation of complex computational peripherals such as hardware accelerators, despite

their importance for applications such as DNN inference. Peripheral operations are generally considered to be atomic (i.e., power cannot be interrupted) [27], [38], [41]. Recent efforts dynamically scale the size of an atomic peripheral operation to fit an estimated energy budget [22], [40]. However, the available energy at runtime is extremely hard to predict accurately and is susceptible to environmental factors. Consequently, inaccurate estimations can lead to energy wastage or impede program progress. Although it is possible to preserve the state of simple peripherals (e.g., built-in sensors and analog-to-digital converters) by backing up the contents of memory-mapped I/O registers [13], [31], [33], it is not the case for complex hardware accelerators with an inaccessible internal state (e.g., TI Low-Energy Accelerator (LEA) [8] and NVIDIA Deep Learning Accelerator [6]).

In this work, we introduce HAWAII, a low-overhead middleware stack for hardware accelerated intermittent DNN inference. HAWAII preserves accelerator progress across power cycles without requiring access to the peripheral internal state and application-level energy estimation. Instead, HAWAII exploits the typical operational behavior of DNN models running on accelerators. We observed that DNN inference is sequential and accelerators generally place their *suboperation* results in the user accessible memory during the execution of an accelerator operation.[1] These suboperation outputs are tracked by HAWAII as *inference footprints* to implicitly accumulate the progress of accelerated DNN inference across power cycles. Our footprint-based approach differs from checkpoint-based and task-based approaches, in that it neither *backups/restores* application context nor *partitions* the application into tasks.

To realize this concept, the first challenge is to capture and preserve the inference footprints with low runtime overhead. This is particularly difficult as the accelerator progress preserved would be simply offset by the runtime overhead. Second, upon power resumption, the captured footprints need to be used to correctly restore the execution context of the hardware accelerator. To address the first challenge, we propose *footprint preservation*, which transparently monitors inference footprints and accelerator outputs, and redirects them to persistent memory *in parallel* to accelerator execution. The second challenge is addressed by *footprint-aware recovery*, which efficiently retrieves the necessary information from the captured footprints and passes them to the hardware accelerator for correct reconfiguration, such that the interrupted inference operation can be instantly resumed rather than re-executed. In addition to accelerator-based inference tasks, our middleware design for footprint preservation and footprint-aware recovery also provides state persistence support for *CPU-based* inference tasks.

We implemented our HAWAII design on the Texas Instruments MSP430FR5994 LaunchPad [9], and used the internal LEA for DNN inference acceleration. We evaluated HAWAII on three DNNs with varied energy budgets.

[1]By an operation, we indicate one specific command supported by the accelerator, and by a suboperation, we indicate the minimum intermediate results written back by the accelerator to a memory region shared with the CPU.

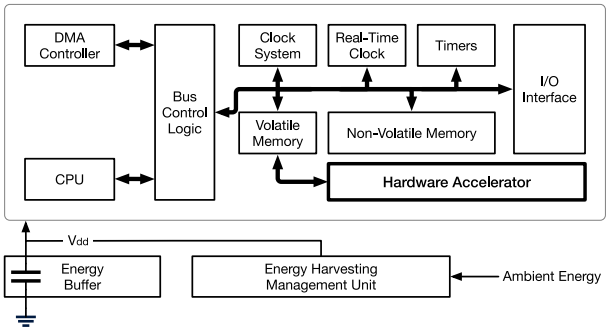

Fig. 1. System architecture of an energy-harvesting intermittent system.

Compared to a conventional task-based approach similar to Alpaca [38] and a software-based intermittent DNN inference approach similar to SONIC [22], HAWAII shows average inference throughput improvements of 50.4% and 84.9%, respectively. Furthermore, HAWAII achieves 27.3% higher inference throughput on average, compared to an approach similar to TAILS [22], an extension of SONIC to support hardware acceleration.

The remainder of this article is organized as follows. Section II provides background information and Section III explains the motivation for this work. Section IV presents our HAWAII design for hardware accelerated intermittent DNN inference, with implementation details given in Section V. The experimental results are reported in Section VI. Section VII presents some concluding remarks.

II. Background

In this section, we first present the general system architecture and current execution solutions of intermittent systems. We then introduce the basic structure of a DNN inference model and outline the operational characteristics of hardware acceleration.

A. Energy-Harvesting Intermittent Systems

Fig. 1 shows the architecture of a typical energy-harvesting intermittent system [34], [35]. Besides essential processing components, such as the CPU and main memory, modern MCUs have increasingly started to use hybrid memory to take advantage of the characteristics of volatile and NVM. Volatile memory (VM) has higher data access performance and lower energy consumption than NVM, but NVM has a higher capacity and can retain data during power failures. Internal peripherals support efficient computation. For example, the DMA controller can be used for hybrid memory data transfer without occupying the CPU, and hardware timers can be used to generate periodic events. Expensive computations such as DNN inference can be offloaded to a hardware accelerator at runtime to reduce energy consumption and latency.

As ambient energy is typically too weak to directly operate a device, the energy harvesting management (EHM) unit slowly accumulates energy into an energy buffer (e.g., a capacitor) [26], [34], [37]. The device is powered ON when the buffered energy level reaches a preset threshold (V_{on}) and powered OFF when the energy buffer is drained by the device to

a preset voltage (V_{off}), and the power cycle repeats accordingly [20]. The operation time of the system depends on the energy buffer size, the current drawn by the device, the power ON/OFF voltage thresholds, the incoming power of the EHM unit, and the inefficiencies of any power conditioning circuits. Power loss clears all the volatile register contents, memory stack, and peripheral state. Therefore, intermittent operation can corrupt processing and impede program forward progress. The program may then face nontermination, where the system would repeatedly attempt to re-execute the program from scratch.

Checkpointing is a conventional solution to address nontermination, where the system is temporarily suspended to backup its state to NVM. Checkpoints can be inserted statically into code [42] or invoked on-demand when power loss is imminent [12]. Upon power resumption, the execution context is restored from NVM and the program *resumes* from the latest checkpoint. However, *data inconsistency* may occur if NVM data are modified between the latest checkpoint and a power failure [35], [36]. Full-system checkpointing (which backups an entire system snapshot) can protect program *idempotence* (i.e., repeated execution without producing a new result) at the cost of significant runtime overhead, related to the checkpoint size [12], [26], [35]. *Task-based* programming models [16], [19], [38] alleviate the runtime overhead but require substantial programmer effort to partition an application into tasks according to the energy buffer capacity and to specify intermittent-related intertask control flow. Tasks need to be executed atomically (i.e., power uninterrupted), and application progress is saved at the end of a task. Upon recovery, to satisfy program idempotence, the system rollbacks to the beginning of the task and all partially computed results are undone, as if the task has never been executed. A task will fail if it requires more energy than available in the energy buffer, and it will be re-executed until successful. Therefore, task-based approaches fundamentally rely heavily on *accurate energy estimation*. Misestimation can still lead to the application nontermination problem, which is an impediment to their adoption.

B. Hardware Accelerated DNN Inference

A DNN model is composed of multiple computational layers and, during inference, each layer is processed sequentially. In each layer, the output of the previous layer is taken as *input* and combined with *weights* and *biases* (obtained via an offline learning phase), to generate the *output* of the layer [43]. The type of each layer depends on the functional requirement of the model. Vision-based applications generally use *convolutional* layers with heavy matrix and sliding dot-product computations to extract image features. *Fully connected* layers require more memory and are typically used after convolutional layers for classification. General-purpose layers, such as *pooling* and *nonlinearity*, are used for dimensionality reduction.

To improve inference performance, expensive DNN computations are offloaded to *hardware accelerators* such as digital signal processors or custom AI accelerators. For example, the TI LEA [8], NVIDIA Deep Learning Accelerator [6], and Greenwaves GAP8 convolution engine [21] are some recent on-chip peripherals that can accelerate certain DNN computations, such as convolution and matrix multiplication/addition. Pooling or nonlinear computations can be executed on the CPU if the accelerator lacks sufficient math support [43]. Therefore, a DNN layer can be classified as a *CPU-based* or an *accelerator-based* inference task.

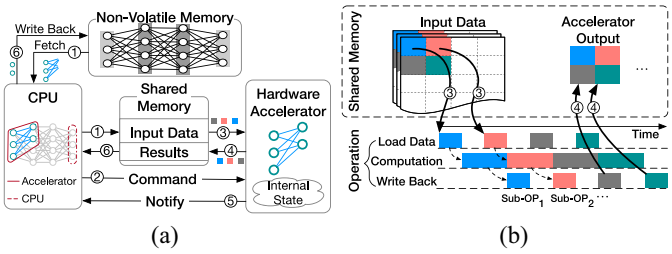

Fig. 2. DNN hardware accelerator. (a) High-level architecture. (b) Data flow inside the accelerator.

Fig. 2 shows the typical system structure and data flow of a hardware-accelerated DNN inference process. The accelerator shares a memory region with the CPU and it cannot access other unassigned memory areas. The DNN model's definition and parameters are held in the NVM. As shown in Fig. 2(a), the CPU first copies the data into the shared memory before interacting with the accelerator. Here, *data* refers to the input, model weights, and parameters. After the memory copy, the CPU initializes the accelerator with specific parameters and invokes the required operation. The CPU communicates with the accelerator via a driver API as the accelerator's internal state is not directly accessible. Once the operation is completed, the accelerator will notify the CPU to retrieve the *results* (i.e., accelerator output) from the shared memory. Typically, the internal memory of a lightweight accelerator is insufficient to store an entire DNN model or layer. Therefore, an accelerator *operation* is separated into smaller, multiple *suboperations*, as shown in Fig. 2(b). The accelerator sequentially loads each data block corresponding to a suboperation into its internal memory and performs the respective computation. Upon completion, the suboperation result is written from the accelerator back to the shared memory.

III. Challenges of Intermittent DNN Inference

Two primary concerns arise when DNN inference is naively combined with existing intermittent execution solutions, which can lead to incorrect execution behavior. First, checkpointing cannot correctly preserve the progress of peripherals such as accelerators with an inaccessible internal state. Second, task-based programming rely heavily on accurate energy estimation, which complicates DNN inference development.

A. Inaccessible Accelerator Internal State

Checkpointing can only save the state of peripherals with full access [13], [40]. As described in Section II-B, the internal state of an accelerator is typically inaccessible and thus leads to the loss of inference progress and incorrect intermittent execution behavior. Fig. 3 illustrates

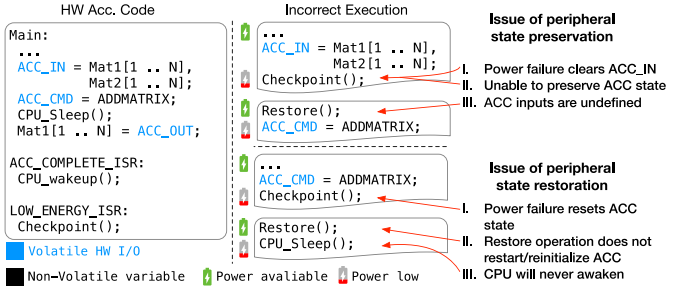

Fig. 3. Incorrect operation due to inaccessible accelerator internal state.

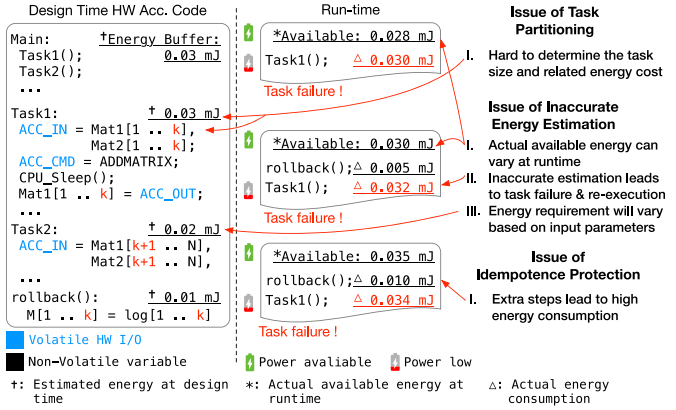

Fig. 4. Task failure due to misestimating runtime energy.

an example. In the provided accelerator code, two matrices are defined as input (ACC_IN). Next, the accelerator command (ACC_CMD) is invoked and the CPU is put to sleep. Upon completion, the ACC_COMPLETE_ISR interrupt service routine (ISR) wakes up the CPU to retrieve the output (ACC_OUT). The system checkpoints when low energy is reached (via LOW_ENERGY_ISR), but such on-demand checkpointing [12], [26] is unable to save the accelerator internal state. When the power resumes, the system continues from the interrupted point but as the accelerator's state was lost, its input (ACC_IN) remains undefined when the CPU invokes the accelerator command.

As mentioned in Section II-B, hardware accelerators require correct initialization before use. Simply checkpointing the accelerator configuration registers and restoring them on reboot as done in prior work [26] is not possible. Additionally, data input must be translated into an accelerator recognizable format. Accelerator reinitialization is unlike restoring a program or a simple peripheral with memory-mapped registers. As shown in the lower right of Fig. 3, when a power failure occurs in the middle of an accelerator operation, upon power resumption, the conventional checkpoint-based *restore* mechanism cannot correctly reinitialize and restart the accelerator. Therefore, the system may not receive a completion signal from the accelerator and thus never wake up the CPU to make progress.

B. Misestimating Task Energy Consumption

As discussed in Section II-A, task-based intermittent programming models rely heavily on accurate energy estimation for application partitioning. However, as described in Section II-B, DNNs have multiple computation layers of varying complexity, making them difficult to be partitioned appropriately. Furthermore, estimating a task's energy cost and the available energy budget is not straightforward. Supply voltage variations can affect the CPU speed and vary a task's runtime energy consumption [10] and factors, such as capacitor internal resistance, current leakage, manufacturing variability, and any incoming power (albeit weak) from the EHM unit can vary the available energy budget. Energy misestimations can cause expensive task re-execution which impacts inference response time. In the example provided in Fig. 4, the single matrix addition operation (previously shown in Fig. 3) has been partitioned into two tasks. The programmer is now burdened with accurate energy profiling to find a suitable matrix partition size k. As shown in Fig. 4 (right top and center), incorrectly estimating the energy available or task energy consumption at design time can lead to task failure and re-execution.

Modifications to the DNN model or task partitions require energy re-estimation, which impedes the DNN model parameterization and scalability (e.g., changing the DNN layer parameters or input size will change the energy consumption). Satisfying program idempotence for a DNN model is also a challenge as it contains millions of computations, and can lead to higher energy consumption than previously estimated. As shown in Fig. 4 (bottom right), ensuring idempotence after each re-execution for complex and lengthy DNN tasks can incur a significant energy cost for progress rollback to revert partial results. Recent work on intermittent DNN inference [22], [40] conceptually still rely on application-level energy estimation. They calibrate the task partition size to fit the estimated available energy and request the programmer to provide a software-only fallback routine in case of energy misestimation. Complex loop reordering and double buffering techniques are employed for tasks with long nested loops to ensure idempotence.

IV. HAWAII: Footprint-Based Intermittent DNN Inference Middleware

A. Design Rationale

In light of the aforementioned challenges, we are motivated to explore a new lightweight intermittent paradigm that enables DNN inference forward progress, without relying on access to the accelerator internal state (Section III-A) and application-level energy estimation (Section III-B). Hence, we propose the novel concept of *inference footprinting*, for preserving the state of hardware-accelerated intermittent DNN inference. The concept is inspired by the following observations. The inference process is unidirectional and each DNN layer consists of neurons which are a set of sequentially processed suboperations. Therefore, the *number of completed suboperations* indirectly represents the inference progress and is referred to as the *inference footprint* in this work. As outlined in Section II-B, the accelerator outputs each intermediate suboperation result into

its shared memory space during the execution of an operation. For example, in a matrix multiplication operation, the intermediate dot-products (i.e., suboperation results) are output to the shared memory as the accelerator multiplies each column of one matrix with the row of the other. Although the internal state is inaccessible, the count of suboperation outputs in shared memory (i.e., the footprint) indirectly reflects the accelerator progress. Similarly, the number of completed DNN layers can be used to indicate the overall network-level inference progress.

To preserve inference progress across power cycles, the system only needs to save the inference footprints (i.e., two counters with respect to completed layers and suboperations) and the intermediate suboperation results in NVM. Note that, as described in Section II-B, conventionally the result of the entire operation would still need to be bulk copied to NVM after completion, hence our approach is tightly coupled with the natural operational behavior of the accelerator.

To make use of footprinting in intermittent DNN inference, we propose the HAWAII middleware stack, which builds upon our observations. In HAWAII, we guarantee atomic execution for a suboperation, not an entire accelerator operation. A suboperation typically requires only a few clock cycles to complete and will require re-execution if power fails during its execution. Therefore, we only need to ensure that the capacitor's energy budget is sufficient for the largest accelerator suboperation. With atomicity constraints significantly relaxed, HAWAII does not require task partitioning based on application-level energy estimation. Since the progress can resume from the interrupted suboperation after power resumption, HAWAII can fetch variable input volumes for one invoked accelerator operation, without breaking an accelerator operation into multiple operations with suitably sized input data. In contrast to checkpointing, we save the output of suboperations, rather than backing up application context, hence HAWAII can preserve forward progress without application suspension and access to peripheral internal state, which drastically reduces the *overhead* of state persistence.

Fig. 5 further illustrates the benefits of HAWAII compared to a task-based intermittent approach. Misestimating the energy cost/availability can cause a task-based approach to fail and repeatedly redo the operation in the next power cycle. Furthermore, the accelerator requires initialization prior to executing each subtask and after completion the corresponding result needs to be copied back to NVM. In comparison, our footprinting-based approach can *resume* an interrupted accelerator operation, by capturing and using the footprints of the accelerator, and complete the operation across multiple power cycles. Note that the cost of our recovery handling reduces in every power cycle, as only the required input data for the remaining suboperations are fetched during inference state recovery.

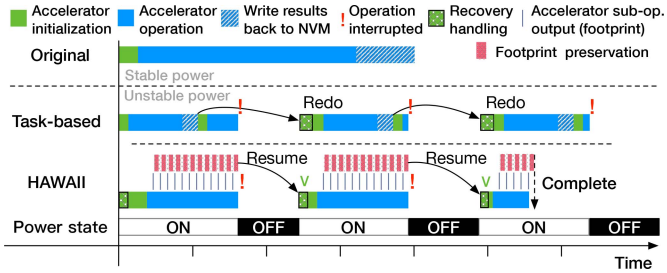

Fig. 5. Illustration of HAWAII runtime execution.

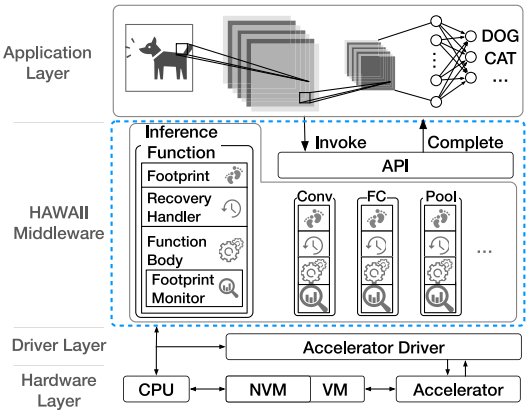

Fig. 6. System architecture with HAWAII middleware.

B. System Architecture

Generally, a lightweight system has a layered software architecture as shown in Fig. 6. The DNN model architecture is defined by the user program in the application layer. Inference inputs are obtained from sensors at runtime. For accelerator-based inference operations, the input data and parameters are first fetched from NVM to the shared VM region and then the required accelerator command is invoked via a driver layer API call, which instructs the accelerator peripheral in the hardware layer. Once the operation is completed, the accelerator places its output in shared memory and notifies the CPU. If system state persistence under intermittent power is managed using checkpointing or task-based programming (Section II-A), some critical drawbacks as described in Section III will occur. Moreover, both solutions require atomicity constraints for an entire accelerator operation and progress is lost if power fails before its completion.

Contrary to existing approaches, the proposed HAWAII design allows for the *accumulated* execution of an accelerator operation across power cycles. Realizing our concept of inference footprinting raises two key challenges. First, the inference progress must be efficiently captured and preserved by footprints with minimum runtime cost. Second, upon power resumption, the captured footprints need to be used to reinstate the interrupted inference process. Low overhead *footprint preservation* is used to address the first challenge of inference state backup, where the accelerator output is monitored *in parallel* to the accelerator execution, and the footprints and suboperation results are redirected to NVM. The second challenge of inference state restoration is addressed by *footprint-aware recovery*, where relevant parameters from the saved footprints as well as correct data inputs are fetched and passed to the accelerator to reinitialize and reconfigure its execution context to resume inference. Note that both footprint

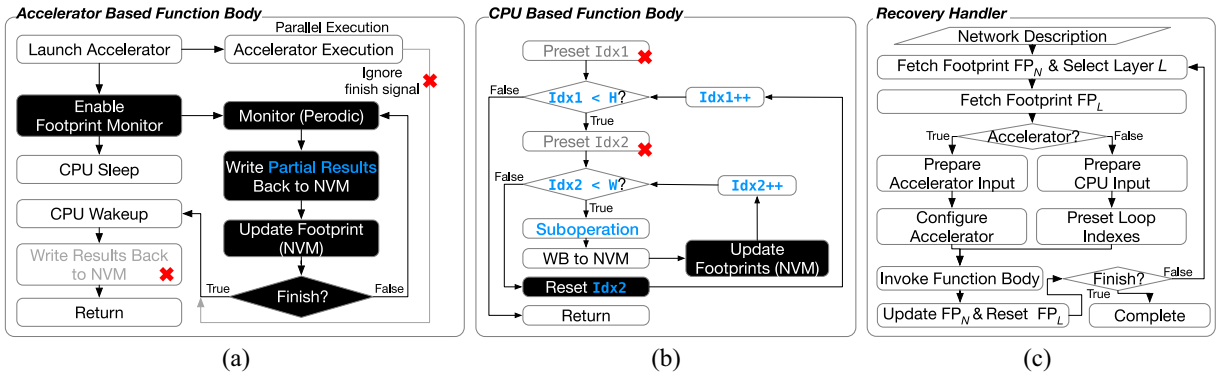

Fig. 7. HAWAII control flow of key components. (a) Accelerator-based footprint preservation. (b) CPU-based footprint preservation. (c) Footprint-aware recovery.

preservation and footprint-aware recovery are transparently executed without programmer involvement.

Our HAWAII design is implemented at the middleware layer as shown in Fig. 6. We refer to a DNN layer as an *inference task*, and each inference task is wrapped into an intermittence-aware *inference function*. HAWAII consists of multiple inference functions, which are exposed via an intuitive API. Each inference function contains four major components: 1) the *footprint variable*; 2) *function body*; 3) *footprint monitor*; and 4) *recovery handler*. The footprint variable indicates the current progress with respect to a network/layer-level suboperation. The function body contains the actual operations in the inference pipeline (e.g., convolution or pooling). The footprint monitor works closely with the function body to perform footprint preservation, where the number of completed accelerator/CPU-based suboperations is periodically captured. The recovery handler performs footprint-aware recovery to correctly resume an interrupted inference task upon power resumption. To account for flexibility, HAWAII supports both accelerator-based and CPU-based inference tasks, but the realization of footprinting for them will differ as explained in the following sections.

Once the DNN structure has been defined, the programmer invokes the *main* inference function to signal HAWAII to start the execution of the DNN model. HAWAII then executes each specified inference task, seamlessly handling any power intermittence-related control flow and recovery procedure, without programmer intervention. The structure of an inference function is illustrated in Fig. 6, and the control flow within the function body and recovery handler is shown in Fig. 7. At the start of each inference function, the recovery handler will first prepare the input data according to the inference footprint. Next, within the function body, the corresponding accelerator/CPU operation is started and the footprint monitor is immediately enabled. Finally, HAWAII will signal the user program upon end-to-end inference completion. The following sections provide details of each core component in our HAWAII design.

C. Footprint Preservation

As shown in Fig. 8, we capture footprints at two levels of the inference workflow: 1) the *number of completed suboperations* within the current layer (denoted FP_L) and 2) the *number of processed layers* at the network level (denoted FP_N). The FP_L and FP_N counters are, respectively, incremented after a suboperation and a network layer are completed. Both these values combined can indicate the overall inference progress sufficiently. The mechanism of the *footprint monitor* for accelerator/CPU-based inference tasks differs as follows.

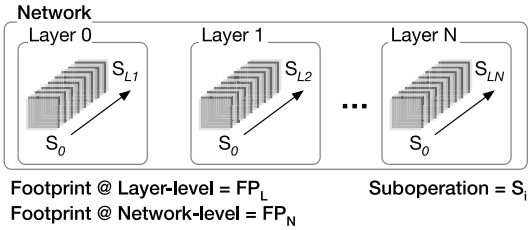

Fig. 8. Footprinting methodology.

1) Accelerator-Based Inference Tasks: As explained in Section IV-A, to maintain accelerator state persistence, we track the count of completed suboperations and the suboperation results in NVM. Fig. 7(a) shows the control flow of the accelerator-based inference function body. In the HAWAII API, each accelerator-based inference function executes different accelerator commands, based on the layer requirement [e.g., fully connected (FC) or convolution]. The footprint monitor is enabled immediately after any accelerator command is invoked. It is then periodically triggered and executed in parallel to the accelerator, to update the suboperation footprint (FP_L) in NVM and transfers the suboperation results to NVM. Thus, by tracking FP_L and saving the intermediate results, we can accumulate progress, and therefore relax the atomicity constraints of the accelerator operation. As each suboperation has a fixed execution cost, the monitoring periodicity can be determined at design time to avoid obsolete monitor invocations. Alternatively, the periodicity can easily be extended to capture aggregate results of B suboperations. By default, $B = 1$, to maximize power failure resiliency, but can also be adjusted dynamically if necessary (e.g., according to the stability of the power source).

Once the footprint monitor is enabled and triggered periodically by a hardware *timer*, the CPU is put to sleep and will be woken up when the accelerator operation is completed. We do not need to bulk transfer the result of an entire operation

to NVM at the end of its execution [hence grayed out in Fig. 7(a)], as the footprint monitor already periodically writes the intermediate results to NVM. Similarly, the completion signal from the accelerator is ignored, as its completion status can be inferred by the footprint monitor.

2) CPU-Based Inference Tasks: Similar to an accelerator-based inference task, the corresponding inputs and parameters of a CPU-based inference task are fetched from NVM before the computation is started. However, unlike the former, the latter is carried out entirely in software. As shown in Fig. 7(b), the main body of a CPU-based inference function, which comprises a set of nested loops, takes in the inputs of the inference task and sequentially performs suboperations on them to generate the output. Nested loops are a common processing structure in a neural network implementation [43]. Thus, each loop is associated with an index counter and all the index counters combined can indicate the current progress (with respect to a suboperation).

For example, in a max-pooling task, the input values are grouped and the maximum value of each group is obtained iteratively. From an implementation perspective, the inner loop would iteratively calculate the maximum within a group and the outer loop would iteratively apply the logic across all available input groups. In the example provided in Fig. 7(b), the FP_L footprint in NVM is updated within the inner loop and reflects the completion of a suboperation. The `idx1` and `idx2` index counters, which reside in VM, are combined to manage the current suboperation and can be implicitly inferred from the FP_L footprint, allowing HAWAII to resume the nested loops at an arbitrary iteration during recovery after power resumption.

D. Footprint-Aware Recovery

The preserved footprints are used to determine the information required to correctly reconfigure and continue the interrupted inference task. This footprint-aware recovery is executed upon power resumption, by the *recovery handler* inside each inference function. As shown in Fig. 7(c), the recovery handler first uses the FP_N footprint and the DNN description to determine the layer from which to continue inference. Next, using FP_L, it determines the information required to reinitialize the accelerator/CPU-based inference function. Specifically, the recovery handler determines the correct parameters to resume from the interrupted suboperation and the data (i.e., layer input data, weights, and biases) that should be refetched from NVM for the remaining suboperations. FP_N is updated after an inference task is completed, and FP_L is reset. The recovery steps for accelerator-based and CPU-based inference tasks are detailed as follows.

1) Accelerator-Based Inference Tasks: As described in Section IV-C1, an accelerator-based inference task consists of a sequence of suboperations. The recovery handler first fetches FP_L from NVM. FP_L is used to determine and prepare the required accelerator input data, configure the accelerator with specific parameters, and determine the output memory location offset. Then, for the accelerator-based inference task, the recovery handler correctly reconfigures the accelerator peripheral to resume inference from the end of the last completed suboperation before a power failure. The respective steps are shown in Fig. 7(c) (left branch).

Only the required *subset* of input data corresponding to the *remaining* suboperations are copied to the shared VM region, which reduces the latency and energy overhead. Similarly, the command parameters are set to specify the *length* of the subset of inputs (i.e., vector or matrix size), which essentially specifies the number of remaining accelerator suboperations to be executed. The output memory location offset ensures that the footprint monitor transfers the intermediate results of the remaining suboperations to the correct NVM location. Once the accelerator is appropriately configured, the accelerator-based inference task's function body (as described in Section IV-C1) is invoked.

2) CPU-Based Inference Tasks: The CPU-based inference function body (Section IV-C2) typically consists of a nested loop software implementation, with a suboperation inside each iteration. Upon power resumption, the CPU-based recovery handler ensures that the nested loop structure correctly resumes from the correct iteration with the required data for the suboperation. As shown in Fig. 7(c) (right branch), the CPU-based recovery handler is more straightforward than the accelerator-based counterpart but follows a similar structure, i.e., FP_L is used to fetch the required subset of data input and determine the loop indices.

In the example provided in Fig. 7(b), the start indices of the loops are calculated as $\texttt{idx1} = \lfloor (FP_L/W) \rfloor$ and $\texttt{idx2} = (FP_L \bmod W)$, where W denotes the inference task's input width. Subsequently, the function body (described in Section IV-C2) is invoked to complete the remaining suboperations in loop iterations. Unlike existing CPU-based intermittent inference approaches [22], our loop control variables and related suboperation progress reside in VM and will be lost altogether when a power failure occurs before a suboperation is complete, as if the incomplete suboperation has never been executed; therefore, partial suboperation progress needs not to be reverted to protect idempotence.

E. Generality

HAWAII can be integrated with different types of accelerators, but we specifically target lightweight accelerators that do not have large internal memory (to accommodate all input/output data of an operation). Therefore, suboperations are fine-grain and intermediate results have to be frequently written back to shared memory. However, accelerators may vary in suboperation granularity. Footprint preservation can capture the output of coarse-grain suboperations with larger partial results (e.g., multicore parallel architectures), but this would relatively increase the amount of progress lost due to a power failure during a suboperation execution. The finer-grained the suboperation, the more progress can be preserved. Furthermore, for accelerators equipped with non-volatile internal memory, HAWAII can easily be extended to reset the accelerator state and clear the internal NVM to revert the partial progress upon each power resumption, as

if the incomplete suboperation has never been executed, to protect idempotence. However, such conditions are rare, as lightweight accelerators would typically use VM instead of NVM for performance considerations.

HAWAII currently supports the most common DNN layers (e.g., convolution, FC, pooling, etc.), but our horizontally scalable design (Fig. 6) allows us to expand its scope easily. Footprint preservation and recovery directly support the sequential processing characteristic seen in typical DNN models. Neural networks with feedback paths (e.g., RNN) can be unrolled into a sequential execution model.

Different model compression techniques can be used to reduce the memory storage, execution time, and energy consumption during inference. Broadly speaking, *general* compression techniques such as quantization or bit-width reduction [32] can be directly applied to commodity hardware platforms (e.g., MCU/DSP), while other techniques such as weight sharing or pruning [24] may require *customized* ASIC/FPGA accelerator designs to function efficiently. Depending on the model compression technique used, we can reduce the cost of computations or the cost of data accesses, or both. HAWAII is orthogonal to model compression and can be coupled with both general and ASIC/FPGA-customized compression techniques to further improve intermittent inference throughput. Generally, DNN models with a higher cost of accelerated computations will benefit more by using HAWAII, as only the *remaining* suboperations are executed after power resumption. Therefore, if the compression technique reduces the cost of data accesses, then computation becomes a bottleneck, which in turn increases the efficacy of using HAWAII.

V. Implementation

We realized our design on the MSP430FR5994 device produced by Texas Instruments (TI). It is commonly used in intermittent systems and includes the TI LEA. LEA offers vector and matrix math acceleration. The platform has 256-KB ferroelectric random-access memory (FRAM) and 8-KB static random-access memory (SRAM), respectively used as NVM and VM. LEA shares 4 KB of the SRAM with the CPU. We developed a custom LEA driver that provides efficient DMA usage and better asynchronous functionality, compared with TI's LEA driver (`DSPLib`).

A. Peripheral Usage

1) Accelerator Commands (LEA): Three LEA commands are used to implement 2-D convolution (CONV) and FC layers. The `LEACMD__FIR` command (1-D convolution/sliding dot-product) is combined with `LEACMD__ADDMATRIX` (matrix addition) to implement a CONV layer and `LEACMD__MAC` (vector multiply–accumulate/dot-product) is used for FC. The number of LEA commands per layer is dependent on the kernel size, the number of input/output channels, and the dimensions of the input. In contrast, the max-pooling layer (POOL) is purely CPU-based and neuron activations within each layer are also performed in software.

2) Efficient Data Transfer (DMA): The multichannel DMA and CPU low-power modes are utilized to reduce latency and energy consumption. In the recovery handler, a DMA channel (`DMA0`) is used to copy the LEA inputs and parameters (i.e., weights and biases) to VM. The footprint monitor is executed *in parallel* to LEA and implemented entirely in hardware by integrating the DMA and timer peripherals. A timer triggers `DMA0` to update the LEA footprints in NVM and a second timer triggers `DMA1` to write back the suboperation results to NVM. To conserve energy, the CPU is put to sleep (low-power mode) during the entire execution of LEA and footprint preservation and only woken up after the LEA command is completed.

3) Footprint Monitor Periodicity (Timer): The footprint monitor is triggered periodically by a hardware timer. The periodicity depends on the LEA suboperation latency. For example, a suboperation of the `LEACMD__FIR` command has a period of $(6 + K)$ clock cycles, where K is the convolution kernel size. Such latency information is generally included in the accelerator specification or can be calculated offline using a straightforward linear regression. We slightly overestimated the latency given in the LEA specification [2], as underestimation can lead to missing the suboperation output.

B. HAWAII API

As shown in Table I, HAWAII exposes an API for implementing intermittent DNN inference. The API consists of custom data structures (e.g., `HAW_LAYER`), accelerator/CPU-based inference functions (e.g., `IF_CONV`), and a main inference function (`HAW_INFERENCE`). The FP_N and FP_L footprints are, respectively, defined within the `HAW_NETWORK` and `HAW_LAYER` data structures. The DNN model is specified using the custom data structures, where each layer must be associated with an inference function, input/output data locations, and parameters. The model is then passed into the main inference function to start the intermittent inference process. The API also maintains idempotence in its implementation by avoiding write-after-read dependencies [40] between two footprint updates in NVM-related code. The code is simple and only involves reading/writing suboperation results from/to and updating footprints in NVM.

VI. Performance Evaluation

Our evaluation demonstrates that compared to the latest intermittent DNN inference approaches: 1) HAWAII, on average has lower overhead under continuous power and 2) HAWAII greatly improves intermittent inference throughput, particularly for DNN models with medium to heavy acceleration.

A. Experimental Setup

We conducted a series of experiments on the TI MSPEXP430FR5994 platform with the LEA peripheral, as described in Section V, with experimental settings summarized in Table II. To emulate execution patterns of intermittent systems, we used a Keithley 2280S power supply unit with an EHM unit that consists of a TI-BQ25504 boost converter,

TABLE I
SUMMARY OF HAWAII API FUNCTIONS AND DATA STRUCTURES

Syntax	Type	Description
`HAW_INFERENCE(HAW_NETWORK *net)`	Main	DNN inference invocation (input: network)
`IF_FC / IF_CONV / IF_POOL (HAW_LAYER *l)`	Inference Function	FC / CONV / Pooling layer invocation (input: layer properties)
`HAW_NETWORK`	Data structure	DNN model: {List of layers (`HAW_LAYER[]`), FP_N, num. layers}
`HAW_LAYER`	Data structure	Layer properties: {Inference func., I/O (`HAW_DATA`), FP_L, parameters (`HAW_PARA`)}
`HAW_PARA`	Data structure	Layer parameters: {Pointer to weights and biases, kernel size}
`HAW_DATA`	Data structure	Input/Output data properties: {data pointer, channel size, input dimensions}

TABLE II
SPECIFICATION OF THE EXPERIMENTAL PLATFORM

MCU	TI-EXP430FR5994 @ 16 MHz
EHM voltage thresholds	ON: 2.88 V, OFF: 2.36 V
Power source	3 mW (1 V, 3 mA)
Energy buffer (capacitance)	100 μF, 330 μF and 1 mF
Execution duration	10 minutes

TABLE III
APPLICATIONS AND RESPECTIVE DNN MODELS USED FOR PERFORMANCE EVALUATION

Model	Layers	# Operations
DNN used for Speech Keyword Spotting (SKS) [14] Dataset: Speech commands [44] Input:25×10 Accuracy: 84.6% Model size: 134 KB *(Light acceleration, I=0.98)*	FC: 250×128 FC: 128×128 FC: 128×128 FC: 128×24	`LEA_MAC`: 408 CPU (activations): 408
CNN used for Human Activity Recognition (HAR) [3] Dataset: Smartphone sensor data [11] Input:128×9 Accuracy: 93% Model size: 15 KB *(Medium acceleration, I=9.03)*	CONV:9×18×1×2 POOL:1×4 CONV:18×36×1×2 POOL:1×4 CONV:36×72×1×2 POOL:1×4 FC: 144×6	`LEA_MAC`: 6 `LEA_MATRIXADD`: 3528 `LEA_FIR`: 3402 CPU (POOL): 1008 CPU (activations): 4038
CNN used for Handwritten Digit Classification (HDC) [1] Dataset: MNIST [28] Input:28×28×1 Accuracy: 99% Model size: 121 KB *(Heavy acceleration, I=37.66)*	CONV:1×20×5×5 POOL:2×2 CONV:20×40×5×5 POOL:2×2 FC: 640×64 FC: 64×10	`LEA_MAC`: 74 `LEA_MATRIXADD`: 4172 `LEA_FIR`: 4100 CPU (POOL): 3520 CPU (activations): 14154

an energy buffer (capacitor), and a power control switch. We generated a power source of 3 mW (1 V, 3 mA), representative of the typical power output produced by a small (6 cm^2) sized solar cell under indoor light [5]. Such a power source is insufficient to completely execute a single inference, resulting in frequent power failures and resumptions. Experiments were conducted under varied energy budgets using low to high energy buffer capacities (100 μF, 330 μF, and 1 mF), as well as under stable continuous power.

Table III shows the three DNN models used in our evaluation, namely, an FC DNN model (referred to as SKS) used for audio-based speech keyword spotting [14], a CNN model (referred to as HAR) used for accelerometer data-based human activity recognition on wearables [3], and a CNN model (referred to as HDC) used for image-based handwritten digit classification [1]. Considering that energy-harvesting devices are typically intended for lightweight IoT applications, these models are representative and selected as they fit into the limited NVM available on our target platform (<256 KB). DNN models typically have a mixture of CONV, FC, and POOL layers with varying dimensions. Compared to FC layers, CONV layers have a larger number of matrix and sliding dot-product operations, and therefore can better utilize the hardware accelerator. The SKS model consists of four FC layers, HAR has three small CONV layers and one FC layer, and HDC has two large CONV layers and two FC layers. We use *acceleration intensity*, I = ([# of MAC operations]/[# of memory block accesses]), to quantify the amount of accelerator usage in a DNN model. Therefore, based on the acceleration intensity of the chosen three DNN models as shown in Table III, SKS, HAR, and HDC can be, respectively, classified as models with *light*, *medium*, and *heavy* acceleration. We could similarly categorize any standard CNN model depending on the level of accelerator usage.

All three models use dense convolution with a stride of 1 and no zero-padding. To adapt the models to our target platform while reducing their memory requirements, model inputs and weight parameters were quantized to a 16-b fixed-point representation (Q15.1 format) from the 32-b floating-point representation used during training, without significant loss of accuracy. Each application was executed for 10 min (repeated inferences), resulting in 500 to 5000 power cycles depending on workload/capacitance, sufficient to mitigate experimental variances while reproducing the results.

We compared our design against the following approaches: a software-only inference implementation (denoted SW), a LEA-based accelerated inference using TI's driver (denoted AC), a task-based intermittent approach using *loop continuation* for software-only inference (denoted LC), and a task-based intermittent approach using LEA-based acceleration (denoted TL). SW and AC do not handle intermittent power and will lose state and restart the entire application upon power resumption. LC stores loop control variables and data buffers directly in FRAM for state persistence, similar to the latest software-only intermittent inference approach, SONIC [22]. TL follows a task-based approach similar to Alpaca [38], but uses hardware acceleration. TL partitions each inference layer into equal-sized tasks, similar to the example in Section III-B, with multiple CPU/LEA-based operations per task. Progress is preserved at task boundaries, and a task failed in the middle is re-executed upon reboot. We compare against static task sizes of (TMAX/32) and (TMAX/512) (respectively, denoted as *TL-L* and *TL-S*), corresponding to large and small task sizes. Here, *TMAX* refers to the maximum number of operations per layer. Finally, we compare against an adaptive TL variant (denoted *TL-A*), which is similar to TAILS [22], an extension of SONIC to add support for hardware accelerators. TL-A performs a one-time calibration at the start to find the suitable task size for the available energy.

We evaluate our design based on two key metrics: 1) runtime overhead and 2) inference throughput. We measure runtime overhead in terms of *execution time* and *energy*

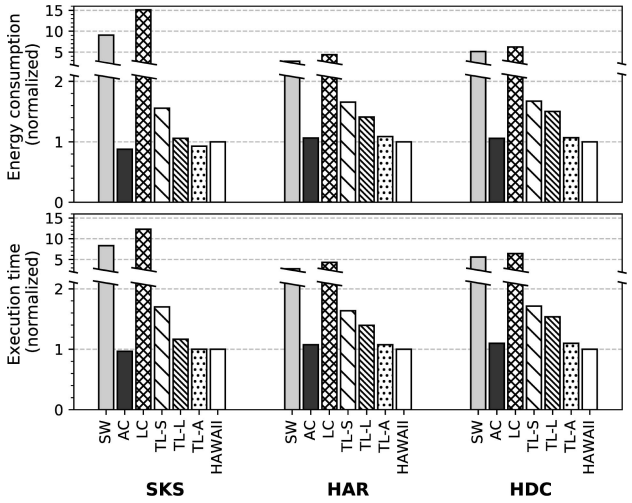

Fig. 9. Execution time and energy consumption for one inference under continuous power.

consumption required to complete one end-to-end inference under stable continuous power. A digital oscilloscope was used to measure execution time and the TI EnergyTrace software tool [7] was used to measure energy consumption. This experiment set evaluates the extra runtime overhead required by HAWAII to enable inference footprinting when power is *stable*. However, HAWAII is primarily developed to improve hardware-accelerated DNN inference under *intermittent power*. In the other experiment set, we measured inference *throughput* under intermittent power, with respect to the number of successfully completed end-to-end inferences within a 10-min experimental run, for all approaches and experimental configurations.

B. Execution Time and Energy Overhead

Fig. 9 shows the execution time and energy consumption overhead of all approaches executing the three DNNs, to complete a single end-to-end inference under continuous power. Results are normalized with respect to HAWAII, which takes 30, 611, and 1430 ms execution time and consumes 0.19, 4.27, and 10.63 mJ of energy to complete one inference under the SKS, HAR, and HDC applications. Energy consumption is directly related to the inference execution time; therefore, they follow a similar trend. The TL variants bulk transfer the accelerator output to the NVM *after* the *atomic* accelerator operation is completed. In contrast, HAWAII efficiently utilizes the DMA engine in parallel to the hardware accelerator execution to save suboperation progress and outputs (i.e., footprint preservation). Moreover, unlike the vendor-provided accelerator library, HAWAII's custom accelerator driver puts the CPU to sleep whenever the DMA or LEA peripheral is in use. These factors enable HAWAII to achieve lower overhead and energy consumption for models with medium to heavy acceleration (i.e., HAR and HDC).

Comparing AC to SW, we can see that hardware acceleration provides 2–8 times execution time speed-up and 3–10 times energy savings, depending on the application. LC shows the highest overhead, due to the lack of LEA and DMA usage, and because of additional FRAM accesses. TL's task management overhead increases as the task size decreases (i.e., a larger number of tasks), which is specifically related to the number of times LEA and DMA operations have to be invoked/initialized. Hence, TL-S has higher time and energy overhead than TL-L. TL-A follows the same task size calibration routine as TAILS [22], where it adaptively selects a suitable task size for a given energy budget, by incrementally reducing the task size until it finds a size that fits the given energy budget. Under stable power (or for a large energy budget), TL-A's calibration defaults to the largest task size; hence, TL-A has a relatively lower overhead than TL-S and TL-L.

HAWAII is primarily developed for *hardware accelerated* DNN inference. Models with light acceleration (e.g., SKS) have primarily multiply–accumulate computations (i.e., operations have only a single suboperation). Consequently, footprint preservation incurs a relatively higher overhead for SKS, compared to the heavily accelerated HAR and HDC models which require more matrix computations. Therefore, under *stable* power, HAWAII shows comparable execution time and 7% higher energy costs than TL-A for SKS, but shows 7.5%–9.8% (average 8.7%) lower execution time and 6.8%–8.8% (average 7.8%) lower energy consumption than TL-A for HAR and HDC. As TL-A defaults to the largest task size under stable power, its associated runtime overhead is small, but even in such a worst-case scenario, we demonstrate that HAWAII's overhead is comparable to TL-A. Thus, even when better and more sophisticated task size calibration techniques arise, we expect HAWAII's overhead under stable power will still be comparable. Compared to TL-L and TL-S, HAWAII, respectively, shows 20%–70% and 6%–67% time and energy overhead reductions across all DNNs. On average, HAWAII has respectively, 7.7 and 8.6 times lower time and energy overhead compared to LC, mainly due to hardware acceleration.

C. Inference Throughput

Fig. 10 shows the inference throughput under intermittent power with varied capacitance levels for all evaluated approaches and DNN applications. Higher capacitance (i.e., larger energy budget) allows for longer execution without power interruption and thus higher throughput. Larger and more complex DNN models take more time and energy to execute, resulting in lower throughput. The SKS model Fig. 10(a) is small enough to complete several inferences in one power cycle, but HAR Fig. 10(b) and HDC Fig. 10(c), take several power cycles to complete one inference. SW and AC have no intermittent support and thus show zero throughput when the available energy is insufficient to complete a single end-to-end inference in one power cycle.

As discussed in Section VI-B, TL with a large task size has lower overhead than with small task sizes, which leads TL-L to have higher throughput than TL-S. However, larger task sizes may fail to fully complete a task when the available energy is very low, resulting in repeated *re-execution* and zero throughput (e.g., TL-L on HDC with 100-μF capacitance). TL-A adaptively halves its task size until it fits the given

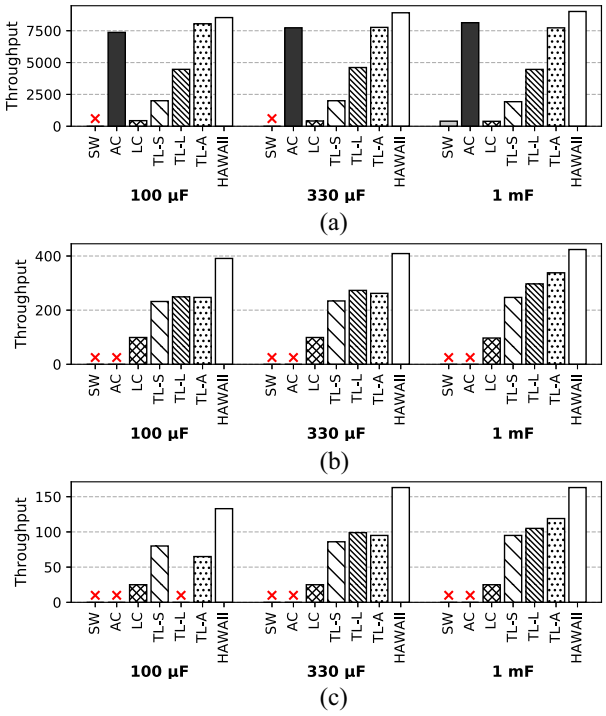

Fig. 10. Inference throughput under intermittent power. (a) SKS (small network). (b) HAR (medium network). (c) HDC (large network).

energy budget but such a calibration process may consume several power cycles to find a suitable task size for a given energy budget [22]. Moreover, TL-A may settle on a relatively larger task size, resulting in a throughput drop (e.g., TL-A has lower throughput than TL-L in HAR and HDC at 330 μF), as the time and energy wastage of re-execution increases with the size of the failed task. Finding an optimum task size for an arbitrary energy budget remains a challenge with task-based execution. LC makes forward progress in all conditions, but shows poor throughput results due to its software-only implementation.

Overall, HAWAII outperforms all evaluated baselines in all experimental configurations under intermittent power. The relative throughput improvement of HAWAII over the TL variants is generally higher in DNN models with medium to heavy acceleration (HAR and HDC). In addition, unlike the task-based approaches, HAWAII needs to neither re-execute an entire inference task nor an accelerator operation. It can accumulate the accelerator progress and resume the interrupted operation in the next power cycle. HAWAII can achieve a 5.7%–51.1% improvement in intermittent inference throughput over TL-A, with an average improvement of 27.3% across all experimental configurations. Compared to the fixed task size (TL-L/S) and software-only (LC) approaches, HAWAII achieves on average 50.4% and 84.9% improvements on throughput, respectively. It is worth noting that for SKS, HAWAII still shows an improvement of 5.7%–14.3%, with an average 11% throughput improvement over TL-A, even though HAWAII is less well suited for DNN models with light acceleration (SKS) compared to models with medium to heavy acceleration (HAR and HDC).

VII. Conclusion

This article presented the concept of inference footprinting for hardware-accelerated DNN inference on intermittent systems. Unlike existing approaches, our approach requires neither access to the accelerator internal state nor application-level energy estimation. To realize this concept, we use footprint preservation and footprint-aware recovery. Footprint preservation transparently tracks the number of completed suboperations and corresponding outputs of an accelerator/CPU-based inference task during execution. Footprint-aware recovery efficiently uses the preserved footprints to reconfigure and resume an inference task upon power resumption. We integrate these principles in HAWAII, a middleware software stack, which allows hardware-accelerated intermittent DNN inference development with minimum programmer effort.

We evaluated HAWAII on a TI MSPEXP430FR5994 platform with a LEA, across different DNN models and energy buffer sizes. Under intermittent power, HAWAII achieves 5.7%–51.1% higher inference throughput compared to a task-based intermittent DNN inference approach with hardware acceleration (TL-A). Compared to a conventional task-based approach (TL) and a software-only intermittent inference approach (LC), HAWAII, respectively, achieves 29.9%–78.6% and 74.7%–95.7% higher throughput. Furthermore, under continuous power, HAWAII, respectively, shows on average 4.3% and 50.1% reductions in time and energy overhead compared to TL-A and TL, as well as eight times lower overhead compared to LC. Our evaluation showed HAWAII is highly suitable for heavily accelerated DNN models that require high throughput even under small energy budgets.

The HAWAII runtime and API source code has been made open [4], which helps developers build low cost, intermittent-aware inference systems. Future work will seek to extend and integrate HAWAII for other types of accelerators and DNNs.

References

[1] *Basic MNIST Example Using PyTorch*. Accessed: Mar. 26, 2020. [Online]. Available: https://github.com/pytorch/examples/tree/master/mnist

[2] *Benchmarking the Signal Processing Capabilities of the LEA on MSP430 MCUs*. Accessed: Mar. 26, 2020. [Online]. Available: http://www.tij.co.jp/jp/lit/an/slaa698b/slaa698b.pdf

[3] *CNN Model for Human Activity Recognition*. Accessed: Mar. 26, 2020. [Online]. Available: https://github.com/healthDataScience/deep-learning-HAR.

[4] *HAWAII Open Source Project*. Accessed: Mar. 26, 2020. [Online]. Available: https://github.com/EMCLab-Sinica/HAWAII_Project

[5] *IXYS–IXOLAR High Efficiency Solar Cell*. Accessed: Mar. 26, 2020. [Online]. Available: http://ixapps.ixys.com/DataSheet/SM111K04L.pdf

[6] *NVIDIA Deep Learning Accelerator*. Accessed: Mar. 26, 2020. [Online]. Available: http://nvdla.org/

[7] *Texas Instruments—EnergyTrace Technology for MSP430*. Accessed: Mar. 26, 2020. [Online]. Available: http://www.ti.com/tool/ENERGYTRACE

[8] *TI Low-Energy Accelerator*. Accessed: Mar. 26, 2020. [Online]. Available: http://www.ti.com/lit/an/slaa720/slaa720.pdf

[9] *TI MSP430FR5994*. Accessed: Mar. 26, 2020. [Online]. Available: http://www.ti.com/product/MSP430FR5994

[10] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "The betrayal of constant power X time: Finding the missing joules of transiently-powered computers," in *Proc. ACM LCTES*, 2019, pp. 97–109.

[11] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones," in *Proc. ESANN*, 2013, pp. 437–442.
[12] D. Balsamo *et al.*, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 12, pp. 1968–1980, Dec. 2016.
[13] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "SyTare: A lightweight kernel for NVRAM-based transiently-powered systems," *IEEE Trans. Comput.*, vol. 68, no. 9, pp. 1–14, Sep. 2018.
[14] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *Proc. IEEE ICASSP*, 2014, pp. 4087–4091.
[15] W.-M. Chen, Y.-T. Chen, P.-C. Hsiu, and T.-W. Kuo, "Multiversion concurrency control on intermittent systems," in *Proc. IEEE/ACM ICCAD*, 2019, pp. 4–7.
[16] W.-M. Chen, P.-C. Hsiu, and T.-W. Kuo, "Enabling failure-resilient intermittently-powered systems without runtime checkpointing," in *Proc. ACM/IEEE DAC*, 2019, pp. 1–6.
[17] W.-M. Chen, T.-W. Kuo, and P.-C. Hsiu, "Enabling failure-resilient intermittent systems without runtime checkpointing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access.
[18] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[19] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," in *Proc. ACM OOPSLA*, 2016, pp. 514–530.
[20] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *Proc. ACM ASPLOS*, vol. 53, 2018, pp. 767–781.
[21] E. Flamand *et al.*, "GAP-8: A RISC-V SoC for AI at the edge of the IoT," in *Proc. IEEE ASAP*, 2018, pp. 1–4.
[22] G. Gobieski, N. Beckmann, and B. Lucia, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *Proc. ACM ASPLOS*, 2019, pp. 199–213.
[23] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE ISCA*, 2016, pp. 243–254.
[24] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Prof. ICLR*, 2016, pp. 1–11.
[25] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan, "Powering the Internet of Things," in *Proc. ACM/IEEE ISLPED*, 2014, pp. 375–380.
[26] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan, "Energy-aware memory mapping for hybrid FRAM-SRAM MCUs in intermittently-powered IoT devices," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 3, pp. 1–23, 2017.
[27] C.-K. Kang, C.-H. Lin, P.-C. Hsiu, and M.-S. Chen, "HomeRun: HW/SW co-design for program atomicity on self-powered intermittent systems," in *Proc. ACM/IEEE ISLPED*, 2018, pp. 1–6.
[28] Y. LeCun, C. Cortes, and C. J. Burges. *The MNIST Database of Handwritten Digits*. Accessed: Mar. 26, 2020. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[29] J. Li, M. Zhao, L. Ju, C. J. Xue, and Z. Jia, "Maximizing forward progress with cache-aware backup for self-powered non-volatile processors," in *Proc. IEEE/ACM DAC*, 2017, pp. 1–6.
[30] Q. Li, M. Zhao, J. Hu, Y. Liu, Y. He, and C. J. Xue, "Compiler directed automatic stack trimming for efficient non-volatile processors," in *Proc. IEEE/ACM DAC*, 2015, pp. 1–6.
[31] Z. Li *et al.*, "HW/SW co-design of nonvolatile IO system in energy harvesting sensor nodes for optimal data acquisition," in *Proc. ACM/IEEE DAC*, 2016, pp. 1–6.
[32] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Prof. ICML*, 2016, pp. 2849–2858.
[33] Y.-C. Lin, P.-C. Hsiu, and T.-W. Kuo, "Autonomous I/O for intermittent IoT systems," in *Proc. ACM/IEEE ISLPED*, 2019, pp. 1–6.
[34] Y. Liu *et al.*, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Proc. ACM/IEEE DAC*, 2015, pp. 1–6.
[35] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *Proc. SNAPL*, 2017, pp. 1–14.
[36] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proc. ACM PLDI*, 2015, pp. 575–585.
[37] K. Ma *et al.*, "Nonvolatile processor architecture exploration for energy-harvesting applications," *IEEE Micro*, vol. 35, no. 5, pp. 32–40, Sep./Oct. 2015.
[38] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," in *Proc. ACM OOPSLA*, 2017, pp. 1–30.
[39] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *Proc. USENIX OSDI*, 2018, pp. 129–144.
[40] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proc. ACM PLDI*, 2019, pp. 1101–1116.
[41] H. R. Mendis and P.-C. Hsiu, "Accumulative display updating for intermittent systems," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–22, 2019.
[42] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," in *Proc. ACM ASPLOS*, 2011, pp. 159–170.
[43] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
[44] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018. [Online]. Available: arXiv:1804.03209.
[45] J. V. D. Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proc. USENIX OSDI*, 2016, pp. 17–32.
[46] M. Xie, C. Pan, M. Zhao, Y. Liu, C. J. Xue, and J. Hu, "Avoiding data inconsistency in energy harvesting powered embedded systems," *ACM Trans. Design Autom. Electron. Syst.*, vol. 23, no. 3, pp. 1–25, 2018.
[47] M. Xie, M. Zhao, C. Pan, J. Hu, Y. Liu, and C. J. Xue, "Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor," in *Proc. ACM/IEEE DAC*, 2015, pp. 1–6.

Chih-Kai Kang (Graduate Student Member, IEEE) received the B.S. and M.S. degrees from the Department of Electronic and Computer Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, in 2010 an 2012, respectively. He is currently pursuing the Ph.D. degree with the Graduate Institute of Electrical Engineering, National Taiwan University, Taipei.

His current research interests include intermittently powered embedded systems, FPGA-based acceleration, and near-memory computing.

Hashan Roshantha Mendis (Member, IEEE) received the master's degree in digital systems engineering (electronics) and the Eng.D. degree in large-scale complex IT systems in the area of real-time systems from the University of York, York, U.K., in 2011 and 2017, respectively.

He is currently a Postdoctoral Research Fellow with the Research Center for Information Technology Innovation, Academia Sinica, Taipei, Taiwan. His research interests include resource management for many-core systems, intermittently powered embedded systems, and embedded deep learning.

Chun-Han Lin (Member, IEEE) received the B.S. and M.S. degrees from the Department of Computer Information Science, National Chiao Tung University, Hsinchu, Taiwan, in 2000 and 2002, respectively, and the Ph.D. degree from the Department of Computer Science, National Tsing Hua University, Hsinchu, in 2010.

He is currently an Assistant Professor with the Department of Computer Science and Information Engineering, National Taiwan Normal University, Taipei, Taiwan. His research interests include embedded systems and sensor networks.

Ming-Syan Chen (Fellow, IEEE) received the Ph.D. degree in computer, information, and control engineering from the University of Michigan at Ann Arbor, Ann Arbor, MI, USA, in 1988.

He is currently the NTU Chair Professor and also an Executive Vice President of National Taiwan University, Taipei, Taiwan, where he is responsible for research and academia affairs. Prior to this position, he was the Dean of the College of Electrical Engineering and Computer Science. He was a Research Staff Member of the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA, from 1988 to 1996, the Director of GICE from 2003 to 2006, the President/CEO of the Institute for Information Industry, which is one of the largest organizations for information technology in Taiwan from 2007 to 2008, and also a Distinguished Research Fellow and the Director of the Research Center of Information Technology Innovation (CITI), Academia Sinica, Taipei, from 2008 to 2015. He has published over 350 papers in his research areas. His research interests include databases, data mining, machine learning, and multimedia networking.

Dr. Chen is a Fellow of ACM and the Chinese Society for Management of Technology.

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

He is currently a Research Fellow (Professor) and the Deputy Director of the Research Center for Information Technology Innovation (CITI), where he leads the Embedded and Mobile Computing Laboratory, and is also a Joint Research Fellow with the Institute of Information Science, Academia Sinica, Taipei, a Jointly Appointed Professor with the Department of Computer Science and Engineering, National Chi Nan University, Nantou, Taiwan, and a Jointly Appointed Professor with the Data Science Degree Program, National Taiwan University. His research interests include embedded systems, mobile systems, and real-time systems.

Dr. Hsiu is a recipient of the 2019 Young Scholars' Creativity Award of the Foundation for the Advancement of Outstanding Scholarship, the 2019 Exploration Research Award of the Pan Wen Yuan Foundation, and the 2015 Scientific Paper Award of the Y. Z. Hsu Science and Technology Memorial Foundation. He serves as an Associate Editor for the *ACM Transactions on Cyber-Physical Systems* and in the Program Committees of many international conferences in his field, such as IEEE/ACM CODES+ISSS, DAC, and ISLPED. He joined CITI as an Assistant Research Fellow in 2009, and was promoted to an Associate Research Fellow in 2013 and to a Research Fellow in 2018. He is a Senior Member of ACM.