Intermittent-aware Distributed Concurrency Control

Wei-Che Tsai, *Student Member, IEEE,* Wei-Ming Chen, *Member, IEEE,* Tei-Wei Kuo, *Fellow, IEEE,* and Pi-Cheng Hsiu, *Senior Member, IEEE*

***Abstract*—Internet-of-Things (IoT) devices are gradually adopting battery-less, energy harvesting solutions, thereby driving the development of an *intermittent computing* paradigm to accumulate computation progress across multiple power cycles. While many attempts have been made to enable standalone intermittent systems, little attention has focused on IoT networks formed by intermittent devices. We observe that the computation progress improved by *distributed task concurrency* in an intermittent network can be significantly offset by data unavailability due to frequent system failures.**

This paper presents an intermittent-aware distributed concurrency control protocol which leverages existing data copies inherently created in the network to improve the computation progress of concurrently executed tasks. In particular, we propose a borrowing-based data management method to increase data availability and an intermittent two-phase commit procedure incorporated with distributed backward validation to ensure data consistency in the network. The proposed protocol was integrated into a FreeRTOS-extended intermittent operating system running on Texas Instruments devices. Experimental results show that the computation progress can be significantly improved, and this improvement is more apparent under weaker power, where more devices will remain offline for longer duration.

***Index Terms*—Distributed task concurrency, data consistency, battery-less devices, intermittent networks.**

I. Introduction

Internet of Things (IoT) applications have become increasingly ubiquitous in many domains like environmental monitoring, which rely on ultra-low power lightweight devices with sensing, computing, and communication capabilities [45]. To ensure the sustainability of such devices, energy harvesting has been proposed as a promising alternative to battery charging [25, 26], thus sparking innovative applications such as self-powered ambient light [53] and audio sensors [33]. However, ambient power sources are inherently weak and unstable, resulting in *transient* power on and off cycles [37]. Consequently, applications on battery-less devices have to be executed *intermittently* [29, 42, 52]. This unique execution behavior rules out existing approaches developed for *fault tolerant networks* [4, 54], which primarily address connection failures between nodes, and also those for *robust distributed systems* [10, 41] or *database systems* [7, 8, 22], which additionally consider occasional system failures of nodes. By contrast, *intermittent systems* suffer from extremely frequent system failures, significantly increasing the difficulty of designing both hardware and software solutions [28, 43], especially for a network formed by intermittent nodes which suffer from frequent and arbitrary power failures [18].

The research challenge for intermittent systems mainly lies in accumulating computation progress while ensuring data consistency across power cycles. Borrowed from traditional systems, numerous variations based on *checkpointing* have been proposed for intermittent systems to accumulate computation progress [6, 15, 27, 39, 44], while accommodating correct system recovery from frequent power failures [36, 50, 51]. Many attempts, such as *differential checkpointing* [3, 5] and *stack trimming* [32, 55], have also been made to reduce the checkpointing overhead. However, checkpointing requires system suspension and can impose significant runtime overhead, which scales with the checkpoint size and frequency, and system recovery by restoring all checkpointed data may not be timely completed when the available energy is scarce. Alternatively, *task-based* approaches partition an application program into multiple atomic tasks and only update the modified data after each task completion [16, 35, 38]. Upon power resumption, the interrupted task is recreated and rerun from the beginning. Existing intermittent execution approaches differ in what data are backed up during system execution and how much progress is re-executed upon power resumption.

In contrast to other task-based approaches that assume *serial task execution*, a *failure-resilient design* [11, 13] has recently been proposed to enable *intermittent-aware task concurrency*, where multiple tasks that share data on the same device can be executed in an interleaving manner to improve computation progress. To guarantee the consistency of shared data when accessed simultaneously by different tasks, *concurrency control* is employed to enforce *serializability*, *atomicity*, and *durability* [8]. Specifically, serializability ensures that the resultant data values produced by concurrently executed tasks are equivalent to the values when these tasks are executed serially in some arbitrary order. To accommodate frequent yet transient power failures, atomicity is enforced to update the modified data in an all-or-none manner to avoid data corruption due to partial updates, while durability is enforced to recover the data lost due to a system failure to a consistent

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

Wei-Che Tsai was with the Research Center for Information Technology Innovation (CITI), Academia Sinica, Taipei 11529, Taiwan, and also with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 10617, Taiwan (e-mail: weiche@ntu.edu.tw).

Wei-Ming Chen was with the Research Center for Information Technology Innovation (CITI), Academia Sinica, Taipei 11529, Taiwan, and is now with Microsystems Technology Laboratories, Massachusetts Institute of Technology, Cambridge, MA 02139, USA (e-mail: wmchen@mit.edu).

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

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

state.

Most prior research on intermittent computing focused on a *standalone* device. Recently, a few attempts were made to support communication peripherals for intermittent devices [9, 34, 40]. The challenge for device-to-device intermittent communication primarily lies in ensuring both the transmitter and receiver are active simultaneously for at least the airtime of one complete packet. Accordingly, hardware-supported [21, 47] and software-assisted solutions [49] were developed to increase the probability of successful packet transmission between two intermittent nodes, making intermittent IoT networks increasingly possible. However, there has been comparatively little research on (even small-scale) networks formed by intermittent nodes, and the impact of intermittency on IoT networking has received little attention.

This paper presents the first attempt to address intermittent-aware distributed concurrency control, which is essential for multiple tasks executed concurrently on different intermittent devices to access shared data. We observe that the computation progress improved by task concurrency can be significantly offset by *data unavailability* due to intermittency. To increase data availability, we propose a distributed concurrency control protocol, which allows tasks to borrow existing data copies *inherently* created on any online devices having ever accessed the same data. Our protocol uses a *borrowing-based data management method* to enable unavailable data to be borrowed from the network as if they were directly accessed from the original devices.

However, this flexibility raises two design challenges. The first challenge is how to atomically commit data modifications based on the borrowed copies back to their original devices and, despite frequent power failures, how to recover data objects scattered on different devices to a consistent state. We enforce atomicity and durability by an *intermittent two-phase commit procedure*, which allows data modifications to be accumulatively committed onto intermittently-powered devices, while maintaining data consistency on different devices at all times. The second challenge is to enforce the serializability of tasks executed concurrently based on the original and also borrowed data copies. This challenge is addressed by a *distributed validation algorithm* with the consideration of *task dependency* implicitly created due to data borrowing. Accordingly, our protocol increases data availability while guaranteeing data consistency in an intermittent network.

Our protocol was integrated into a FreeRTOS-extended intermittent operating system [2] and deployed on a small-scale intermittent network formed by Texas Instruments (TI) MSP430 devices to evaluate its performance by *forward progress*[1]. Experiments were conducted based on a simplified smart farming application, under different power sources and network sizes. Compared to a distributed concurrency control protocol that integrates the failure-resilient design [11] and two-phase commit protocol [23], our protocol improves forward progress by 20 to 33% by increasing the data availability, where the improvement is more evident under a weaker power source and in a larger network. We also conduct a breakdown analysis on the *blocking time* and *communication overhead* to provide useful insights and suggest potential extensions for intermittent-aware distributed concurrency control.

The remainder of the paper is organized as follows. Section II provides background information and Section III explains the motivation for this work. In Section IV, we present our protocol, with system implementation issues discussed in Section V. Experimental results are reported in Section VI, and Section VII discusses the limitations of our protocol and potential extensions. Finally, we draw our conclusions in Section VIII.

II. Background

A. Distributed Concurrency Control

An IoT network comprises multiple lightweight devices connected via low-datarate wireless communication to cooperatively perform a joint application, as shown in Figure 1. Each device runs multiple tasks that concurrently access data objects in hybrid memory comprising volatile memory (VM) and non-volatile memory (NVM). Moreover, tasks executed on any device (denoted as an *accessing* node) can read and write data objects maintained by other devices (denoted as *owner nodes*), thereby allowing tasks to concurrently run on different devices to improve overall computation progress. Note that any device can simultaneously be an accessing node that accesses remote data objects and an owner node that shares local data objects with others. However, because a device only maintains its local tasks and data objects, the data consistency among devices need to be maintained by the cooperation of multiple nodes in the network.

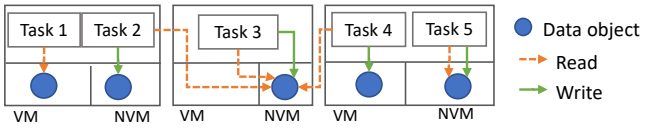

Fig. 1: Multiple tasks sharing data in a distributed IoT network

To tolerate temporary network connection and/or node failures, a distributed control protocol, which may rely on some *data synchronization* [10, 41] or *recovery mechanism* [7, 8, 22], is essential to maintain the consistency of shared data objects. When task concurrency is allowed, a distributed network commonly uses *validation*-based concurrency control [30], which eliminates complicated lock management, to enforce serializability, atomicity, and durability across multiple nodes. Specifically, all data modifications made by tasks are in their own working space, and whenever a task attempts to commit its data modifications, the serializability is enforced through a distributed backward validation algorithm [7, 23], where a joint validation decision is made based on the partial result validated by each owner node. If serializability is not violated, the task proceeds to commit; otherwise, it is aborted and rerun.

Moreover, the committed data modifications must not cause data corruption due to partial updates. To ensure atomicity, once a task is validated as serializable, its data modifications must be committed in an all-or-none manner [31], typically via some consensus protocol like *two-phase commit* [23], to prevent the inconsistency of data objects across nodes. In

[1]Forward progress, defined as the computation workload finished within a given period of time, is a major metric widely used to evaluate intermittent system performance when the task completion time varies significantly from run to run due to non-deterministic power failure periods.

the first phase, all participant nodes reach a consensus on whether to commit, while in the second phase, they obey the consensus to commit either all or none of the data modifications. Despite a system failure, to enforce durability, the modifications committed into persistent memory (or storage) must be recoverable to a consistent state [48]. Traditionally, some mechanism like *logging-based checkpointing* [48] is employed on every node to log all data access operations at runtime and periodically checkpoint the data modifications and logs into persistent memory, allowing the data objects scattered across the network to be recovered to a consistent state via a distributed recovery procedure [8, 22].

B. Intermittent-aware Concurrency Control

Figure 2 shows a typical hardware architecture of an intermittent device. To operate with weak ambient power, the device uses an energy management unit to buffer harvested energy into a capacitor. When the capacitor voltage reaches an upper (resp. lower) threshold, the device is powered on (resp. off), leading to intermittent execution. To accumulate computation progress across power cycles, the device is equipped with hybrid memory that comprises VM and NVM connected to the memory bus and directly accessible by the CPU. The VM features high access speed but suffers from data loss upon power failures, whereas the NVM is just the opposite. Early studies on intermittent systems use *checkpointing* [6, 27, 44] to accumulate computation progress and *logging* to accommodate system recovery. Specifically, during checkpointing, tasks are suspended to backup their data and contexts from VM to NVM, and during recovery, the system is restored to the latest checkpoint, while the logs saved in NVM are traversed to undo partially updated data and redo interrupted tasks. Consequently, logging-based checkpointing can recover the system to a consistent state but suffers from long runtime suspension and recovery.

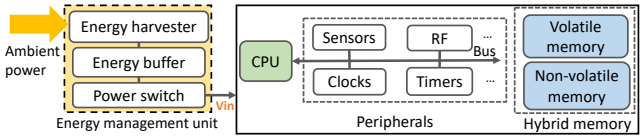

Fig. 2: System architecture of an intermittent device

When multiple tasks are concurrently executed on an intermittent system, which could experience extremely frequent power failures (in milliseconds or seconds, depending on the capacitor used and the energy consumed by executed tasks), the suspension overhead incurred by checkpointing can offset the progress improvement achieved by task concurrency [12], and also system recovery based on logging may not be timely completed within such a short power-on period [14]. In contrast, task-based approaches only update the data modifications after each task is completed, alleviating the runtime overhead to allow for more frequent system failures [16, 38], at the cost of potential overall task re-execution. Recent work has proposed a *failure-resilient design* [11] using two-version data to enable task concurrency on an intermittent system without checkpointing or logging. Specifically, a unique *consistent* version is kept to be always consistent with the progress of finished tasks and is available in NVM after power resumption. Moreover, a *working* version is dedicated for each task to store its data modifications in VM and can be atomically committed into the consistent version in NVM only if the task is serializable. To eliminate runtime checkpointing, a task can only commit its working version at the end of execution, and once the working version is successfully committed, the task is deemed finished to prevent the consistent version from being repeatedly modified, protecting *idempotence* [36]. Unfinished tasks are volatile in VM upon power failures as if they had never been executed, and after power resumption, they are directly recreated based on the consistent version, achieving instant recovery without logging.

The failure-resilient design, which was originally developed for a standalone intermittent node, can be extended and applied to an intermittent network formed by multiple intermittent nodes. The two-phase commit protocol [23], in which the participant nodes are required to record the execution states of the protocol, allows the consensus to be achieved in an accumulative manner, without all participant nodes being online simultaneously. Accordingly, whenever being prepared to finish, a task executed on a node can simply use the two-phase commit protocol, with the backward validation algorithm [11] incorporated, to atomically commit its data modifications into the corresponding consistent versions on different nodes if the task is validated as serializable, while the consistent versions can be used to correctly recreate and rerun unfinished tasks interrupted due to power failures, thereby enabling distributed task concurrency in an intermittent network.

III. Observation and Motivation

This work is complementary to prior research on standalone intermittent systems, in that checkpointing overhead is a decisive factor for an intermittent device to efficiently accumulate computation progress across power cycles, while *data unavailability* might have a significant impact on the overall computation progress in an intermittent network. To investigate the impact of intermittency on IoT networking, we conducted an experiment on an intermittent network comprising three TI MSP430 devices. Each device is equipped with a 40mF capacitor as the energy buffer, and a TI CC1101 Radio Frequency (RF) transceiver to communicate with the other devices. A FreeRTOS-extended operating system, which integrates the failure-resilient design and the two-phase commit protocol (as described in Section II-B), was installed on each device to enable distributed task concurrency in the intermittent network.

Considering that intermittent devices are typically intended for low-datarate, lightweight applications, we emulated a simplified smart farming application running on the three-node intermittent network to monitor the amount of sprinkled water. One device serves as the owner node, which holds a shared data object representing the quantity of water, while the other two devices serve as accessing nodes. Three respective tasks that repeatedly run on the nodes concurrently read the shared data object, calculate the amount of water to be sprinkled, and then commit their respective values to the data object. This experimental setup emulates a simple IoT application, where three concurrently executed tasks share a data object in a network.

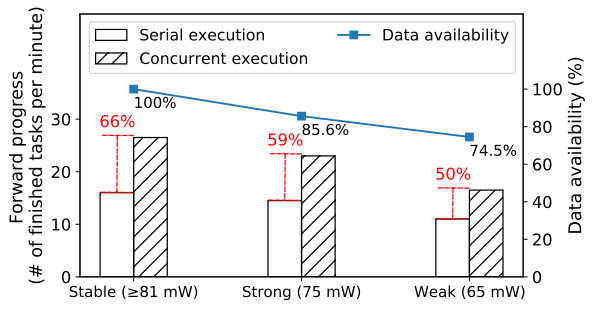

Fig. 3: Forward progress vs. data availability

We measure the *forward progress* (defined as the number of finished tasks per minute) achieved respectively by serial and concurrent task execution, under different power sources manufactured by a programmable power supply. Three power sources, namely stable (≥81 mW), strong (75 mW), and weak (65 mW), are used to emulate different ambient power scenarios and thus different amounts of data availability. The data availability is defined as the probability that the data object is available when a task attempts to access it remotely. As shown in Figure 3, concurrent task execution can significantly improve the forward progress achieved by serial task execution. However, as the data availability decreases, the improved forward progress drops substantially. Specifically, under stable power, when the data availability is 100%, task concurrency can improve the forward progress by 66%. When the power source is insufficient to operate the devices continuously, depending on the degree of intermittency, the data availability drops to 85.6% under strong power and 74.5% under weak power, while the improvements respectively drop to 59% and 50%. The result shows that data unavailability has a significant impact on not only the forward progress of an intermittent network, but also the efficacy of distributed task concurrency.

We further investigate the key cause. During serial task execution, the task will be blocked and yield the CPU to other tasks if the remote data object is unavailable and, whenever the task is scheduled to occupy the CPU, it will try to access the remote data object again. The process will be repeated until the owner node recovers from the power failure. This not only prolongs the task execution time, but also wastes the harvested energy and shortens the online duration of the node. The data unavailability has an even more adverse impact on concurrent task execution, because more tasks sharing the same data object would be blocked by the offline owner node simultaneously. Under weaker power, where more owner nodes would remain offline for longer duration, the forward progress improved by task concurrency would be more significantly offset by data unavailability. This observation provides the motivation for intermittent-aware distributed concurrency control, and inspires our protocol design.

IV. Intermittent-aware Distributed Concurrency Control

A. Design Rationale and Challenges

To increase data availability, we propose a borrowing-based distributed concurrency control protocol, which leverages existing data copies inherently created in an intermittent network for tasks to make forward progress even when owner nodes are currently offline. Note that these copies inherently exist on those nodes having ever accessed the same data objects (even without our protocol), and the memory spaces of the copies can be reclaimed when they become invalid. This is different from *data replication* stored permanently on multiple nodes with the replicates kept consistent via *data synchronization*, as widely used in traditional distributed systems.

To enable tasks to borrow unavailable objects from other nodes as if the data objects were directly accessed from their owner nodes, we propose a *borrowing-based data management* method in Section IV-C, where a data object can have multiple versions and copies to be simultaneously accessed and borrowed by multiple tasks. However, allowing data borrowing raises additional challenges.

First, to enforce atomicity and durability, the data modifications based on the borrowed copies should be atomically committed onto the corresponding owner nodes, and despite frequent power failures, data objects scattered on different owner nodes must be recoverable to a consistent state. This is particularly challenging because online nodes may arbitrarily encounter power failures during a distributed recovery procedure and invoke new recovery procedures, thus stagnating the intermittent network with non-terminated distributed recovery. In Section IV-C, we propose an *intermittent two-phase commit protocol* to atomically commit the data modifications of a task onto different owner nodes in an accumulative manner, while maintaining the consistency of data objects on owner nodes at all times, thereby eliminating the need for a distributed recovery procedure.

The second challenge is to allow multiple tasks to be concurrently executed based on different versions and copies, while maintaining serializability. Serializability validation becomes more complicated because allowing tasks to borrow data copies that have been modified on other nodes (denoted as *source nodes*) creates implicit task dependency, where a task will become non-serializable if any borrowed copies cannot be committed by the source nodes due to serializability violation. In Section IV-D, we propose a *distributed validation algorithm* with task dependency consideration, to validate the serializability of a task that can access and borrow data objects from multiple nodes.

B. Borrowing-based Data Management

1) Data Versions and Copies: We enable borrowing-based concurrency control by multi-version data management, where each data object has two versions, namely the *consistent* version and the *working* version. The consistent version represents the latest value committed by a *serializable* task and is persistently preserved in NVM on the corresponding owner node. In contrast, working versions temporarily store intermediate values written by unfinished tasks and will be lost in VM on the respective accessing nodes. Two additional copies, called the *duplicated* copy and the *modified* copy derived from the consistent version and the working version inherently during task execution, can be leveraged to increase data availability.

As shown in Figure 4, when a task reads the consistent version of a data object, which can be either local or remote, a duplicated copy will inherently be created in local VM on

the accessing node. Then, the duplicated copy will be kept unmodified and can be read by other tasks by default to utilize high speed VM, thereby avoiding slow data access to the consistent version every time. Consequently, if the task attempts to first modify the data object, a working version dedicated for the task will be created in a copy-on-write manner in local VM to store its intermediate value. At the end of execution, the working version of the task will be inherently saved into local NVM as the modified copy to avoid re-execution due to power failures. If power fails before the modified copy is completely saved, the task will be recreated and rerun after power resumption, as if it had never been executed. In contrast, the modified copy will be committed back to the consistent version on the owner node, and the task is deemed finished afterward.

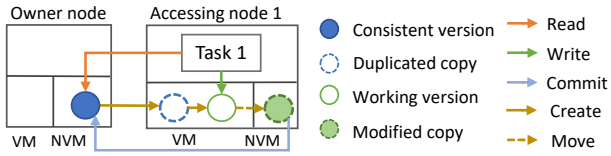

Fig. 4: Data versions and copies created during execution

2) Borrowing Strategy: When a task attempts to read a data object, but the corresponding owner node is offline, any duplicated or modified copy of the data object in the network can be borrowed instead, as shown in Figure 5, preventing the task from being blocked due to data unavailability. A duplicated copy can be borrowed as if the task directly read the original consistent version of the data object on the owner node. Similarly, a modified copy can be borrowed as if the task early read an updated consistent version. If multiple copies exist in the network, the latest modified copy that represents the latest value in the network is always preferred, thereby improving forward progress in an optimistic manner. Moreover, the borrowed copies can further be borrowed by other tasks executed on the same or different accessing nodes. However, if any task creating the copies is aborted due to serializability violation, the borrowed copies will be deleted to prevent them from being further borrowed by other tasks. To borrow data copies from the network, we use a broadcasting implementation, which will be detailed in Section V-A.

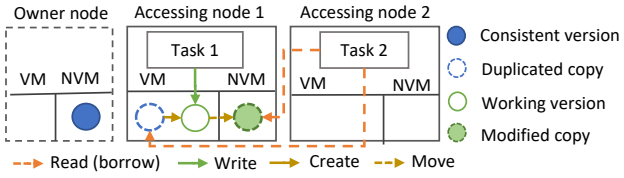

Fig. 5: Borrowable data copies

C. Intermittent Two-phase Commit

1) Atomic Commit and Instant Recovery: Two-phase commit is a standard technique used to atomically commit data modifications to distributed nodes. Our protocol additionally defines when and what to commit, thereby eliminating the need for a distributed recovery procedure to accommodate frequent and arbitrary node failures. We borrow a similar idea from [11], where the consistent versions are always kept consistent with the progress of finished tasks on a single intermittent node. To this end, a task can only update the consistent versions via an atomic commit operation at the end of execution, so that tasks left unfinished due to a power failure are simply lost in VM and can be directly recreated and rerun based on the consistent versions in NVM after power resumption, enabling instant recovery. Differently, our atomic commit procedure allows the consistent versions scattered across the network to be always kept consistent with the progress of tasks finished on different nodes. Accordingly, to prevent the consistent versions from being partially updated due to power failures or repeatedly updated by finished tasks, an accessing node can only commit a serializable task's modified copies atomically into the consistent versions on their owner nodes via a two-phase commit procedure, as follows.

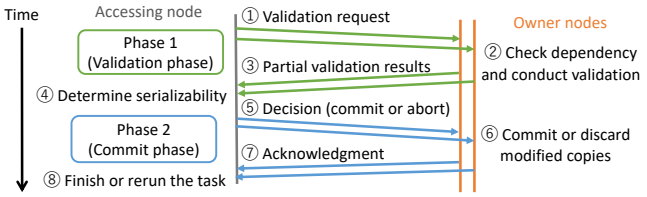

Fig. 6: Two-phase commit with backward validation

The two-phase commit procedure incorporates a backward validation algorithm for all participant nodes to reach a consensus on the abortion of the task or the commitment of its modified copies. As shown in Figure 6, in the first phase, the accessing node sends a validation request, along with the task identity and its corresponding modified copies, to each participant owner node and then waits for its partial validation result. Upon receipt of all the partial validation results, the accessing node jointly validates the serializability of the task and makes a final decision. Note that once the consensus is reached, all participant nodes must obey and eventually complete the task abortion or commitment to avoid data inconsistency in the network. If serializability is not violated, in the second phase, the accessing node asks each owner node to atomically commit the received modified copies; otherwise, it aborts the task and asks the owner nodes to discard the received modified copies. Upon receipt of the acknowledgements from all owner nodes that the modified copies have been committed or discarded, the accessing node finishes or reruns the task accordingly, and then the two-phase commit procedure ends.

In an intermittent network, nodes typically suffer from frequent power failures and may not be online at the same time. To allow the two-phase commit procedure to be completed in an accumulative manner, the accessing node that invokes the procedure records the current phase. When some owner nodes are offline, the accessing node will keep sending requests of partial validation to those nodes in the first phase, or keep sending the joint validation decision in the second phase. Similarly, when the accessing node is offline, the owner nodes wait for the accessing node to send the validation request again and reply with their partial validation results in the first phase, or wait for the joint validation decision and reply with their acknowledgements in the second phase. We use a polling-based implementation to avoid request flooding. Implementation issues will be detailed in Section V-A.

Because updating the consistent version of any data object must be mutually exclusive, it can only be updated via a two-phase commit procedure invoked by one accessing node at a time. Thus, after completely receiving a validation request, an owner node first checks whether any of the data objects related to this request are currently involved in another commit procedure. If neither is currently involved, the owner node validates whether these data objects can be committed using the backward validation algorithm and then replies its partial validation result. Otherwise, the owner node holds off this request and keeps silent to the accessing node until all these data objects are not involved in another commit procedure. Consequently, two accessing nodes may wait for each other in their own first phases, resulting in a deadlock. We adopt a *timeout mechanism* [19] to break the deadlock by forcing the accessing nodes to abandon the current procedures. Implementation issues will be detailed in Section V-A.

2) Atomicity and Durability Enforcement: We enforce atomicity and durability by leveraging the characteristics of VM and NVM. For each data object, its versions and copies accessed by the three operations are appropriately allocated in the hybrid memory, as summarized in Figure 7. Although a task can read a data object via its consistent version, any task cannot directly write on the consistent version but on its own working version in local VM. At the end of execution, a task saves its working versions into local NVM as the modified copies, which are afterward atomically committed to the consistent versions on their owner nodes. Therefore, the consistent versions in NVM will never be affected during task execution and can only be atomically updated via a two-phase commit procedure after the task is validated as serializable, enforcing the atomicity.

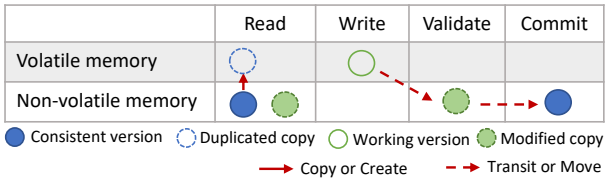

Fig. 7: Data version and copy allocation

Power failures may happen during task execution. After power resumption, if the modified copies have not been completely saved, the partially saved modified copies are discarded, while the working versions and duplicated copies are directly lost in VM, so the task can be directly recreated and rerun as if it had never been executed. In contrast, if the modified copies are completely saved, the task is not rerun, and the two-phase commit procedure directly resumes updating the consistent versions. Once the modified copies are successfully committed, the task is marked as finished to prevent the consistent versions from being repeatedly updated. Thus, despite power failures, the consistent versions will always remain consistent with the progress of finished tasks, which allows instant recovery while enforcing durability.

D. Distributed Validation with Task Dependency

1) Task Dependency Check: Distributed validation is common to guarantee task serializability across multiple nodes. Ours additionally resolves task dependency implicitly created due to data borrowing. Before a task can commit its modified copies, the task must be validated as serializable in the network via a distributed validation algorithm, which shares a similar idea with a centralized validation algorithm [11, 13]. Differently, the previous algorithm is intended for a single intermittent node and thus cannot validate the serializability of a task that can access remote data objects on multiple intermittent nodes, especially when the task is also allowed to borrow duplicated and modified copies from the network. Data borrowing creates implicit *task dependency*, where a task that has ever borrowed any modified copies from other nodes can commit its modified copies only after all the related modified copies on the source nodes are successfully committed into the corresponding consistent versions. If any modified copy on some source node cannot be committed due to serializability violation, the task also becomes non-serializable. As a result, once a non-serializable task is aborted, all the tasks that have ever borrowed its modified copies should also be aborted, resulting in a cascaded abortion.

To resolve the dependency of a task having borrowed modified copies from the network, the accessing node queries the corresponding owner nodes about whether the modified copies on the source nodes have been committed into their consistent versions. An owner node replies a positive if all related modified copies have been committed, and a negative if any of the modified copies cannot be committed due to serializability violation. If neither a positive nor a negative has been determined, an owner node remains silent as if it were offline. The accessing node repeatedly queries the owner nodes until it receives all the positives or any negatives. Note that the modified copies on the source nodes will be either committed or discarded, so the accessing node will eventually receive all the positives or any negatives. Upon receipt of any negative, the accessing node aborts and reruns the task. In contrast, task dependency is resolved, and the task can proceed to be validated for serializability.

2) Distributed Backward Validation: Our distributed validation algorithm extends the centralized validation algorithm [13]. To maintain a serializable order, each finished task is associated with a *validity time interval*, in which the task can be executed *logically* in isolation. The centralized algorithm derives the validity time interval of the task under validation according to its read and write operations (recorded in VM during task execution), in an attempt to serialize the precedence relationship between the task and all finished tasks which have ever accessed some of the same data objects. Specifically, when a task reads (or writes) some data object, its interval should shrink to avoid overlapping with the interval of any finished task that has committed the same object, so that it can viewed as finished before (or started after) the latest finished task that shares objects with the task. If a nonempty interval can be derived for the task, the task is serializable; otherwise it is not.

As shown in Figure 6, we incorporate the centralized validation algorithm into our intermittent two-phase commit procedure, allowing validation to be completed in a distributed manner without all nodes being online simultaneously. To validate a task, in the validation phase (Phase 1), the accessing node sends each participant owner node a request of partial

validation. The request carries the task identity and its modified copies to be committed onto the owner node, as well as all read and write operations on the related data objects. Provided that task dependency has been resolved, each owner node uses the centralized validation algorithm to derive a time interval based on the received request and all serializable tasks that have committed some of the same data objects on the owner node. The derived time interval (which represents the partial validation result) is then replied to the accessing node. Upon receipt of all the time intervals, the accessing node derives their *intersection* as the validity time interval for the task, thereby determining serializability. In the commit phase (Phase 2), the accessing node sends the validity time interval (which implies the decision) to the owner nodes, and then all participant nodes obey the consensus to complete the task abortion or commitment, as presented in Section IV-C.

Note that the respective inputs required for the dependency check procedure and the validation algorithm can be sent together via one single communication, while their outputs can be combined and replied via another communication, as shown in the validation phase in Figure 6. Specifically, once the task dependency is found to be insolvable, the owner node can reply an empty interval, in substitution for a negative reply, to the accessing node. In contrast, if the task dependency on the owner node is resolvable, the owner node can just reply the time interval derived by the validation algorithm, with the positive reply omitted. However, our implementation introduces two extra communications to decrease the probability of a deadlock, by reducing the number of accessing nodes waiting for one another in their own validation phases. More details will be discussed in Section V-B.

3) Serializability Enforcement: Previous work [13], where all tasks that share data objects run on the same intermittent node, has proven that any serializable task validated by the centralized algorithm is *conflict-serializable* [24]. The idea is to show that the to-be-validated task has a non-empty validity time interval which does not overlap with the validity time interval of any serializable task that has committed some data objects shared with the task. Consequently, the task can be viewed as having been executed in isolation in its validity time interval. Similarly, for any task validated by the proposed distributed algorithm, its serializability can be proved by showing that the intersection of the time intervals replied by all participant owner nodes does not overlap with the validity time interval of any serializable task that has committed some data objects shared with the task.

During our distributed validation algorithm, which is mutually exclusive, no tasks other than the task under validation will be validated as serializable. Because each owner node uses the centralized algorithm in [13], the time interval derived for the task must not overlap with the validity time interval of any serializable task that has committed some shared data objects on the owner node. To make a joint validation decision, the accessing node will eventually receive the time intervals from all participant owner nodes, and their intersection must be within all those received intervals. Thus, the intersected interval will not overlap with the validity time interval of any serializable task that has committed some data objects shared with the task. Thus, serializability is enforced.

V. An Operating System for Intermittent Networks

To realize an operating system for intermittent networks, we integrated our borrowing-based distributed concurrency control protocol into an intermittent operating system [2], which was built upon FreeRTOS according to the failure-resilient design intended for standalone intermittent systems [13]. Our protocol is realized on top of the task scheduler and memory manager, as shown in Figure 8, with around 4800 lines of C code. The key requirement for the integration is that the operating system must still accommodate a power failure at an arbitrary line of code. Our operating system is then deployed on several TI MSP430 devices, equipped with TI CC1101 RF transceivers to form an intermittent network. Since node communications are time and energy consuming, our implementation principle is to reduce the number of communications.

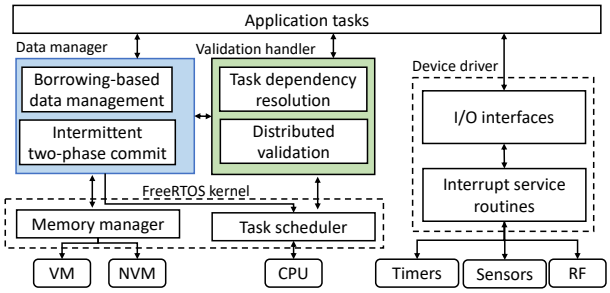

Fig. 8: FreeRTOS-extended system architecture

A. Data Manager

The data manager maintains multiple data versions and copies. Apart from local data manipulation supported in [2], our data manager allows remote data manipulation. When a task attempts to access a remote object, the data manager on the accessing node will keep broadcasting a request with the data identity to the network until the desired object is received. Upon receipt of the request, the data manager on another node will reply with its consistent version or duplicated/modified copy (with a timestamp representing data freshness) if it exists. If the data manager requesting the data object receives more than one version or copy, it passes one of them to the task according to our borrowing strategy in Section IV-B. The broadcasting implementation requires fewer node communications than querying the nodes one by one.

When a task attempts to commit its modified copies, the data manager will invoke a two-phase commit procedure presented in Section IV-C to reach a consensus. If a commit consensus is reached, each participant owner node uses the *shadowing mechanism* implemented in [13] to atomically save all the received modified copies into local NVM. The consistent version of a data object can only be updated by one commit procedure at a time. To enforce a mutually exclusive update, the data manager maintains a queue to record those currently involved data objects. If two nodes wait for each other to complete their commit procedures, a deadlock occurs. We employ a timeout mechanism [19], where a node that cannot timely complete its commit procedure should ask all participant owner nodes to abandon the current procedure and

re-invoke the procedure later again. The timeout is fixed at 3 seconds in our implementation but can also be dynamically determined for each commit procedure depending on factors like power stability and the number of participant nodes.

The two-phase commit procedure can be completed in an accumulative manner, without all participant owner nodes being online simultaneously. To this end, the accessing node invoking the commit procedure records the current phase and those owner nodes having yet to reply. Despite power instability, the accessing node, upon recovery, will keep polling those owner nodes until it collects all necessary replies in each phase. A power-interrupted reply will be resent rather than resumed. To enable an interrupted request or reply to be resumable, the wireless module needs to be compatible with some I/O-based checkpoint mechanism [34], which is not applicable to the used RF transceiver. In an intermittent network, the polling-based implementation usually incurs a smaller number of undeliverable replies than a pushing-based implementation, in which all participant owner nodes also record the procedure states and proactively resend replies.

B. Validation Handler

The validation handler performs the distributed validation algorithm while resolving potential task dependency, as presented in Section IV-D. A task may borrow multiple data copies from different nodes. To reduce the number of communications during the dependency check, the accessing node asks the owner node, instead of those source nodes, to check whether those modified copies on the source nodes have been committed, cannot be committed, or have yet to be committed. The three cases respectively indicate that the tasks creating the modified copies are finished, aborted, and unfinished, based on which task dependency can be resolved accordingly.

Because multiple nodes can check their task dependencies simultaneously without a conflict, our implementation reduces the mutually exclusive section of the two-phase commit procedure by excluding the task dependency check. To this end, we introduce two extra communications between an accessing node and each participant owner node, one for the owner node to inform whether the task dependency is resolvable, and the other for the accessing node to inform whether to invoke the validation algorithm. Given every accessing node invokes distributed validation only after the task dependency has been resolved, the number of accessing nodes waiting in their own validation phases will be reduced, thereby decreasing the probability of a deadlock. Such an implementation weighs extra communications for control information and ineffective communications due to deadlocks.

We integrate backward validation into the two-phase commit procedure to ensure mutual exclusion while reducing the number of communications. Furthermore, the distributed validation algorithm requires time synchronization among nodes to correctly derive time intervals. Whenever a node recovers from a power failure, it should synchronize its time with other online nodes in the network. To simplify the implementation, we use an always-on time keeping node. The time granularity is set as 200 ms, which is sufficiently coarse-grained to tolerate the RF transmission delay of a time-synchronization packet. To exempt an intermittent network from the time-keeping node, some *time-keeping mechanism* [17], with custom hardware support, could be adopted.

C. Node Communications

Compared with WiFi and Bluetooth modules, RF transceivers have lower energy consumption and thus are typically used for intermittent devices [47, 49]. Our prototype uses the TI CC1101 RF transceiver for node communications. However, this low-power transceiver supports neither *time-division multiple access* (TDMA) nor *frequency-division multiple access* (FDMA), and thus may suffer from collision when multiple nodes try to transmit simultaneously. To mitigate collision, we implemented a hardware-supported *clear channel assessment* (CCA) [46] in the transceiver's device driver, preventing multiple nodes from transmitting simultaneously. Before a node transmits, it uses the CCA to appraise the channel. If the channel is clear, it transmits immediately; otherwise, it retries again after backoff of 100 ms, which is the average packet transmission time.

Despite the channel clearance mechanism, the hardware used to implement the CCA cannot always appraise the channel accurately, resulting in increasingly severe channel collision with the network scale. We observed that packets could hardly be transmitted using the RF transceiver when a network comprises more than 5 nodes. Current ultra-low power transceivers support neither TDMA nor FDMA, resulting in severe channel collision. In contrast, transceivers supporting TDMA or FDMA are too energy-consuming to be driven by energy harvested in a capacitor. As a result, the off-the-shelf transceivers currently available limit the scale of our prototype network (to 4 intermittent nodes plus 1 time-keeping node when the TI RF transceiver is used). However, our protocol is applicable to larger-scale intermittent networks if ultra-low power wireless modules supporting multiple access and multi-hop relay routing become available for intermittent devices.

VI. Performance Evaluation

A. Experimental Setup

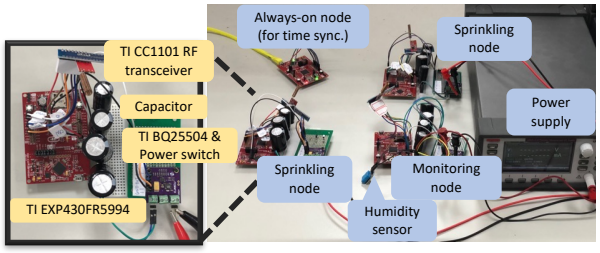

Fig. 9: The experimental environment

To evaluate the proposed protocol, we installed our FreeRTOS-extended intermittent operating system on TI MSP430 LaunchPad devices equipped with TI CC1101 RF Transceivers to form a small-scale intermittent network, as described in Section V. The experimental environment is shown in Figure 9. Each device is powered by an energy management unit (EMU) composed of a TI BQ25504 boost converter, a 40mF capacitor to buffer energy, and a power

switch to turn on (resp. off) the power supply when the capacitor voltage rises above 2.8 V (resp. drops below 2.4 V). Additionally, an always-on time keeping device is dedicated for time synchronization in the intermittent network. Table I details the specifications.

TABLE I: Hardware and software specifications

Hardware	
MCU	TI 16bit MSP430
Memory	8KB SRAM & 256KB FRAM
Software	
OS	FreeRTOS V10.1.0
Power	
Capacitance	40 mF
Switch On/Off Voltage	2.8 V / 2.4 V
Stable, Strong, Medium, Weak	81, 75, 70, 65 mW

We further expanded the simplified smart farming application used in Section III to run on an intermittent network of a monitoring node and (N-1) sprinkling nodes, where N= 3 or 4. Specifically, the monitoring node maintains two data objects representing respectively the humidity of soil and the quantity of water, as well as runs two tasks, where one task updates the humidity object by accessing a humidity sensor, while the other checks both objects to indicate whether the network is functioning correctly via an LED light. Each sprinkling node maintains a data object representing the amount of water to be sprinkled, and also runs two tasks, where one task remotely reads the humidity object to update the local sprinkle object, while the other reads the sprinkle object, sends a control packet to an external sprinkler, and then updates the sprinkle and remote quantity objects. These $2N$ tasks are repeatedly run on N nodes to concurrently read, write, and commit (N+1) shared data objects. The workload per node is fixed to ensure that, regardless of the network size, a node will suffer from a similar pattern of power failures under the same power supply. Although the currently available hardware does not allow us to implement a larger-scale intermittent network, the experimental setup is sufficient for us to evaluate the efficacy of data borrowing while drawing potential extensions.

To emulate energy harvesting, each device was powered by a Keithley 2280S programmable power supply. Four power sources, stable ($\geq$81 mW), strong (75 mW), medium (70 mW), and weak (65 mW), were manufactured to emulate average magnitude of power generated by a small (6cm^2) solar cell as the weather changes from sunny to cloudy [37]. Each power source lasted 10 minutes, which was sufficient to mitigate experimental variances while making experiments reproducible. Under stable power, the device operates continuously. In contrast, the other power sources were sufficient to finish at least one task in a single power cycle[2], but insufficient to operate the device continuously, leading to repeated yet unpredictable power failures. Under intermittent power, intermittent nodes can experience different power on and off cycles, where the online duration ranged from 4 to 10 seconds, while the offline duration was approximately 3 to 4 seconds, depending on the capacitor size, power strength, and task workload. Note that our protocol is developed for intermittent power, yet its incurred overhead can be evaluated under stable power.

A distributed control protocol can be orthogonal to intermittent execution approaches developed for a standalone device. We compared our borrowing-based protocol (denoted as OURS) with an intermittent-aware distributed concurrency control protocol (CON) and a serial execution protocol (SER). CON integrates the failure-resilient design [11] and the two-phase commit protocol [23] to enable distributed task concurrency, as described in Section II-B. The difference between OURS and CON is whether data borrowing is allowed, while the difference between SER and CON is that tasks are respectively executed serially and concurrently. All the protocols were respectively integrated into a FreeRTOS-extended intermittent operating system [2] for a fair comparison, and evaluated in terms of *forward progress*, which is a metric widely used to evaluate intermittent system performance. We also conduct a breakdown analysis on the *blocking time* and *communication overhead* to explore opportunities for further performance improvement.

B. Forward Progress

First, we measured forward progress (defined as the number of finished tasks per minute) under different combinations of power sources (from stable to weak) and network sizes (N=3 or 4). In the smart farming application, larger forward progress means more intensive environmental monitoring. Different power sources allow us to observe how power instability decreases data availability and thus forward progress. Similarly, different network sizes allow different numbers of borrowable copies, allowing us to observe how data borrowing increases data availability and thus forward progress.

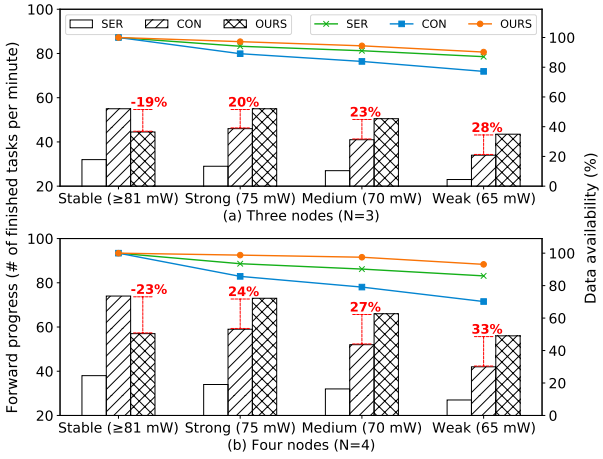

Fig. 10: Forward progress under various power sources and network sizes

Figure 10 shows the forward progress achieved by each evaluated protocol, under different power sources and network sizes. As expected, the forward progress decreases as the power source becomes weaker, because tasks require longer times to finish when nodes suffer from more frequent power failures. Although OURS and CON tremendously outperform SER, the decrease in forward progress is more apparent under OURS and CON than under SER, due to more tasks being

[2]The application developer may employ some existing approaches to assist in energy estimation and task partitioning [16, 38].

blocked by data unavailability when executed concurrently, as observed in Section III. However, OURS enables data borrowing to compensate for data unavailability, thus achieving more forward progress than CON. Specifically, under stable power, OURS achieves less forward progress than CON, due to the extra overhead needed to borrow data copies regardless of the 100% data availability. In contrast, under intermittent power, OURS improves the progress of CON more significantly under weaker power, due to a larger difference in data availability.

By comparing the counterpart results in Figures 10(a) and 10(b), we observe that the data availability under CON decreases with power instability (i.e., the blue line) more dramatically when N=4 than N=3. This is because the larger the network size, the more the tasks that request data objects from the owner node, causing the owner node to exhaust its energy more quickly. Consequently, the owner node suffers from more frequent power failures, and longer duration especially under weaker power, resulting in a more severe decrease in data availability. In contrast, the data availability under OURS decreases with power instability (i.e., the orange line) more slowly when N=4 than N=3. The reason is that the larger the network size, the more the borrowable copies, which is particularly beneficial to data availability under weaker power. In a larger network, due to a larger difference in data availability, OURS can achieve an even more significant progress improvement over CON.

Overall, OURS shows the largest forward progress among the three evaluated protocols under intermittent power. Compared with CON, OURS improves the forward progress respectively by 20 to 28% and by 24 to 33% for the 3-node and 4-node intermittent networks. The improvement is more evident under weaker power, where the owner node suffers from frequent power failures of longer duration, and in a larger network with more borrowable copies.

C. Blocking Time and Communication Overhead

On the surface, data borrowing increases data availability to reduce the blocking time at the cost of extra communications required to borrow data copies and check task dependency. In fact, if data borrowing is not allowed, one node has to keep retrying to access the desired data objects until the owner nodes recover, while data borrowing can leverage inherent data copies to reduce ineffective retries. To validate whether OURS can reduce the blocking time and communication overhead required by CON, we conduct a breakdown analysis on the blocking time (defined as the average time required for a task to successfully obtain a desired remote object) and the communication overhead (defined as the average number of communication retries required for a task that shares remote objects to finish its execution).

Figure 11 shows the blocking time caused by each evaluated protocol, under different power sources and network sizes. SER shows the longest blocking time among the protocols, because only one task can access a data object at a time. The blocking time is even longer when tasks have to execute for a longer time under weaker power, or have to wait for more tasks that are accessing shared objects in a larger network. By leveraging task concurrency, both CON and OURS cause a

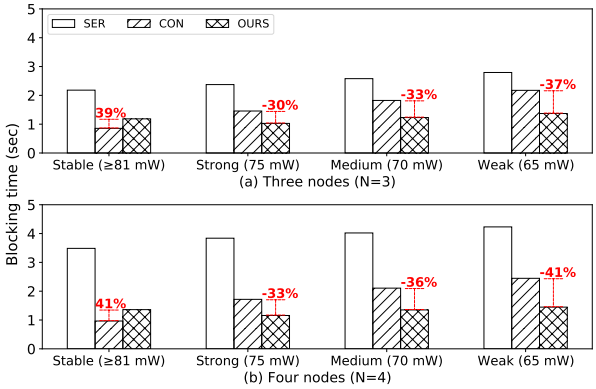

Fig. 11: Blocking time under various power sources and network sizes

substantially shorter blocking time than SER. Under stable power, OURS shows a longer blocking time than CON, because OURS still spends extra time and energy to borrow data copies given 100% data availability. However, OURS can significantly reduce the blocking time of CON under intermittent power, where OURS leverages the borrowable copies to proceed with forward progress. The reduction in the blocking time is more obvious under a weaker power source and in a larger network, due to a larger difference in data availability between OURS and CON, as shown in Figure 10.

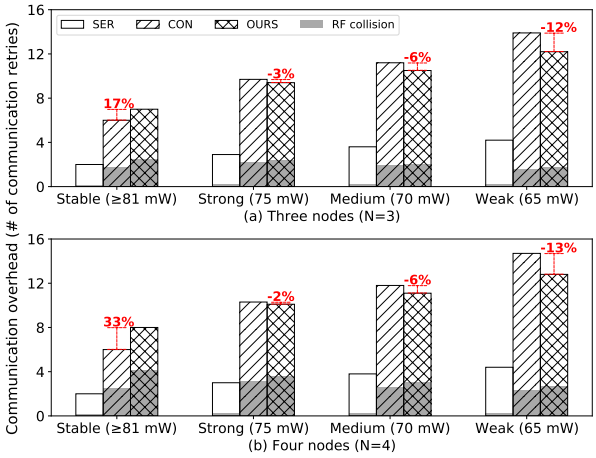

Fig. 12: Communication overhead under various power sources and network sizes

Figure 12 shows the communication overhead caused by each protocol under different power sources and network sizes. SER shows the lowest communication overhead among the protocols, because it does not require extra communications for serializability validation. By contrast, both CON and OURS incur high communication overhead, especially under weak power where each data access or serializability validation may require multiple communication retries to complete. Under stable power, OURS shows higher communication overhead than CON, because OURS requires extra communications for data borrowing and dependency checking. Interestingly, OURS shows lower overhead than CON under intermittent power. The reason is that CON requires more retries to complete each data access under weaker power, but

OURS can leverage the inherent data copies. On average, CON requires extra 1.1 to 3.1 retries, while OURS requires only 0.3 to 1.2 retries, depending mainly on power instability. Furthermore, we observe that some communications suffer from collision (i.e., the gray background in the bar plots), which is more severe in a larger network because more nodes try to transmit simultaneously, as discussed in Section V-C. Note that the collision has a more adverse impact on OURS than on CON, because both channel collision and data unavailability contribute to the blocking time of CON, whereas channel collision dominates the blocking time of OURS, as shown in Figure 11. In other words, the difference in forward progress between OURS and CON shown in Figure 10 will be more significant if channel collision is alleviated.

Overall, OURS shows the lowest blocking time among the evaluated protocols, and also causes lower communication overhead than CON under intermittent power. Compared with CON, OURS reduces the blocking time by 30 to 41%, and the communication overhead by 2 to 13%. These reductions, which play an important role in the overall forward progress improvement shown in Figure 10, are more evident under a weaker power source or in a larger network.

VII. Discussion on Scalability

To accommodate network connection and/or system failures, traditional distributed systems typically rely on *data synchronization* [10, 41] or *recovery mechanisms* [7, 8, 22]. A distributed synchronization/recovery procedure, when applied to an intermittent network, may not be timely completed before other nodes encounter power failures, thus stagnating the network. By contrast, our protocol maintains data consistency among nodes at all times to eliminate distributed synchronization or recovery, making it more applicable and scalable for intermittent networks than traditional distributed protocols. However, the channel collision, although attributed to the RF transceiver available for our implementation in Section V-C, largely limits the scale of our prototype network developed for quantitative evaluation. Here, we provide a discussion on the scalability of our protocol and opportunities to further improve its performance.

Suppose that a low-power wireless module that supports multiple access and multi-hop relay routing is available, and the hardware module is sufficiently energy-efficient to operate with relatively weak power and quickly recover from power failures, making a larger-scale intermittent network possible. Accordingly, data borrowing may rely on relay nodes and thus induce more node communications, especially when relay nodes suffer from frequent power failures. To reduce the communication overhead, *data-centric routing* [20] could be used in our data borrowing mechanism to broadcast object requests to the network and then aggregate duplicated copies on intermediate nodes through different paths. Existing routing protocols [4, 54], which aim to increase the probability of successful data delivery in sensor networks, are orthogonal to our distributed concurrency control protocol developed to improve the computation progress in intermittent networks.

Importantly, the borrowing strategy used in Section IV-B2 can be sophisticated to adapt to large-scale networks. As observed in Section VI-C, there is a trade-off between extra communications required by data borrowing and ineffective communications incurred by data unavailability. Under a weaker power source, where the desired data object will most likely be unavailable, data borrowing would be more beneficial to reduce ineffective communications. In contrast, when the owner node is online, data borrowing simply induces unnecessary communications. Thus, whether to invoke data borrowing ought to depend on if the owner node is expected to be online soon. Moreover, in a small network, if no borrowable copy exists, data borrowing will be unhelpful. In contrast, the larger the network, the more borrowable copies will exist, yet the accessing node may receive multiple identical copies from different nodes. Actually, a single data copy is sufficient to increase data availability. Thus, the strategy should be adapted to borrow data copies from a subset of nodes (e.g., neighbors only) rather than the whole network. There is also some interplay between the power instability and the current network size (in terms of online nodes only). A modeling methodology would facilitate the analysis of such a compound impact on the communication-related trade-off, thereby determining which nodes to borrow and when. An ideal strategy is supposed to obtain the desired data object with the minimum blocking time and communication overhead.

VIII. Conclusion

This paper introduces data borrowing to adapt distributed concurrency control to intermittent networks. Inspired by an observation that the forward progress improved by task concurrency would be significantly offset by data unavailability, data borrowing leverages data copies inherently created in a network to increase data availability and thus improve forward progress. To accommodate extremely frequent node failures, we develop an intermittent-aware protocol to enable data borrowing while guaranteeing data consistency by enforcing serializability, atomicity, and durability.

Our protocol was integrated into a FreeRTOS-extended intermittent operating system [2] and deployed on several Texas Instruments devices to form an intermittent network for small-scale smart farming. By significantly increasing data availability, our protocol can improve the forward progress by 20 to 33%, compared to a distributed concurrency control protocol that integrates the failure-resilient design [11] and two-phase commit protocol [23]. Our protocol is found to be particularly suitable for intermittent networks under frequent power failures.

The source code of our intermittent operating system has been made openly available [1], facilitating the development of intermittent-aware IoT applications. There is still room to explore towards large-scale intermittent networks, as well as the impact of intermittency. Future work will seek to scale up our prototype network by integrating emerging hardware technologies for networking the future IoT [17, 18]. We will also extend our software protocol by considering a more scalable borrowing strategy, where a modeling methodology would be helpful and worthy of further investigation.

References

[1] An Intermittent Operating System for Intermittent Networks. https://github.com/EMCLab-Sinica/Intermittent-Distributed.

[2] An Intermittent Operating System. https://github.com/EMCLab-Sinica/Intermittent-OS.

[3] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. Efficient intermittent computing with differential checkpointing. In *Proc. of ACM LCTES*, pages 70–81, 2019.

[4] H. Alwan and A. Agarwal. A survey on fault tolerant routing techniques in wireless sensor networks. In *Proc. of IEEE SENSORCOMM*, pages 366–371, 2009.

[5] F. A. Aouda, K. Marquet, and G. Salagnac. Incremental checkpointing of program state to NVRAM for transiently-powered systems. In *Proc. of IEEE ReCoSoC*, pages 1–4, 2014.

[6] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE TCAD*, 35(12):1968–1980, 2016.

[7] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM CSUR*, 13(2):185–221, 1981.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac. Sytare: A lightweight kernel for NVRAM-based transiently-powered systems. *IEEE TC*, 68(9):1390–1403, 2019.

[10] E. A. Brewer. Towards robust distributed systems (abstract). In *Proc. of ACM PODC*, page 7, 2000.

[11] W. Chen, P. Hsiu, and T. Kuo. Enabling failure-resilient intermittently-powered systems without runtime checkpointing. In *Proc. of ACM/IEEE DAC*, pages 1–6, 2019.

[12] W.-M. Chen, Y.-T. Chen, P.-C. Hsiu, and T.-W. Kuo. Multiversion concurrency control on intermittent systems. In *Proc. of ACM/IEEE ICCAD*, pages 1–8, 2019.

[13] W. M. Chen, T. W. Kuo, and P. C. Hsiu. Enabling failure-resilient intermittent systems without runtime checkpointing. *IEEE TCAD*, 39(12):4399–4412, 2020.

[14] W.-M. Chen, T.-W. Kuo, and P.-C. Hsiu. Heterogeneity-aware multicore synchronization for intermittent systems. *ACM TECS*, 20(5s):61:1–22, 2021.

[15] J. Choi, H. Joe, Y. Kim, and C. Jung. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *Proc. of IEEE RTAS*, pages 331–344, 2019.

[16] A. Colin and B. Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proc. of ACM OOPSLA*, page 514–530, 2016.

[17] J. de Winkel, C. Delle Donne, K. S. Yildirim, P. Pawełczak, and J. Hester. Reliable timekeeping for intermittent computing. In *Proc. of ACM ASPLOS*, pages 53–67, 2020.

[18] C. D. Donne, K. S. Yildirim, A. Y. Majid, J. Hester, and P. Pawełczak. Backing out of backscatter for intermittent wireless networks. In *Proc. of ACM ENSsys*, pages 38–40, 2018.

[19] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[20] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin. Highly-resilient, energy-efficient multipath routing in wireless sensor networks. *ACM SIGMOBILE*, 5(4):11–25, 2001.

[21] K. Geissdoerfer and M. Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *Proc. of USENIX NSDI*, 2021.

[22] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries. A recovery algorithm for a distributed database system. In *Proc. of ACM PODS*, pages 8–15, 1983.

[23] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, 1978.

[24] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[25] J. Hester and J. Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proc. of ACM SenSys*, 2017.

[26] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan. Powering the internet of things. In *Proc. of ACM ISLPED*, pages 375–380, 2014.

[27] H. Jayakumar, A. Raha, and V. Raghunathan. Quickrecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Proc. of IEEE VLSID*, pages 330–335, 2014.

[28] 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. of ACM/IEEE ISLPED*, pages 1–6, 2018.

[29] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu. Everything leaves footprints: Hardware accelerated intermittent deep inference. *IEEE TCAD*, 39(11):3479–3491, 2020.

[30] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[31] B. W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265. Springer, 1980.

[32] 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. of IEEE/ACM DAC*, pages 1–6, 2015.

[33] Y.-C. Lin, T.-A. Hsieh, K.-H. Hung, C. Yu, H. Garudadri, Y. Tsao, and T.-W. Kuo. Speech recovery for real-world self-powered intermittent devices. In *Proc. of IEEE ICASSP*, pages 26–30, 2022.

[34] Y.-C. Lin, P.-C. Hsiu, and T.-W. Kuo. Autonomous I/O for intermittent IoT systems. In *Proc. of IEEE/ACM ISLPED*, pages 1–6, 2019.

[35] S. Liu, W. Zhang, M. Lv, Q. Chen, and N. Guan. Latics: A low-overhead adaptive task-based intermittent computing system. *IEEE TCAD*, 39(11):3711–3723, 2020.

[36] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proc. of ACM PLDI*, pages 575–585, 2015.

[37] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *Proc. of IEEE HPCA*, pages 526–537, 2015.

[38] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent execution without checkpoints. In *Proc. of ACM OOPSLA*, pages 1–30, 2017.

[39] K. Maeng and B. Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proc. of USENIX OSDI*, pages 129–144, 2018.

[40] K. Maeng and B. Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proc. of ACM PLDI*, pages 1101–1116, 2019.

[41] F. Myter, C. Scholliers, and W. De Meuter. A CAPable distributed programming model. In *Proc. of ACM SPLASH*, pages 88–98, 2018.

[42] K. Qiu, N. Jao, K. Zhou, Y. Liu, J. Sampson, M. T. Kandemir, and V. Narayanan. MaxTracker: Continuously tracking the maximum computation progress for energy harvesting ReRAM-based CNN accelerators. *ACM TECS*, 20(5s):78:1–23, 2021.

[43] K. Qiu, M. Zhao, Z. Jia, J. Hu, C. J. Xue, K. Ma, X. Li, Y. Liu, and V. Narayanan. Design insights of non-volatile processors and accelerators in energy harvesting systems. In *Proc. of ACM GLSVLSI*, pages 369–374, 2020.

[44] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on rfid-scale devices. In *Proc. of ACM ASPLOS*, pages 159–170, 2011.

[45] F. Samie, L. Bauer, and J. Henkel. IoT technologies for embedded computing: A survey. In *Proc. of IEEE/ACM CODES+ISSS*, pages 1–10, 2016.

[46] Texas Instruments. Low-Power Sub-1 GHz RF transceiver datasheet. https://www.ti.com/lit/ds/symlink/cc1101.pdf.

[47] A. Torrisi, D. Brunelli, and K. S. Yildirim. Zero power energy-aware communication for transiently-powered sensing systems. In *Proc. of ENSsys*, pages 43–49, 2020.

[48] J. S. M. Verhofstad. Recovery techniques for database systems. *ACM CSUR*, 10(2):167–195, 1978.

[49] Y. Wu, Z. Jia, F. Fang, and J. Hu. Cooperative communication between two transiently powered sensor nodes by reinforcement learning. *IEEE TCAD*, 41(1):76–90, 2022.

[50] M. Xie, C. Pan, M. Zhao, Y. Liu, C. J. Xue, and J. Hu. Avoiding data inconsistency in energy harvesting powered embedded systems. *ACM TODAES*, 23(3), 2018.

[51] 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. of ACM/IEEE DAC*, pages 1–6, 2015.

[52] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. InK: Reactive kernel for tiny batteryless sensors. In *Proc. of ACM SenSys*, pages 41–53, 2018.

[53] D. Zhang, J. W. Park, Y. Zhang, Y. Zhao, Y. Wang, Y. Li, T. Bhagwat, W.-F. Chou, X. Jia, B. Kippelen, C. Fuentes-Hernandez, T. Starner, and G. D. Abowd. OptoSense: Towards ubiquitous self-powered ambient light sensing surfaces. *In Prof. of ACM IMWUT*, 4(3):103:1–27, 2020.

[54] Z. Zhang. Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: overview and challenges. *IEEE CST*, 8(1):24–37, 2006.

[55] M. Zhao, C. Fu, Z. Li, Q. Li, M. Xie, Y. Liu, J. Hu, Z. Jia, and C. J. Xue. Stack-size sensitive on-chip memory backup for self-powered nonvolatile processors. *IEEE TCAD*, 36(11):1804–1816, 2017.

Wei-Che Tsai (Student Member, IEEE) received the B.S. and M.S. degrees in computer science and information engineering from National Chiao Tung University and National Taiwan University, Taiwan, in 2017 and 2020, respectively. He was a Research Assistant with the Research Center for Information Technology Innovation, Academia Sinica, Taiwan, in 2021. His research interests include embedded systems and intermittent computing.

Wei-Ming Chen (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and information engineering from National Taiwan University, Taiwan, in 2013, 2015, and 2020, respectively. He is currently a Postdoctoral Associate with Microsystems Technology Laboratories at Massachusetts Institute of Technology, USA. Prior to that, he was a Postdoctoral Scholar with the Research Center for Information Technology Innovation, Academia Sinica, Taiwan, in 2021. His research interests include embedded systems, real-time systems, and tiny machine learning.

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

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

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

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