iFKVS: Lightweight Key–Value Store for Flash-Based Intermittently Computing Devices

Yen-Hsun Chen, Ting-En Liao, and Li-Pin Chang, *Senior Member, IEEE*

***Abstract*—Energy harvesting enables long-running sensing applications on tiny Internet of Things (IoT) devices without a battery installed. To overcome the intermittency of ambient energy sources, system software creates intermittent computation using checkpoints. While the scope of intermittent computation is quickly expanding, there is a strong demand for data storage and local data processing in such IoT devices. When considering data storage options, flash memory is more compelling than other types of nonvolatile memory due to its affordability and availability. We introduce iFKVS, a flash-based key–value store for multisensor IoT devices. In this study, we aim at supporting efficient key–value operations while guaranteeing the correctness of program execution across power interruptions. For indexing of multidimensional sensor data, we propose a quadtree-based structure for the minimization of extra writes from splitting and rebalancing; for checkpointing in flash storage, we propose a rollback-based algorithm that exploits the capabilities of byte-level writing and one-way bit flipping of flash memory. Experimental results based on a real energy-driven testbed demonstrate that with the same index structure design, our rollback-based approach obtains a significant reduction of 45% and 84% in the total execution time compared with checkpointing using write-ahead logging (WAL) and copying on write (COW), respectively.**

***Index Terms*—Checkpoint, flash storage, intermittent computation, key–value store.**

I. Introduction

SMART Internet of Things (IoT) applications, such as building automation, transportation, healthcare, and surveillance, involve a large number of distributed sensing devices to monitor the environment and take action when necessary. Typically, such tiny IoT devices are distributed over a large-scale wireless network and deployed in remote locations for long-lasting operation without human maintenance. To meet the deploy-and-forget requirement, a promising direction for the development of IoT devices is toward batteryless design. Instead of draining energy from batteries, these devices operate with ambient energy, which can be harvested from solar energy [1], radio signal energy, kinetic energy, and thermal energy [2].

Energy-harvesting devices use capacitors as an energy buffer. Because ambient energy is highly unstable, devices risk losing all computation progress when the stored energy is depleted. *Checkpoints* are therefore introduced to manage the loss across power interruptions: at a proper timing, a checkpoint is committed to save the program context to nonvolatile memory, and on power recovery, the latest checkpoint is restored to resume program execution. In particular, continuous checkpoints are periodically committed [3], while just-in-time (JIT) checkpoints are committed right before the device halts due to insufficient energy [4]. In contrast, atomic tasks commit changes to the global memory on their completion [5]. With the proposed checkpoints and atomic tasks, program execution is thus intermittent, i.e., applications can be long-running across unexpected power interruptions.

IoT devices constantly sample readings from multiple microsensors. Recent studies have shown that, instead of uploading sensor data to the cloud for processing, IoT devices can benefit from local data storage and processing: sensor data can be digested and compressed locally to reduce the cost of wireless data transmission [6], queries can be handled locally in sensors for improved responsiveness [7], and in healthcare applications, processing data locally in sensor devices enables anonymization and access control of personally identifiable data [8]. A multisensor IoT device acquires multidimensional data, where each data record typically comprises a timestamp and multiple sensor readings. Although a file system can easily store and query time-series data, it cannot, however, handle event-based queries efficiently. Such queries often involve different conditions on multiple dimensions [9], e.g., "List all events in the last week where the air temperature is above 40 °C and the relative humidity is above 60%." In contrast, these queries are better handled by a key–value store through multidimensional data indexing.

The inclusion of a key–value store in energy-harvesting devices cannot succeed without considering the memory cost. This is because, compared to working memory, data storage demands a much larger space. We consider flash memory because it offers a superior capacity per unit cost and is readily available on many platforms. To the best of our knowledge, this study is the first on the support of intermittency for flash-based key–value store. Now, in addition to fast key–value operations, new design challenges arise regarding efficient checkpoint operations: first, rewriting in flash is not possible without erasure. With this constraint, to restore a checkpoint,

Manuscript received 10 August 2024; accepted 10 August 2024. This work was supported in part by the National Science and Technology Council, Taiwan, under Grant MOST 110-2221-E-A49-029-MY3 and Grant NSTC 113-2221-E-A49-188-MY3. This article was presented at the International Conference on Embedded Software (EMSOFT) 2024 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Li-Pin Chang.)*

The authors are with the Department of Computer Science, College of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 30010, Taiwan (e-mail: eric246871@gmail.com; teliao0116@gmail.com; lpchang@cs.nycu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2024.3443698

every change that occurs after the most recent checkpoint must be explicitly undone from flash memory. Second, in prior work, checkpointing involves only the program context but not the storage space. Restoring the program state and undoing flash changes separately without global coordination creates inconsistency between the program and storage.

This study presents iFKVS, a lightweight key–value store for energy-harvesting devices. In addition to functional correctness, iFKVS aims for a reduction in write cost because, compared to reading, writing flash memory is more expensive in terms of time and energy. iFKVS uses two techniques to write flash: 1) byte logging and 2) one-way bit flipping (see Section II-B). For indexing of multidimensional sensor data, iFKVS employs a flash-efficient design of quadtrees [10], for which each tree node is managed as a tiny log space. This way, while efficient insertions are possible through fine-grained, out-of-place logging, existing data of a node remain intact for subsequent checkpoint restoration.

Checkpointing for iFKVS is based on rollback, a backup-before-modify approach. iFKVS maintains a global undo log to collect backups of tree nodes and program contexts in chronological order. Thanks to our log-structured node design, when modifying a tree node, rather than making a full backup of the node, iFKVS simply marks the flash address of the most recent write to the node. On checkpoint restoration, all data written after this mark will be undone from flash memory. Now, when recovering from a power interruption, iFKVS examines the global undo log, undoes node changes, and restores the program context. The process of undoing node changes is also optimized for flash memory: rather than erasing changes made after the previous checkpoint, iFKVS neutralizes these changes by zeroing them out using one-way bit flipping. On the other hand, to commit a checkpoint, iFKVS simply makes a backup of the program context and clears up the global undo log. In summary, this work makes the following contribution:

1) proposing a flash-efficient key–value index design for multidimensional sensor data;
2) introducing a flash-efficient checkpoint algorithm to enable intermittent computation on a flash-based key–value store;
3) presenting a mechanism for global coordination between the program state and flash state across checkpoints.

We successfully implemented our iFKVS design on Texas Instruments MSP430F5529 SoC. For performance evaluation and comparison, we also implemented the proposed tree structure on top of two additional checkpoint techniques, write-ahead logging (WAL) and copying on write (COW). Our results show that iFKVS achieved an overhead reduction by up to 84% in terms of the total execution time required to complete a workload under realistic power fail events.

II. Background and Motivation

A. Intermittent Computation

Batteryless IoT devices harvest ambient energy, store it in a capacitor, and operate with this stored energy until it is depleted. However, when running out of the stored

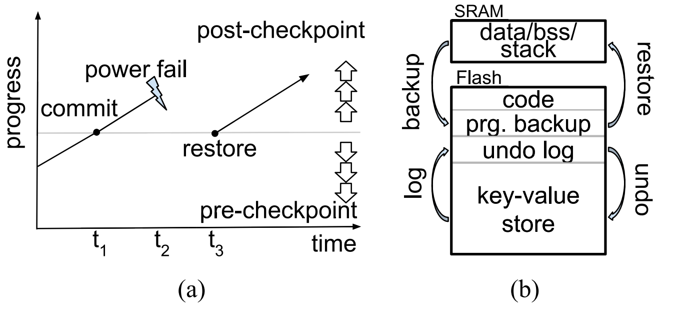

Fig. 1. Concept and design of intermittent computation. (a) Program progresses with checkpoints. (b) Memory organization of SRAM (working memory) and flash (storage).

energy, the devices lose all volatile program context, including contents in the CPU registers and volatile memory. To avoid re-execution from scratch, energy-harvesting devices timely commit a checkpoint to preserve the program context. A piece of nonvolatile memory is thus adopted to store the program state for later restoration. Fig. 1(a) shows program execution with checkpoints: The program commits a checkpoint at time t_1 and continues to execute. Later at time t_2, the device ceases to operate due to a power interruption. At time t_3, the capacitor is sufficiently charged and the device restarts. By restoring the prior checkpoint, the execution progress is reverted to that at t_2, i.e., only the progress between t_1 and t_2 is lost. Here, we refer to everything that happens before the checkpoint to be *pre-checkpoint*, and all the others to be *post-checkpoint*. For example, restoring the checkpoint effectively discards the post-checkpoint progress between t_1 and t_2.

In this study, we consider a typical memory organization, which is readily available on many embedded platforms, as shown in Fig. 1(b). The upper half is a piece of volatile SRAM that serves as the working memory. The SRAM holds read–write memory sections, including global variables and task stacks. The lower half is a piece of large, nonvolatile flash memory, commonly referred to as NOR flash. Because the flash is byte-addressable and capable of execute-in-place (XIP), it serves as a unified memory space for code storage (through XIP) and sensor data storage (through key–value store). A small portion of the flash memory is reserved for the program context backup and the flash undo log.

With continuous checkpoints [3], programs continue to execute after committing a checkpoint. On power recovery, post-checkpoint writes must be undone from flash memory. In contrast, with JIT checkpoints [4], programs suspend right after committing a checkpoint. However, energy estimation is not always correct [11], so JIT checkpoints are not free from post-checkpoint progress and a flash undo algorithm is always necessary.

B. Flash Memory Characteristics

Performance: In prior studies, both byte-addressable flash memory (specifically, NOR flash) and ferroelectric RAM (FRAM) are often considered in the design of energy-harvesting devices. Table I shows a comparison between the two. Both flash and FRAM can store executable code because they are byte-addressable and pin-compatible with

TABLE I
COMPARISON OF FLASH AND FRAM

	(NOR) Flash		FRAM	
Read write size	Byte, word		Byte, word	
eXecute-In-Place	Yes		Yes	
Erase size	512B segment		—	
Price (256KB)	USD 0.38 (WinBond)		USD 7.4 (Fujitsu)	
	Latency	Energy	Latency	Energy
Read	600 ns/byte	6 nJ/byte	250 ns/byte	1.4 nJ/byte
Write	18 μs/byte	209 nJ/byte	250 ns/byte	1.7 nJ/byte
Erase	50 μs/byte	420 nJ/byte	—	—

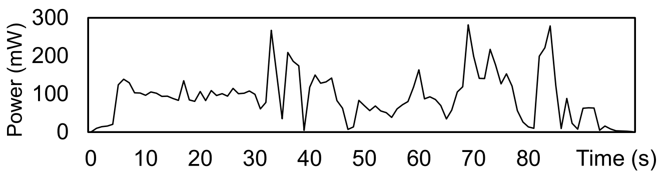

Fig. 2. Outdoor solar power collected by a 5.4-cm^2 solar panel.

the processor. Flash supports byte-level writes and is thus highly friendly for managing key–value index structure. While FRAM is erase-free, flash involves erasure of segments to reclaim writable memory space. We measured the latency and energy performance of flash memory and FRAM using the EnergyTrace feature from Texas Instruments MSP430F5529 and MSP430FR5994, respectively. The lower half of Table I shows that the read latencies of flash and FRAM are within the same order of magnitude, although flash is slower than FRAM. In contrast, flash writes are slower and consume more energy and require subsequent erasure.

Cost: Table I shows that flash holds a significant advantage in terms of storage design, as it costs only about 5% of the price of FRAM for the same 256-kB size. As of mid-2024, a 10-mF capacitor costs U.S. \$3, and a 128-kB flash-based MSP430 SoC costs U.S. \$9, making FRAM less attractive for storage-demanding sensing applications.

Applicability: The use of flash memory is subject to the scenario and specification of the target application [12]. Consider an outdoor sensing application requiring at least 512 kB of data storage. The first issue is the ambient power. Fig. 2 depicts that outdoor solar power from a small panel often exceeds 100 mW.[1] In our experiments, a flash-based MSP430 SoC operates at a duty cycle of 82% under 3.3 V×7 mA≈23 mW with a 10-mF capacitor, so this ambient power is more than sufficient to drive the flash-based SoC. The second issue is on capacity. As of 2024, within the lineup of MSP430-based SoCs, the maximum embedded FRAM size is 256 kB, whereas embedded flash can reach 512 kB. External memory is considered for size expansion [13], [14]. While the largest external (serial) FRAM from major distributors is 16 Mb, the largest external NOR flash reaches 1 Gb. Nevertheless, when the data writing rate is extremely high [15] or when the ambient power is weak, flash is unlikely a viable option.

Operations: Index management heavily involves fine-grained writes, e.g., adding a pointer or changing a flag bit. In flash, a bit of 1 can be changed to 0, but once a bit is set to 0, it cannot be changed. Bits in flash can only be reset to 1 through segment erasure. Byte writing in flash is feasible if the target address is unwritten. The system software can individually set a bit to 0 using a bitmask. For example, overwriting 0x55AA with 0xFF77 results in 0x5522. This operation, called *one-way bit flipping*, will be used in our approach to neutralize post-checkpoint data and to change flag bits.

[1]The power traces are used for evaluation in Section V-F.

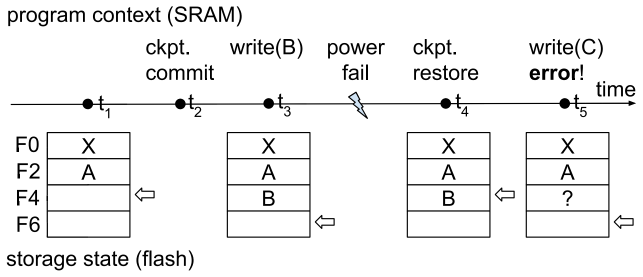

Fig. 3. Timeline of intermittent execution and instantiation of inconsistency between program and storage.

C. Motivational Example

A storage-enabled energy-harvesting device operates on not only the program context but also the storage state. However, to support intermittent computation, existing checkpoint algorithms are concerned with the program context only, and little effort has been made toward how to checkpoint on both. Without proper coordination between the two, after restoring a checkpoint on power recovery, an energy-harvesting device risks state inconsistencies between program and storage.

Fig. 3 demonstrates an inconsistency resulted from program-only checkpointing. The lower half depicts the flash state, in which a segment, a unit for flash erasure, has four words at addresses from 0F0h to 0F6Ch. The program context, depicted in the upper half, is in SRAM. Let the program maintains a write pointer that refers to the next available flash address for writing. Now, at time t_1, words X and A have been written to flash and the write pointer refers to 0F4h. The program commits a checkpoint at time t_2 and then at time t_3, it samples word B from a sensor, writes B to 0F4h, and advances the write pointer to 0F6h. After time t_3, the device undergoes a power outage. At time t_4, the device restores the prior checkpoint and reverts the program context to the state at time t_2. Notice that the write pointer, part of the program context, is returned to 0F4h, and this address has been occupied by word B. At time t_5, the program samples a new word C from the sensor and attempts to write C to the already-written address 0F4h. As flash memory prohibits in-place updating without prior erasure, the write results in erroneous contents at address 0F4h.

Power events can also cause storage metadata inconsistencies, as noted in [16]. Such errors can be avoided by extending the semantics of checkpoint to the storage. Checkpointing in flash memory is challenging because unlike the erase-free nonvolatile memory such as FRAM, flash memory cannot be rewritten without prior erasure. Restoring a checkpoint requires to undo all post-checkpoint writes explicitly form flash memory. For example, in Fig. 3, word B at address 0F4h must be erased from flash memory. However, flash memory erases in terms of segments, and in this example, the checkpoint algorithm must first create a backup of words X and A, erase the segment (0F0h to 0F7), and then copy words X and A back. In this study, our design goal is to maintain synchronization between program and storage for checkpoint operations, while also exploring innovative methods to efficiently neutralize post-checkpoint writes from flash memory.

D. Related Work

With energy harvesting, long-term computation employs checkpoints to survive power interruptions with managed progress loss. There are different types of checkpoint methods.

1) Continuous checkpoints, illustrated by Choi et al. [3] and Maeng and Lucia [17], are committed by the system software in a regular time interval and restored on power recovery.
2) Compiler-directed checkpoints, including the technique proposed by Liu et al. [18], rely on compile-time analysis of write-after-read (WaR) dependencies among variables. Checkpoints are committed at the boundaries of idempotent program blocks.
3) Atomic tasks, pioneered by Maeng et al. [5], commit local changes to the global memory upon their completion.
4) JIT checkpoints, demonstrated by Maeng and Lucia [4], involve additional hardware to monitor the voltage of the capacitor. A checkpoint is committed only when the voltage level is critically low. Our key–value store is designed to be independent of the checkpointing method.

While energy is a sacred resource in energy-harvesting devices, the introduction of advanced energy buffers [11], [19] has transformed them from dumb, wimpy devices into capable platforms. For example, Fraternali et al. [20] demonstrated wireless communication with Bluetooth low energy (BLE) on batteryless devices. Gobieski et al. [21] showed the feasibility of handwriting recognition and keyword spotting based on compact CNN models powered by energy harvesting. Montanari et al. [22] further optimized the tradeoff between energy usage and inference accuracy using multiresolution and multiexit CNN models. Mendis et al. [23] proposed intermittency-aware architecture search for neural networks to boost the chance of successful interference. Smart applications need extensive computation and data access, making local storage and retrieval of sensor data increasingly critical.

The use of flash memory and other nonvolatile memory options enables the local storage of sensor data. Dai et al. [24] demonstrated a flash-based, log-structured file system for sensor devices. Exploiting the append-only property of sensing applications, the file system preallocates a dedicated flash space for each file to grow. Tsiftes et al. [25] presented Coffee to enhance this approach using micro-logs to optimize small updates. Mazumder and Hallstrom [15] proposed LoggerFS to explore hybrid memory organization, i.e., FRAM serves as a high-speed write buffer and NAND flash is the final storage of file data. Compared with file systems, key–value stores better handle event- or value-based queries. Lin et al. [6] presented flash-efficient implementation of a hash table and a grid file for sensors. Fevgas and Bozanis [26] proposed buffering random writes for efficient insertion into flash-based grid files. Chang and Hsu [27] introduced soft lists, a flash-native implementation of skip lists with efficient flash garbage collection. However, in these studies, checkpointing is either not discussed at all or considered too expensive.

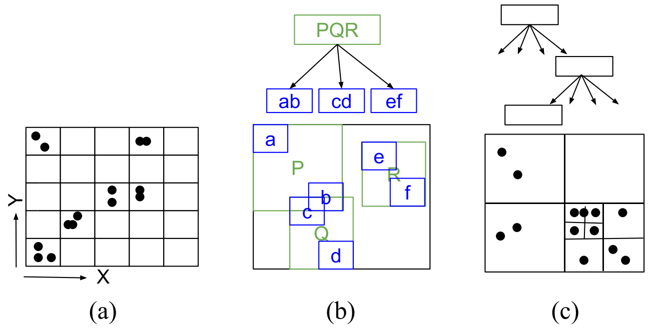

Fig. 4. Index structures for multidimensional data: (a) grid file, (b) R-tree, and (c) quadtree.

iNVMFS, proposed by Wu et al. [16], is a lightweight, checkpoint-enabled file system. However, iNVMFS is heavily based on the in-place updating capability of erase-free FRAM, which makes it incompatible with flash memory. Our work deviates from iNVMFS by addressing the unique constraints of flash operations. In addition, our approach handles queries on multidimensional data while iNVMFS does not.

III. Indexing and Checkpointing

This section introduces multidimensional data indexing and storage checkpointing, laying the foundation for our approach. Experienced readers can proceed to Section IV.

A. Indexing of Multisensor Data

For long-term sensor data storage, a file system conveniently creates day directories for hour files, in which sensor data are logged. This permits simple retrieval of data within a specified time interval. An IoT device collects multidimensional data from multiple sensors. However, queries on multisensor data involve distinct conditions on different dimensions, and to process such queries, a file system has no choice but inspects every record in all corresponding hour files.

There have been excellent index designs for multidimensional data, and for the ease of illustration, we consider a dimension size of two. Fig. 4(a) shows a grid file [6], in which data items are distributed among fixed-sized cells of the grid on the XY plane. As cells occupy the same size, address calculation and data transfer of cell data are highly efficient. A drawback of grid files is the poor space utilization under a highly uneven data distribution, and in addition, cell overflows may trigger expensive rehashing of the grid. Fig. 4(b) depicts an R-tree [28], which manages the bounding boxes in a tree-based hierarchical manner. Like B-trees, R-trees split and merge for height balancing and space compacting. However, these self-balancing operations introduce a large amount of writes. In addition, an R-tree may

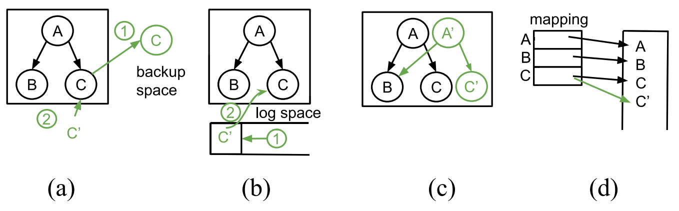

Fig. 5. Various techniques for checkpointing in storage. (a) Rolling back. b) WAL. (c) COW. (d) Log-structuring.

search multiple paths on query because it permits a partial overlap between boxes.

Fig. 4(c) shows a quadtree [10], in which a node represents a region. When a node is full, the region it represents is partitioned into four quadrants, each of which is associated with a new child node. In this study, our key–value store is designed based on quadtrees because a few of their properties well match our purpose: unlike grid files, in which all cell spaces have been preallocated, quadtrees creates new nodes only when necessary. Unlike R-trees, which permit overlapping between bounding boxes and involve extra writes for self-balancing, quadtrees efficiently search in disjoint quadrants and do not use extra writes for rebalancing.

B. Checkpointing in Storage

Checkpointing in working memory is typically based on variables, whereas storage checkpointing involves larger objects (such as tree nodes or data blocks) to save metadata space. Existing techniques, as discussed below, share a principle that avoids to modify pre-checkpoint data objects.

Fig. 5(a) illustrates the operation of rolling-back, a backup-before-modify approach. Suppose that we modify node *C* in the tree structure. In the first step, the pre-checkpoint node *C* is copied to a backup space and is ready for in-place updating. In the second step, the original node *C* is updated in place and becomes node C'. Committing a checkpoint involves discarding the backup node, whereas restoring the previous checkpoint necessitates replacing node C' with node C, effectively reverting to the previous state. In contrast, Fig. 5(b) shows WAL and how it handles the same update. In the first step, the up-to-date node C' is added to a log space (also known as the redo log) without altering the pre-checkpoint node C. Committing a checkpoint involves overwriting node *C* with node C' (step 2) followed by clearing the log. Restoring the prior checkpoint can be done by discarding all post-checkpoint data, i.e., clearing the log. Both rolling back and WAL involve in-place node updating in their second steps, which seems prohibited in flash. However, this operation can be implemented by node logging, as will be shown in later sections.

Fig. 5(c) depicts COW. The update to node *C* is handled in an out-of-place manner to avoid modifying pre-checkpoint data. After this, the parent node *A* must also be updated to refer to node C', and the update is again handled in an out-of-place manner. The copying of nodes, operated by the wandering tree algorithm, propagates upstream until reaching the root or a post-checkpoint node. *A* checkpoint is committed

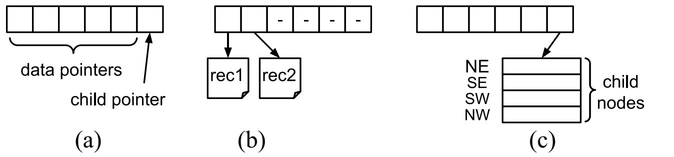

Fig. 6. Node structure. Each node is a tiny log space. (a) Node structure. (b) Leaf node. (c) Internal node.

by discarding nodes *A* and C, which are unreachable from the new root A'. The prior checkpoint is restored by discarding all post-checkpoint nodes, i.e., nodes A' and C'. Fig. 5(d) shows log-structuring, which performs out-of-place writing always. *A* signature of log-structuring is the use of a mapping table in the working memory, which eliminates the necessity for path copying, as seen in COW. Here, updating to node *C* is accomplished by appending a new node C' and then updating the mapping table accordingly. The mapping table must be backed up and restored on checkpoint operations.

We implemented rolling back, WAL, and COW in our experimental study. Notice that log structuring is not considered because the tiny SRAM of the IoT platform that we use cannot afford the space overhead of the node mapping table.

IV. Lightweight Key–Value Store With Efficient Checkpoint Support

A. Node and Tree Structure

To achieve efficient query processing with a reduced write frequency, we propose a flash-efficient implementation of quadtrees. *A* quadtree is composed by nodes, and in our design, nodes contain pointers only, as shown in Fig. 6(a). *A* node is an array of pointer slots, and because flash memory permits byte writing, a pointer slot can be either available, in-use, or obsolete (neutralized). An in-use pointer can be neutralized by writing zeros through one-way bit flipping, as will be discussed later. Our design treats a node as a tiny logging space of pointers. This design greatly aids checkpoint operations because new pointers are inserted in the order of time, so it will be straightforward to distinguish pre-checkpoint pointers from post-checkpoint ones.

Quadtrees always insert new data to leaf nodes. Fig. 6(b) shows that two new records are inserted to a leaf node, and the first two slots are allocated to pointers referring to the data records. The pointer slots of a leaf node are sequentially allocated to refer to new records, and when all pointer slots have been used, a node-split procedure is taken. Let the dimension size be two for the purpose of illustration. As Fig. 6(c) shows, after the split, the original node becomes the parent of four new leaf nodes (NE to NW). Here, we take a few measures to simplify the node structure for saving flash writes. First, to save pointers, we propose allocating the four child nodes in a contiguous flash space so that the parent node uses only one pointer to refer to all the child nodes. Second, on node split, the node space is partitioned into equal subspaces without referring to a pivot, and this saves another pointer. Note that if a high data skewness is anticipated, a pivot pointer can be added for pivot-based splitting. Third,

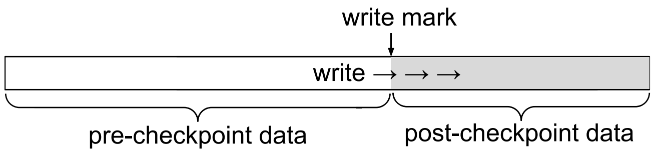

Fig. 7. Our log-and-mark approach. The write mark indicates the current write address of the log at checkpoint committing.

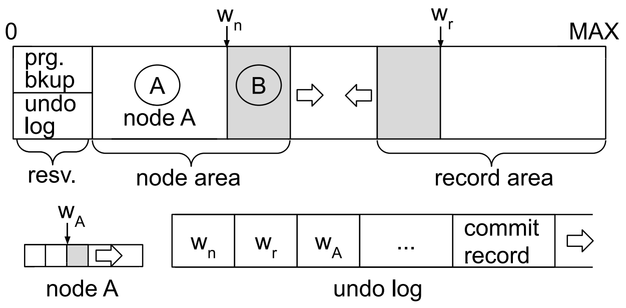

Fig. 8. Data layout in flash (upper half) and contents in node *A* and the undo log (lower half). Gray portions are post-checkpoint data.

we introduce *lazy split*, which retains all data pointers of a node during splitting to avoid unnecessary flash writes. In contrast, traditional quadtrees require migrating data items from a parent node to its child nodes during a split.

B. Global Undo Log

Because key–value operations involve writing to flash, we detail the design of our global undo log and explain how it operates. Fig. 7 depicts the core idea of our design, the log-and-mark approach: flash writing is always handled through logging. Checkpoint committing marks the current write address of the log, and all subsequent writes that occur after this mark produce post-checkpoint data. Writing to flash is subject to two simple rules: 1) pre-checkpoint data cannot be modified and 2) post-checkpoint data must be undone from flash on checkpoint restoration.

Fig. 8 shows an example of the flash-space layout. A reserved area appears at the beginning of the flash, and this area contains the backup space for the program state in addition to the global undo log. In our design, the main flash space is divided into two areas, one for tree nodes and the other for data records. The two areas are both log spaces. Node allocation starts at the lowest address, and record allocation begins at the highest address. The two disjoint areas grow toward each other. By separating nodes from records, pointers can greatly reduce the number of bits needed to refer to objects, as they only need to encode object offsets instead of full addresses.

Writing to the node area and the record area follows the log-and-mark approach. The lower half of Fig. 8 shows the global undo log, in which all write marks are collected. In particular, the node area is associated with write mark w_n, which indicates the current write address of the node area when the most recent checkpoint is committed. In other words, everything that appears before w_n is considered pre-checkpoint, node *A* for instance, while anything written after

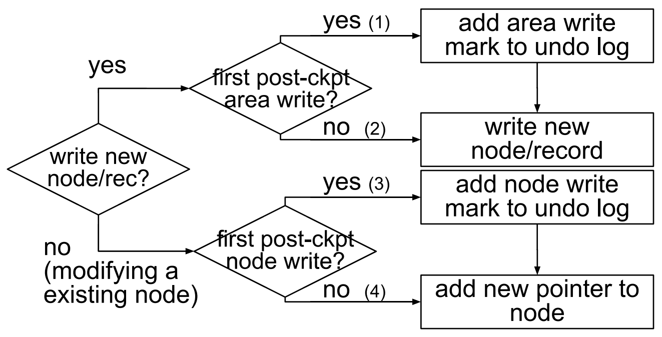
write new node/rec?
yes
first post-ckpt area write?
yes (1)
add area write mark to undo log
no (2)
write new node/record
no (modifying a existing node)
first post-ckpt node write?
yes (3)
add node write mark to undo log
no (4)
add new pointer to node

Fig. 9. Write decision flows.

w_n is classified as post-checkpoint, as represented by node B. Accordingly, the record area is associated with a write mark w_r as the boundary between pre-checkpoint records and post-checkpoint ones. Notice that because tree nodes themselves are tiny logging spaces, a node may contain both pre-checkpoint data and post-checkpoint ones. For example, although node *A* is in the pre-checkpoint node area, it accepts new pointers after the latest checkpoint. To reflect this, a write mark w_A is created for node A, marking that everything appears after w_A a piece of post-checkpoint data.

Fig. 9 details the write control flow. For example, the fourth decision flow shows the procedure to add a new pointer to node *A* in Fig. 8. Basically, when writing to an area, our design examines whether this is the first write to the area after the most recent checkpoint. *A* write mark is created for the area *only if* it is the first write to the area, and the write mark is added to the global undo log. No write mark will be created on subsequent writes. Writing to an existing tree node follows the same logic. Consequently, a node has a write mark in the global undo log only if it has been modified since the most recent checkpoint. Our design uses a tiny hash table in the working memory to efficiently check for the presence of a write mark in the undo log. As will be detailed later, during checkpoint restoration, the write marks aid the identification of post-checkpoint data, e.g., the gray portions in Fig. 8.

C. Key–Value Operations

A query on a key–value store is either a *get* or a *scan*. *A* get operation looks for an exact match; a scan operation, or a range query, returns all data records that satisfy a set of conditions across different dimensions. Scans are more useful to event-based queries. Our tree index follows the original quadtree algorithm for query processing with the following exceptions: because our nodes are log spaces, records associated with a node are unsorted, requiring examination of all records during query processing. In addition, with our lazy split, internal nodes are not empty, so their contents must also be inspected for matches.

Write-oriented operations include *put*, *update*, and *delete*. *A* put operation inserts a new record. To handle a put request, our tree index first writes a new data record and then modifies the tree index. The index modification includes locating the leaf node to insert, conducting node split if necessary, and adding a new record pointer to the target node. Here, to support

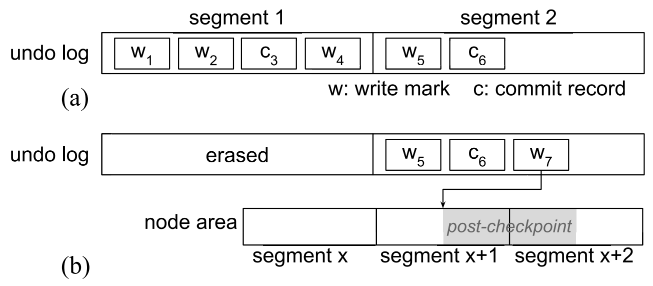

Fig. 10. Global undo log (a) right after checkpoint commit and (b) after multiple post-checkpoint writes to the node area.

checkpoint operations, writing to flash must comply with the principle of our log-and-mark approach. As Fig. 9 shows, the first and the second decision flows show how to write a new node or a new record, while adding a new pointer to an existing node goes through the third or the fourth decision flow. As will be explained in later sections, our checkpoint method is based on rollback. The original rolling-back method requires to create a backup of a node on the first post-checkpoint modification to the node [see Fig. 5(a)]. Interestingly, because our node structure is a log, post-checkpoint writes do not affect pre-checkpoint data in nodes. In other words, the node backup step of rolling back is implemented simply by creating a node write mark and adding it to the global undo log.

In this study, random deletion and update of existing data records are not considered, as they are less useful to sensing applications [6], [15], [24]. Instead, our flash cleaning policy removes expired data from flash, as will be shown later.

D. Checkpoint Operations

Commit: Checkpoints are committed periodically or on low-power events. To commit a checkpoint, a backup of the program context, including the CPU registers, data/bss section, and stack section, are written to a reserved flash space (see Fig. 8). The commit procedure is then finished by writing a commit record to the global undo log. *A* commit record contains a pointer referring to the program backup, a commit signature, and a checksum of the record. Now, because our method for checkpointing in flash is based on rollback, when a commit record has been written, all write marks in the global undo log are obsolete (since nothing needs to be rolled back). In our design, the global undo log is a circular buffer composed by a set of flash segments. Fig. 10(a) shows a global undo log of two segments, in which a new checkpoint has been committed by writing commit record c_6. In this example, all write marks before the final commit record c_6 can be discarded. Here, segment 1 can be erased but not segment 2, because commit record c_6 must remain valid.

Restore: A checkpoint is restored when the device recovers from a power interruption. The restoration procedure first scans the global undo log for the last commit record. *A* write mark appearing after the last commit record is a pointer to the start of post-checkpoint data. These data, located in the node area, the record area, or in a tree node, must be undone from flash. For example, in Fig. 10(b), write mark w_7 is added to the global undo log to indicate that new nodes have been written to the node area after the latest checkpoint. To restore the flash state, the undo procedure erases all the post-checkpoint nodes from the node area. Because segment $x + 1$ contains both pre-checkpoint nodes and post-checkpoint ones, it undergoes *erase-based undo*: The undo procedure copies the pre-checkpoint nodes to a backup space, erases the segment, and copies the nodes back. Segment $x + 2$ is erased directly because it contains post-checkpoint nodes only. This procedure then restores the program context to resume execution.

The erase-based undo procedure resets the flash space occupied by post-checkpoint data, ensuring that the flash state is precisely reverted to its state at the latest checkpoint. This involves an extra overhead of data copying. To avoid this overhead, we propose relaxing the definition of global consistency to ensure that the program never mistakenly writes to a flash space that has already been written. To achieve this, we introduce *discard undo* to neutralize post-checkpoint data. Recall that flash is capable of one-way bit flipping, i.e., individual bits can be zeroed out and this operation is idempotent. Now, on checkpoint restoration, for each write mark found in the global undo log, our approach scans flash space after the mark and writes zero until an erased byte (whose value is 0xff) is encountered. The first free addresses of the node area and the record area are updated accordingly. This procedure applies to tree nodes as well, and in subsequent key–value operations, zero (neutralized) pointers will be ignored.

The undo log plays an essential role to the checkpoint correctness. We safeguard write marks and commit records using checksums. During a reboot scan, if the last log object is a valid commit record, then both the program and storage have been checkpointed; otherwise, both will be restored. For the former case, recall that a commit record has a pointer to the program backup, so the device simply restores the backup and resumes execution. For the latter case, only the last log object might fail the integrity check due to interrupted writing. *A* corrupted commit record is discarded, and a broken write mark is also discarded as its corresponding post-checkpoint data are not yet written. After this, valid write marks and the prior commit record in the log [e.g., w_7 and c_6 in Fig. 10(b)] are used for flash undo and program restoration, respectively. For flash undo, while the discard undo (our main proposal) is idempotent, the erase-based undo requires a reserved flash space with a checksum-protected header to make the copy-erase-copyback procedure fail-safe.

E. Flash Space Cleaning

As flash prohibits in-place updating, new data are written to free space. Over time, the amount of free space becomes insufficient and garbage collection must be involved. Conventional copy-based garbage collection strategically selects a segment for erasure, and before erasing the segment, all valid data must be migrated to another free space. In spite of the extra overhead of data movement, this procedure requires an extra layer of indirection because data movement silently changes the flash addresses of data objects.

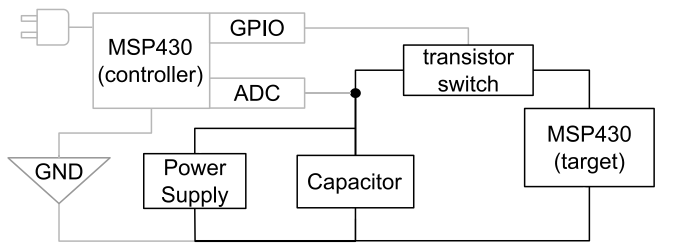

Fig. 11. Our testbed for energy-driven experiments. The target board runs our key–value store.

Random deletion and update of sensor data are not useful in sensing applications [6], [15], [24]. In contrast, queries are more concerned with recent events [9], so old data can be expired and deleted from local storage. We propose a copy-free, partition-based flash cleaning method. The entire flash memory is divided into a few equal-sized partitions, each of which is managed by an independent instance of our tree index. With this design, query processing involves all tree instances. Partitions are used in a first-in–first-out manner for writing: new data are written to the current partition, and when it is full, the oldest partition is erased. Notice that because the oldest partition contains a pre-checkpoint tree instance, the partition is marked but cannot be erased until a new checkpoint has been committed. This partition-based method not only eliminates the need for data copying but also ensures wear leveling in flash memory.

Both the performance of insertion and query scale with the quadtree algorithm. The overhead of commit is bounded by the log size due to the recycling of obsolete log segments, and the overhead of restore depends on the amount of post-checkpoint data for undo, i.e., the checkpoint interval length.

V. Experimental Results

A. Experimental Setup

1) Parameters and Metrics: We implemented our approach, called iFKVS, based on a Texas Instruments LaunchPad. The platform involves an MSP430F5229 SoC, which is equipped with a 16-bit MSP430 embedded processor. The processor is rated at 8 MHz and is integrated with 8 kB of SRAM and 128 kB of flash. The flash segment size is 512 bytes. Our iFKVS implementation uses less than 200 bytes of SRAM for the volatile program context, including the read–write sections for data, bss, and stack. Furthermore, its executable code in flash is less than 7 kB. In addition to the executable binary, the embedded flash is shared by the global undo log and our key–value store. In particular, the latter part uses 80 kB, which is partitioned into four equal-sized partitions. The node size and the record size are 64 and 8 bytes, respectively.

We conducted experiments on an energy-driven testbed, as depicted in Fig. 11. The target board runs our key–value store, and its power source involves a power supply and a 10-mF capacitor. The power supply, a Keysight E36312A, delivers 7 mA at 3.3 V to the system. In this configuration, the target board draws more power than the power supply can provide, resulting in the target board remaining nonfunctional until the capacitor has charged sufficiently. A separate controller board (powered by an adaptor) monitors the voltage of the capacitor through its ADC. The capacitor discharges when the target board is in operation. When the capacitor voltage drops below 2.3 V, the controller turns off the transistor switch to detach the capacitor from the target board for charging. This also causes a blackout to the target board. If the capacitor voltage raises to 3.3 V during charging, the controller turns on the transistor switch to attach the capacitor to the target board, and the target board is powered on.

TABLE II
Checkpoint Designs of NVRAM-Based File Systems

	Metadata	Data blocks
iNVMFS [16]	WAL	COW with in-place writing
BPFS [30]	COW	COW with atomic writes
PMFS [31]	RB	COW on large updates

The experimental dataset is constructed based on the database of real weather stations [29]. Each data record in our dataset consists of a timestamp, a relative humidity, and a temperature level. According to the database, we set the relative humidity between 0% and 100% at a resolution of 0.1% and the temperature level between −20 °F and 100 °F at a resolution of 0.1 °F. We use fixed-point numbers to represent the humidity and temperature values. Based on the value intervals, we utilize a uniform distribution to generate twenty thousand records for one dataset and a normal distribution for the same number of records in the other dataset. The quadtree does not index sequential timestamps to avoid skewness. For multidimensional queries, the timestamp of matched records is inspected separately. During experiments, a dataset is entirely written to the key–value store, and a checkpoint is committed every 100 insertions. Checkpoints are restored when the target recovers from asynchronous power events.

Our primary performance metric is the total execution time, which includes not only the overhead of inserting the entire dataset into the key–value store but also the latency contributed by periodic checkpoint committing and restoring a checkpoint in the event of power recovery. The shorter the total execution time is, the lower the overheads of key–value operations and checkpoint operations are. We also report the latency of each type of operation for analysis.

2) Methods Under Evaluation: While we are not aware of directly comparable prior studies, we select the NVRAM-based file systems in Table II as evaluation baselines, with necessary modifications for flash compatibility.

iNVMFS [16] employs WAL on file system metadata and COW on file data blocks. As previously shown in Fig. 5(b), post-checkpoint writes are appended to a redo log and later written back to their destination addresses. The writing-back procedure is idempotent on flash, allowing it to be safely repeated if interrupted. We revise iNVMFS by 1) disabling in-place overwriting of post-checkpoint data and 2) dropping the COW feature as sensor data records are never modified.

BPFS [30] is based on COW, as shown in Fig. 5(c). BPFS exploits NVRAM atomic writes to avoid path copying on small updates. As atomic writing is not available in flash,

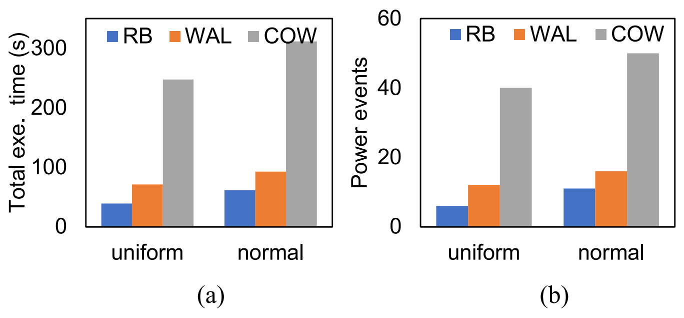

Fig. 12. Overall performance under different data distributions. (a) Total execution times. (b) Power event counts.

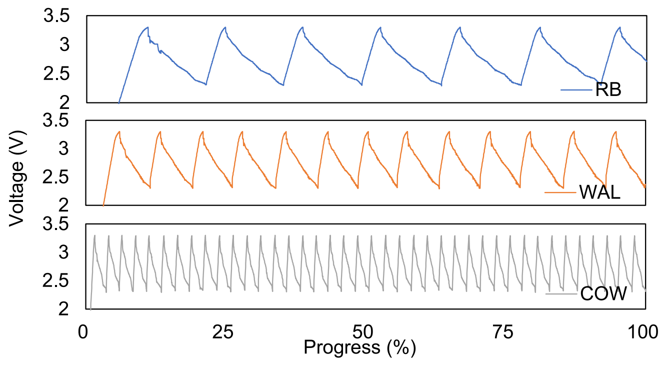

Fig. 13. Capacitor voltage with respect to workload progress with the uniform dataset. Notice that the *x*-axis denotes the *progress* rather than the wall-clock time.

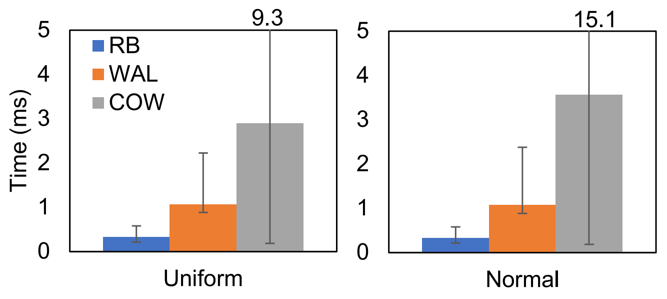

Fig. 14. Time overhead of insertion operations.

we revise BPFS by 1) logging small updates in a post-checkpoint node and 2) copying a pre-checkpoint node on its first update. Path copying consumes free space at a high rate, leaving many obsolete nodes in flash memory. After filling up the current flash partition, the revised BPFS initiates copy-based compaction if the partition's space utilization is lower than a threshold. The revised BPFS achieve a level of space utilization comparable with other methods.

PMFS [31] uses rollback on metadata and COW on data blocks. PMFS performs rollback through in-place overwriting because NVRAM is erase-free, but it degrades into erase-based undo for flash compatibility. Instead of modifying PMFS, we compare iFKVS with erase-based undo against iFKVS with discard undo (our main proposal) in Section V-E.

We refer to our iFKVS as RB for its rollback-based design. Similarly, we denote the revised iNVMFS as WAL, and the revised BPFS as COW. All these methods share the quadtree structure and the partition-based space cleaning policy.

B. Total Execution Time

Fig. 12(a) shows the total execution time of RB (equivalently, our iFKVS), WAL, and COW. The total execution time involves the time to insert all the 20 000 data records and the time to commit a checkpoint every 100 insertion operations. In addition, the time also includes the overhead to restore a checkpoint on power recovery. Results show that RB completes the workload much earlier than WAL and COW. Specifically, under the uniform data distribution, RB finishes inserting all the records in 39 s, achieving significant reductions of 45% and 84% compared with WAL and COW, respectively. Results also show that all the methods have longer execution times under the normal data distribution. This is because with the normal distribution, keys in each dimension (humidity or temperature) are more concentrated. Popular tree paths grow deeper than others, leading to key–value operations along these paths taking more time to complete.

Among the factors that contribute to the total execution time, the insertion overhead is the primary factor, followed by the overhead of checkpoint commit. This is because insertion and commit are the two most frequent operations during the workload. Now, RB has the lightest overhead to insert a record, because it creates a write mark in the global undo log only on the first post-checkpoint modification to a node, to the node area, or to the record area. Subsequent post-checkpoint writes proceed without writing extra information to the undo log. To commit a checkpoint, RB simply discards the contents in the global undo log. In contrast, WAL must add a log record to the redo log on every write. In addition, when committing a checkpoint, for every log record, WAL must copy the written data from the log record to the destination flash address. As for COW, when an insertion operation goes to a pre-checkpoint node, COW has to copy the node and all pre-checkpoint nodes along the path toward the root node. In addition, when the current partition is full, COW compacts the partition (in an out-of-place manner) and commits a checkpoint. In other words, the highest runtime overhead of COW results from the wandering tree algorithm and the partition compaction procedure.

Another factor that contributes to the total execution time is checkpoint restoration. Fig. 12(b) shows the power event counts. While RB experiences 6 times of power interruption, WAL and COW encounter 12 and 40 times of power outage, respectively, under the uniform data distribution. Furthermore, Fig. 13 depicts the voltage of the capacitor throughout the progress of the entire workload. RB pushes the progress much faster than WAL and COW, i.e., it undergoes much fewer charging cycles throughout the workload. In contrast, WAL and COW progress slowly and the capacitor is recharged more often. The frequent checkpoint restoration operations upon device restarts further increase their total execution times.

C. Key–Value Operation Overhead

In this section, we report the average, maximum, and minimal latencies of key–value operations, including insertion and query. Fig. 14 shows the insertion latency. The trend in the insertion latencies appears highly consistent with the

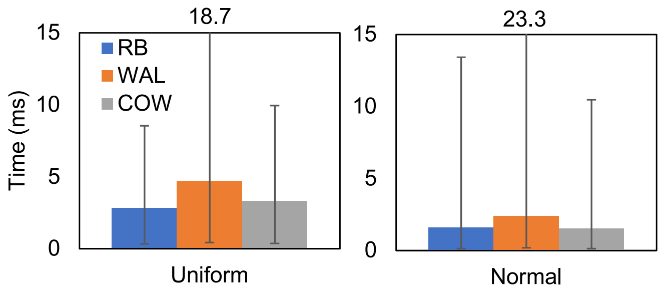

Fig. 15. Time overhead of range query operations.

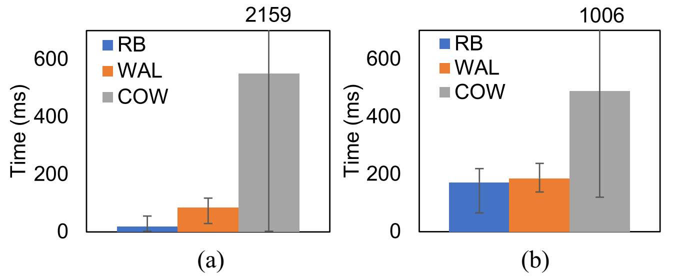

Fig. 16. Time overhead (a) to commit a checkpoint and (b) to restore a checkpoint. Data distribution is uniform.

trend in the total execution times, as insertion is the most frequent operations in the workload. Insertion with a normal distribution is slower than with a uniform distribution because popular tree paths are relatively deeper and insertion on these paths are slower. RB is the fastest on insertion thanks to the log-and-mark design: it adds a write mark to the global undo log to distinguish pre-checkpoint data and post-checkpoint data. WAL takes about three times as long as RB to insert a record on average (1.1 ms versus 0.3 ms). This is because WAL amplifies the write cost by including a target flash address in each log record in addition to the written data. Furthermore, when searching the tree index during insertion, WAL examines the redo log for updates to a node. Searching for updates takes extra time, even though we have implemented a tiny hash table in the working memory to aid the search. COW is the slowest on insertion, about nine times slower than RB on average (2.9 ms versus 0.3 ms). Notably, the maximum latency of COW is extremely high. This is attributed to the high cost of copying a full path for the wandering tree algorithm.

We evaluate the query performance upon the completion of the workload because at this time, the key–value store has been fully stressed and contains sufficiently many records for query. Specifically, RB, WAL, and COW have about 4500 valid records in flash before our query test. We present results of range queries (scan) rather than point queries (get) because get is hardly useful to sensing applications. In addition, get is a subset of scan. In this test, we specify a random range on each dimension and retrieve the first 256 records from the key–value store. Fig. 15 shows that RB and COW are comparable in terms of the query latency. In contrast, WAL performs noticeably worse than the other two, because every time when reading a tree node, WAL must examine the redo log for any recent updates to the node. The queries are faster under a normal distribution as they cover unpopular (shorter) paths.

D. Checkpoint Operation Overhead

Fig. 16(a) shows the time overhead of checkpoint commit for RB, WAL, and COW. Notably, RB commits a checkpoint much faster than WAL and COW. The overhead to commit a checkpoint with RB includes 1) making a backup of the program context; 2) writing a commit record to the global undo log; and 3) erasing unused segments of the global undo log (see Fig. 10). For the first item, based on DMA, the process of backing up the 200 bytes of SRAM program context to flash memory accounts for less than 10% of the total commit latency. For the third item, RB merely erases one segment after three times of commit on average. This is because RB slowly consumes the global undo log space by writing tiny write marks to the log *only* on the first post-checkpoint write to an area or node (see Fig. 9).

The commit overhead of WAL is much higher than that of RB. On every write, WAL adds a log record consisting of a target flash address and the written data, and therefore it must erase segments from its redo log more often. In addition, WAL must also write back all the pending log records in its redo log to complete checkpoint commit. Here, COW shows an extremely high overhead. COW basically switches the root to commit a checkpoint. However, due to its quick consumption of flash space through path copying, if the current partition is full and its space utilization for valid records is low, COW compacts the partition to free up space. We measured that COW improves the valid record count by about 1.7 times through compaction, making it comparable to RB and WAL.

Fig. 16(b) shows the time overhead to restore a checkpoint. While commit is a periodic event, restore is conducted only upon power recovery. To restore a checkpoint, RB loads the backup of program context and undoes post-checkpoint flash writes from flash. As detailed in Section IV-D, instead of employing the expensive erase-based undo, RB uses discard undo, which efficiently neutralizes all post-checkpoint data through one-way bit flipping. In contrast, for WAL, all post-checkpoint data is contained within its redo log, and therefore on checkpoint restoration, WAL erases the redo log to discard all post-the checkpoint writes. COW suffers from the highest overhead to restore a checkpoint. This is because COW consumes flash space at a high rate and, on checkpoint restoration, a large amount of post-checkpoint data must be erased from flash.

Fig. 17 shows the CDF of iFKVS (RB) operation latencies. Insert and commit show a stable latency, with exceptions of slow insertion due to node splitting and slow commit for erasing log segments. The restore latency is proportional to the amount of post-checkpoint data undone, and the scan latency depends on the tree heights of the scanned records.

Based on the technique in [32], we test the checkpoint correctness by replaying the workload on iFKVS with and without power events. Besides random blackouts, we inject asynchronous power fails to checkpoint commit, checkpoint restore, and record insertion. When done replaying the workload, we retrieve all records through the quadtree and hash

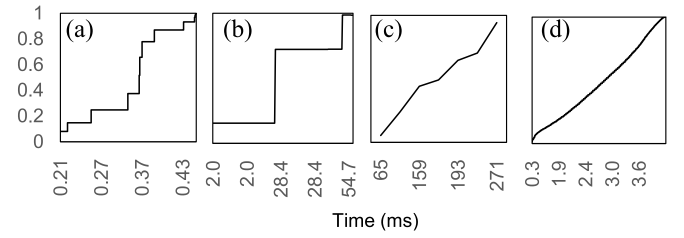

Fig. 17. CDF of iFKVS (RB) op. latencies under uniform. (a) Insert. (b) Commit. (c) Restore. (d) Scan.

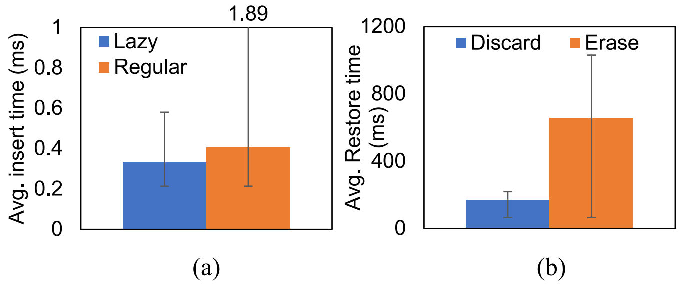

Fig. 18. Evaluating iFKVS (RB) with (a) lazy or regular node split and (b) discard undo or erase undo. Data distribution is uniform.

their contents. The quadtree and the undo log are not hashed as they are affected by power events. The correctness of our checkpoint operation is verified by the consistent hash values observed both with and without power events.

E. Differential Evaluation

We evaluate our iFKVS design by turning on and off individual features. Fig. 18(a) shows the insertion latency with and without our lazy split method. With lazy split, when a node splits, the node retains all its data (pointers actually) without migrating them to its new child nodes. Lazy split reduces the average insertion latency by about 20% and significantly lowers the maximum time length. Fig. 18(b) depicts the time overheads to restore a checkpoint with the proposed discard undo method and the erase-based undo method. While discard undo neutralizes post-checkpoint data through bit flipping, erase undo must undergo a copy-erase-copyback procedure to erase post-checkpoint data from flash. Our results indicate that restoring a checkpoint with discard undo requires only 26% of the time compared with using erase undo.

F. Irregular Power Traces

We also evaluate our iFKVS using irregular power traces. To collect power traces, we place a 5.4-cm^2 solar panel alongside a road, where passing pedestrians can obstruct the sunlight. We record the power output of the panel, and the power traces are stored in the power supply for replay. A fragment of the traces can be found in Fig. 2. Due to the ample solar power, we scale down the amplitude of the traces by 0.1× to emulate the power from a smaller solar panel. Fig. 19(a) shows the total execution times of RB and WAL, and RB completes the workload much earlier than WAL. COW is not included here because it suffers from stagnation during checkpoint recovery and does not complete the workload. Fig. 19(b) depicts the

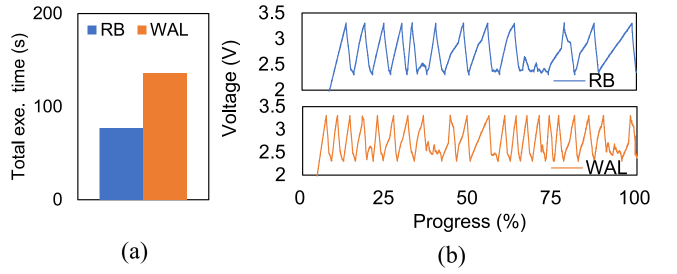

Fig. 19. (a) Total execution times and (b) capacitor voltage under irregular power traces. Data distribution is uniform.

TABLE III
ENERGY ANALYSIS SUMMARY

	7mA	6mA	5mA	Flash-op only	iFKVS(FRAM)
Time (s)	38.9	51	97	19.9	11.7
Charge cycles	6	11	23	3	1

capacitor voltage, showing 1) unpredictable fluctuation in the voltage level due to an unstable power input and 2) more charging cycles for WAL throughout the workload due to its high operational overheads.

G. Flash Lifetime Analysis

Two major factors determine the flash lifespan: 1) the data sampling (writing) rate and 2) the flash size. In Section V-B, iFKVS completes 20 000 record insertions in about 39 s and performs 914 segment erases. This reflects a sampling rate of (20 000/39 s)≈513 Hz. As ambient conditions like temperature and humidity do not change abruptly, we assume a practical sampling rate of 10 Hz. Therefore, 20 000 record insertions require (20 000/10) × 1 s = 2000 s. iFKVS rotates partitions to ensure even erasure (see Section IV-E). Consider a 256-kB flash memory, which has 512 segments. At a 10-Hz sampling rate, evenly erasing all the 512 segments once requires 2000 s × (512/914) ≈ 1120 s. According to [33], flash endures 10^5 erases, so it takes 1120 s × 10^5 ≈ 3.6 years to retire the flash memory. This analysis does not involve the undo log; however, its wear can be managed by adjusting the log length.

H. Energy Analysis

We evaluate iFKVS using three power levels, 7 mA (the default), 6 mA, and 5 mA, at 3.3 V. Table III reports the total execution times and charge cycle counts for each power level, and the observed duty cycles (active periods) for these levels are 82%, 66%, and 49%, respectively. The reduction in the power strength significantly amplifies the execution overhead, highlighting the loss of post-checkpoint progress, slow charging, and cost of checkpoint restoration.

Both the microprocessor and flash memory contribute to energy consumption. We monitor how many flash reads, writes, and erases are used for indexing and checkpointing, and we replay these amounts of flash operations without the iFKVS stack. The flash-only replay uses 19.9 s in 3 charge cycles, while the full iFKVS stack uses 38.9 s in 6 cycles. Therefore, storage-related flash operations contribute to about half of the total energy usage.

We also modify our flash-based iFKVS to run on an FRAM-based SoC MSP430FR5994. We observe that the FRAM-based iFKVS completes the workload in 11.7 s with 1 charge cycle. Here, FRAM demonstrates its power efficiency for both program execution and storage management. However, as discussed in Section II-B, flash is more cost-efficient, with its applicability depending on sustainable power consumption.

VI. Conclusion

In the foreseeable future, near-data processing will be crucial for energy-harvesting sensing applications. This work aims to close the gap between intermittent computation and flash-based key–value storage. We target on two issues. First, checkpointing must involve not only the program context but also the storage state to achieve global consistency. Second, writes after the most recent checkpoint must be undone from flash memory for fast power recovery. We propose a global undo log that guarantees the synchrony between the states of the program and the storage. For checkpointing in flash, we propose a novel log-and-mark approach, which treats everything as a log space, including the node area, the record area, and each individual node. With this, post-checkpoint data can be easily identified and undone from flash. Based on a unique property of flash memory, one-way bit flipping, we propose replacing the erase-based flash undo procedure with idempotent zero-filling writes. Our experiments show that, compared to a WAL-based approach and a COW-based method, our design achieves a significant reduction in the total execution time by 45% and 84%, respectively.

As part of a smart building project, our system prototype is deployed by meeting room windows to optimize room reservation and air-conditioning plans using event-based queries on illumination and temperature readings. Furthermore, because page-based NAND flash offers a higher storage density, we are thrusting toward an FRAM-NAND hybrid approach for a cheaper, larger key–value store.

References

[1] D. Brunelli, C. Moser, L. Thiele, and L. Benini, "Design of a solar-harvesting circuit for batteryless embedded systems," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 56, no. 11, pp. 2519–2528, Nov. 2009.

[2] M. Magno and D. Boyle, "Wearable energy harvesting: From body to battery," in *Proc. 12th Int. Conf. Design Technol. Integr. Syst. Nanoscale Era (DTIS)*, 2017, pp. 1–6.

[3] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving stagnation-free intermittent computation with boundary-free adaptive execution," in *Proc. IEEE Real-Time Embedd. Technol. Appl. Symp.*, 2019, pp. 331–344.

[4] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2019, pp. 1101–1116. [Online]. Available: https://doi.org/10.1145/3314221.3314613

[5] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proc. ACM Program. Lang.*, vol. 1, pp. 1–30, Oct. 2017.

[6] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, and W. A. Najjar, "Efficient indexing data structures for flash-based sensor devices," *ACM Trans. Storage*, vol. 2, no. 4, pp. 468–503, 2006. [Online]. Available: https://doi.org/10.1145/1210596.1210601

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[8] M. A. Sahi et al., "Privacy preservation in e-Healthcare environments: State of the art and future directions," *IEEE Access*, vol. 6, pp. 464–478, 2018.

[9] Y. Diao, D. Ganesan, G. Mathur, and P. J. Shenoy, "Rethinking data management for storage-centric sensor networks," in *Proc. CIDR*, 2007, pp. 22–31.

[10] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1–9, Mar. 1974.

[11] J. Choi, H. Joe, and C. Jung, "CapOS: Capacitor error resilience for energy harvesting systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4539–4550, Nov. 2022.

[12] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2011, pp. 159–170.

[13] K. Akhunov, E. Yildiz, and K. S. Yildirim, "Enabling efficient intermittent computing on brand new microcontrollers via tracking programmable voltage thresholds," in *Proc. 11th Int. Workshop Energy Harvest. Energy-Neutral Sens. Syst.*, 2023, pp. 16–22.

[14] M. Nardello, L. Caronti, and D. Brunelli, "Intermittent intelligent camera with LEO sensor-to-satellite connectivity," in *Proc. 11th Int. Workshop Energy Harvest. Energy-Neutral Sens. Syst.*, 2023, pp. 79–85.

[15] B. Mazumder and J. O. Hallstrom, "A fast, lightweight, and reliable file system for wireless sensor networks," in *Proc. 13th Int. Conf. Embedd. Softw.*, 2016, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2968478.2968486

[16] Y.-J. Wu, C.-Y. Kuo, and L.-P. Chang, "iNVMFS: An efficient file system for NVRAM-based intermittent computing devices," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3638–3649, Nov. 2022.

[17] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 129–144.

[18] S. Liu, W. Zhang, M. Lv, Q. Chen, and N. Guan, "LATICS: *A* low-overhead adaptive task-based intermittent computing system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3711–3723, Nov. 2020.

[19] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 767–781. [Online]. Available: https://doi.org/10.1145/3173162.3173210

[20] F. Fraternali, B. Balaji, Y. Agarwal, L. Benini, and R. Gupta, "Pible: Battery-free mote for perpetual indoor BLE applications," in *Proc. 5th Conf. Syst. Built Environ.*, 2018, pp. 168–171.

[21] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 199–213. [Online]. Available: https://doi.org/10.1145/3297858.3304011

[22] A. Montanari, M. Sharma, D. Jenkus, M. Alloulah, L. Qendro, and F. Kawsar, "ePerceptive: Energy reactive embedded intelligence for batteryless sensors," in *Proc. 18th Conf. Embedd. Netw. Sens. Syst.*, 2020, pp. 382–394. [Online]. Available: https://doi.org/10.1145/3384419.3430782

[23] H. R. Mendis, C.-K. Kang, and P.-C. Hsiu, "Intermittent-aware neural architecture search," *ACM Trans. Embedd. Comput. Syst.*, vol. 20, no. 5s, pp. 1–27, 2021.

[24] H. Dai, M. Neufeld, and R. Han, "ELF: An efficient log-structured flash file system for micro sensor nodes," in *Proc. 2nd Int. Conf. Embedd. Netw. Sens. Syst.*, 2004, pp. 176–187.

[25] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the coffee file system," in *Proc. Int. Conf. Inf. Process. Sens. Netw.*, 2009, pp. 349–360.

[26] A. Fevgas and P. Bozanis, "Grid-file: Towards to a flash efficient multi-dimensional index," in *Proc. 26th Int. Conf. Data Manage. Cloud, Grid P2P Syst.*, 2015, pp. 285–294.

[27] L.-P. Chang and C.-H. Hsu, "Soft lists: *A* native index structure for nor-flash-based embedded devices," in *Proc. Asia South Pac. Design Autom. Conf.*, 2009, pp. 799–804.

[28] A. Guttman, "R-trees: *A* dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1984, pp. 47–57.

[29] "Colorado agricultural meteorological network (CoAgMET) data set." Accessed: Jul. 1, 2023. [Online]. Available: https://coagmet.colostate.edu/

[30] J. Condit et al., "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 133–146.

[31] S. R. Dulloor et al., "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–15.

[32] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 17–32.

[33] "MSP430 flash memory characteristics," Application note, Texas Instrum., Dallas, TX, USA, 2018. [Online]. Available: https://www.ti.com/lit/an/slaa334b/slaa334b.pdf