Heterogeneity-aware Multicore Synchronization for Intermittent Systems

WEI-MING CHEN, Academia Sinica and National Taiwan University, Taiwan
TEI-WEI KUO, City University of Hong Kong, China and National Taiwan University, Taiwan
PI-CHENG HSIU, Academia Sinica, National Taiwan University and National Chi Nan University, Taiwan

Intermittent systems enable batteryless devices to operate through energy harvesting by leveraging the complementary characteristics of volatile (VM) and non-volatile memory (NVM). Unfortunately, alternate and frequent accesses to heterogeneous memories for accumulative execution across power cycles can significantly hinder computation progress. The progress impediment is mainly due to more CPU time being wasted for slow NVM accesses than for fast VM accesses.

This paper explores how to leverage heterogeneous cores to mitigate the progress impediment caused by heterogeneous memories. In particular, a *delegable* and *adaptive synchronization* protocol is proposed to allow memory accesses to be delegated between cores and to dynamically adapt to diverse memory access latency. Moreover, our design guarantees *task serializability* across multiple cores and maintains *data consistency* despite frequent power failures. We integrated our design into FreeRTOS running on a Cypress device featuring heterogeneous dual cores and hybrid memories. Experimental results show that, compared to recent approaches that assume single-core intermittent systems, our design can improve computation progress at least 1.8x and even up to 33.9x by leveraging core heterogeneity.

CCS Concepts: • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: Multicore synchronization, task concurrency, data consistency, batteryless devices, intermittent computing

ACM Reference Format:
Wei-Ming Chen, Tei-Wei Kuo, and Pi-Cheng Hsiu. 2021. Heterogeneity-aware Multicore Synchronization for Intermittent Systems. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 61 (September 2021), 22 pages. https://doi.org/10.1145/3476992

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign bland System Synthesis (CODES+ISSS), 2021.
This work was supported in part by the Ministry of Science and Technology, Taiwan, under grant MOST 110-2222-E-001-003-MY3.
Authors' addresses: W.-M. Chen, Research Center for Information Technology Innovation (CITI), Academia Sinica, and Department of Computer Science and Information Engineering, National Taiwan University, Taiwan, d04922006@csie.ntu.edu.tw; T.-W. Kuo, Department of Computer Science, City University of Hong Kong, China, and Department of Computer Science and Information Engineering, National Taiwan University, Taiwan, ktw@ntu.edu.tw; P.-C. Hsiu (corresponding author), Research Center for Information Technology Innovation (CITI), Academia Sinica, College of Electrical Engineering and Computer Science, National Taiwan University, and Department of Computer Science and Information Engineering, National Chi Nan University, Taiwan, pchsiu@citi.sinica.edu.tw.

1539-9087/2021/9-ART61 $15.00
https://doi.org/10.1145/3476992

1 INTRODUCTION

Various *heterogeneous* multicore architectures have recently been explored to fulfill increasingly diverse demands for applications in the Internet-of-Things (IoT) domain [22, 41]. For instance, many ultra low-power microcontrollers (e.g., STMicro STM32WB55 and Cypress PSoC62) have featured heterogeneous dual cores to allow IoT devices to combine computing performance and energy efficiency. However, the rapidly increasing number of IoT devices and their maintenance costs make them inapplicable to be powered with batteries [17]. A promising alternative is energy harvesting, but ambient energy is inherently unstable and weak, leading to frequent power failures and resumptions [30]. *Intermittent computing*, which leverages the complementary characteristics of heterogeneous memories, has emerged as a new computing paradigm and created innovative applications like solar-powered ambient sensors [48], self-powered cameras [35], wireless-powered cellphones [42], and battery-free game consoles [13]. However, applications executed on such devices need to run intermittently [15, 21] and accumulate computation progress across power-on cycles [4, 37]. Such execution behavior increases the difficulty of designing hardware chips [18, 28, 43] and system software [20, 31], and also makes existing peripheral designs intended for continuously-powered systems inapplicable [5, 26, 33, 34].

Intermittent systems typically use *checkpointing* to frequently save task progress and data in volatile memory (VM) to non-volatile memory (NVM) so that the systems can be recovered after power resumption [11, 19, 32, 46]. Various consistency-aware checkpointing approaches have been proposed to avoid *inconsistency* between the data preserved in NVM and the task progress restored into VM [36] due to re-execution of a *non-idempotent* task [29, 39, 44, 45]. Although many attempts have been made to improve checkpointing efficiency [14, 24, 47, 49], checkpointing-based approaches normally suffer from long system suspension and recovery times [8]. To alleviate runtime checkpointing overhead, *task-based* approaches partition an application program into multiple atomic tasks and only update data modifications after task completion [12, 27, 40]. Various programming models [29] and compiler supports [31] have been developed to assist task partitioning. Nonetheless, these studies assume *serial task execution* in order to be manageable for programmers.

Multitasking operating systems typically allow for *concurrent task execution* so that multiple tasks that share data can be executed in an interleaving manner to improve CPU utilization [23]. It is particularly challenging to enable concurrency control on intermittent systems due to frequent yet transient power failures. Recently, some *failure-resilient* designs [8–10] have been proposed to enable intermittent-aware task concurrency. Specifically, the designs guarantee the *serializability* of concurrently executed tasks as if they were serially executed in some arbitrary order. To allow instant system recovery from power failures, *durability* is achieved by keeping the data in NVM always consistent with the task progress restored into VM, and *atomicity* is enforced to update data in an all-or-none manner and thus avoid data corruption due to partial updates. However, these studies assume *single-core* systems and do not address *multicore task synchronization*, which is essential to ensure the consistency of shared data when accessed simultaneously by tasks executed on different cores.

This paper presents the first attempt to study intermittent-aware multicore synchronization. We observe that the overhead required to enable intermittent computing can easily reduce the achievable computation progress, mainly because intermittent systems require *alternate* and *frequent* accesses to heterogeneous memories to advance and preserve progress, resulting in more CPU time being wasted for slow NVM accesses than for fast VM accesses. Importantly, the wasted CPU time varies significantly with the memory access latency, and the reduced computation progress substantially increases with the CPU computing capability. These observations motivate us to

explore how to leverage heterogeneous cores to mitigate the computation progress reduction caused by heterogeneous memories.

To better utilize CPU resources, we propose a heterogeneity-aware design, which allows the high-speed core to primarily push computation progress forward and the low-speed core to help carry out slow memory accesses. Our design extends the *two-version data management* used in a failure-resilient design [9] to provide intermittent-aware multicore concurrency control. However, a major challenge lies in how to enable *delegable* and *adaptive synchronization*, while enforcing task serializability across multiple cores. Specifically, delegable commitment allows slow memory accesses to be delegated from the high-speed core to the low-speed core, thereby transferring the wasted CPU time between cores. Moreover, adaptive synchronization allows tasks to adaptively wait for fast and slow memory accesses, thus reducing the wasted CPU time. Lastly, the serializability of tasks concurrently executed on different cores is enforced via *two-phase backward validation*. When realized on intermittent systems, the design should be sufficiently lightweight to accommodate transient power on and off cycles.

We integrated our heterogeneity-aware design into FreeRTOS [3], a real-time operating system, and conducted extensive experiments on a heterogeneous dual-core device featuring hybrid memories, namely Cypress CY8CKIT-062-WiFi-BT, to measure achieved *forward progress*[1]. Compared to *failure-resilient non-checkpointing* [9] and *system-wise checkpointing* [19] designs, our design can respectively improve forward progress by 1.8 to 11.6x and 4.7 to 33.9x by leveraging the heterogeneity of CPU cores, where the improvement is more evident for intermittent devices with more asymmetric operating frequencies. We also conduct a breakdown analysis on the required *system costs* and incurred *runtime overheads* to provide some valuable insights into intermittent-aware multicore synchronization.

The remainder of this paper is organized as follows. Section 2 provides background information and Section 3 explains the motivation for this work. Section 4 presents our heterogeneity-aware design, with implementation issues discussed in Section 5. The experimental results are reported in Section 6. Section 7 contains some concluding remarks.

2 BACKGROUND

2.1 Intermittent-aware Concurrency Control

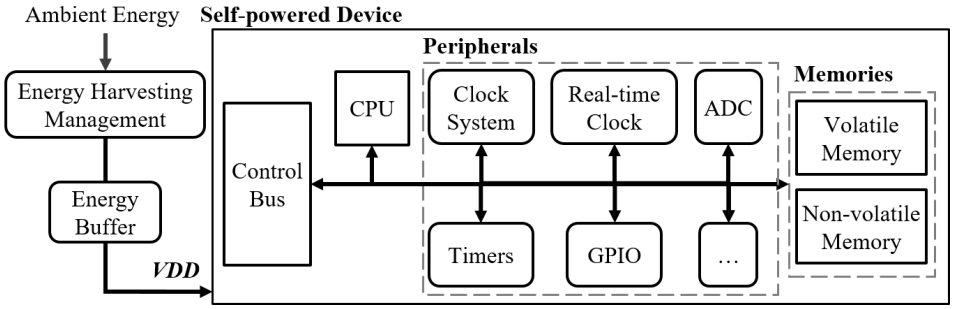


Fig. 1. System architecture of an intermittent device.

Figure 1 shows the system architecture of a typical intermittent device, using energy harvesting as the power source and a capacitor as an energy buffer. The system is respectively powered on

[1]Forward progress, defined as the computation workload finished per time unit, is a widely used performance metric for intermittent systems due to their non-deterministic power-on cycles.

and off when the capacitor voltage respectively reaches two preset thresholds. To accommodate frequent power failures, the device is normally equipped with hybrid memories composed of volatile memory (VM) and non-volatile memory (NVM), both of which are directly accessible by the CPU. The VM features high performance but will lose data after power failures, whereas the NVM can preserve data across power cycles but suffers from high latency for data accesses. Borrowed from traditional (database) systems, checkpointing is adopted to backup and restore data between VM and NVM to accumulate computation progress across short power-on periods [4, 19, 32, 37, 43], while *logging* is used to accommodate system recovery from power failures to a *consistent* state.

To improve computation progress, multitasking systems typically support *task concurrency*, where multiple tasks that share data can be executed in an alternating manner [23]. As a result, the resultant values of the data depend on the precedence order of data access operations conducted by tasks. To ensure that the resultant data values are predictable, *concurrency control* is normally adopted to enforce *serializability*, ensuring that the concurrent task execution is equivalent to the case where these tasks are executed *serially* in some arbitrary order. However, enabling task concurrency on checkpointing-based intermittent systems may result in long runtime suspension and recovery times, given frequent power failures (e.g., from hundreds of milliseconds to a few seconds [8, 9]). This is because checkpointing needs to suspend running tasks to obtain an exclusive access to VM and NVM, and recovery via log traversing needs to redo (resp. undo) data modifications made by finished (resp. unfinished) tasks after power resumption. When more tasks are concurrently executed, the overheads become more evident due to the increased size of checkpoints and logs, which can even offset the computation progress improved by concurrent task execution. Moreover, intermittent systems may be unable to conduct timely system recovery via log traversing within short power-on periods.

Recent work has proposed a *failure-resilient* design to enable task concurrency on intermittent systems without checkpointing and logging [9]. The design uses two-version data in heterogeneous memories to provide the flexibility of concurrent task execution by allowing every data object to be accessed simultaneously by multiple tasks. The data object in NVM is kept consistent with the progress of finished tasks at all times, ensuring that a *consistent* version is always available in NVM to allow instant recovery without logging. Thus, *durability* is achievable despite frequent power failures. On the other hand, data modifications made by each task on a data object will be conducted on the task's own *working* version in VM. To eliminate checkpointing, only when a task is prepared to finish, its working versions can be committed into the consistent versions if its execution is validated as *serializable* with a backward validation algorithm. Otherwise, the task is aborted and rerun as if it had never been executed; accordingly, *idempotence* is protected because repeated execution will produce the same result as one single execution. To avoid partial updates on the consistent versions due to power failures, *atomicity* is enforced by an atomic commit operation. However, the failure-resilient design assumes single-core intermittent systems and does not address multicore task synchronization.

2.2 Multicore Task Synchronization

When multiple tasks sharing data objects run on different cores, simultaneous updates on any data object can potentially lead to a non-serializable outcome. Thus, data access operations conducted by these tasks must be synchronized to ensure the consistency of shared data objects. Synchronization protocols typically employ *spinlocks* or *semaphores* to ensure that shared data objects can only be updated in a *critical section*, where only one task can enter at a time. Specifically, before entering the critical section, a task needs to acquire a corresponding lock which will be released after the task leaves the critical section. In a spinlock-based protocol, a task uses a `while` loop to keep the core repeatedly request the lock, also known as *busy-waiting*. In contrast, under a semaphore-based

protocol, a task which fails to acquire the lock will enter a *sleep-waiting* state, be put into a waiting queue, and yield the CPU core to other tasks.

Each kind of protocol has its own pros and cons with respect to the waiting time of a task. Specifically, spinlocks provide a quick response time to the task but will waste CPU time if the waiting time is lengthy, whereas a waiting task can release the CPU to other tasks under a semaphore-based protocol but have to spend additional CPU time for context switch and maintenance of the waiting queue. To reduce the wasted CPU time in traditional systems, early research has proposed *adaptive synchronization*, which allows a task to transit between waiting mechanisms at runtime according to the length of the waiting time [25]. During transition, previous approaches typically incur a non-negligible overhead to traverse hierarchical locking structures, suspend/resume waiting tasks, and migrate locks between memory hierarchies [6, 7]. The incurred overhead and waiting time are assessed prior to transition, so that a transition can be invoked only when the assessment finds it to be beneficial.

Unlike traditional systems, where data accessed by tasks are mainly in VM, intermittent systems require frequent and alternate accesses to VM and NVM, thus increasing the number of waiting tasks which suffer from high variations in the waiting time (due to different memory access latency). With that in mind, tasks have to frequently transit between waiting mechanisms to adapt to the variations. When the power source is weaker, where alternate accesses to VM and NVM will be more frequently, the overhead incurred by assessment-based approaches could offset the CPU time gained by adaptive synchronization and even might be unable to guarantee timely transitions.

3 HETEROGENEOUS MULTICORE INTERMITTENT SYSTEMS: OBSERVATION AND MOTIVATION

To explore the extent of computation progress reduced by heterogeneous memories to enable intermittency, as well as how heterogeneous cores can help improve progress, we conducted an experiment on a Cypress device, equipped with a CPU comprising a big core (Cortex-M4) and a LITTLE core (Cortex-M0+) and with hybrid memories comprising VM (256KB SRAM) and NVM (1MB Flash), under a stable power source. In the experiment, each core repeatedly runs two tasks, each of which reads a shared data object, performs an arithmetic function on the object, and finally updates the object.

We compare the computation progress achieved by a conventional design used by continuously-powered systems and a failure-resilient design [9] developed for intermittently-powered systems (Section 2.1). The main difference between them is that once a task is ready to finish, the failure-resilient design needs to update the data object in NVM to prevent the progress from being lost due to a potential power failure, whereas the conventional design only updates the data object in VM, which provides high performance when a power failure will never occur. Figure 2 shows the *forward progress* (defined as the number of finished tasks per second) respectively achieved by the two designs using each waiting mechanism (spinlock or semaphore) on each core type (big or LITTLE). As expected, the failure-resilient design significantly reduces the forward progress achieved by the conventional design by between 62% and 92%. The reduction is mainly because the conventional design only requires VM accesses, whereas the failure-resilient design incurs many NVM accesses, and much more CPU time is wasted for slow NVM accesses than for fast VM accesses, regardless of the waiting mechanism used.

This experiment allows us to make the following two observations. First, by comparing the progress achieved by the two designs, the reduction on the big core is about 4 times that on the LITTLE core. In other words, the cost of performing a memory access is much higher on the big core than on the LITTLE core. Note that although the amount of wasted CPU time remains similar regardless of which core the memory access is performed on, the big core can make more progress

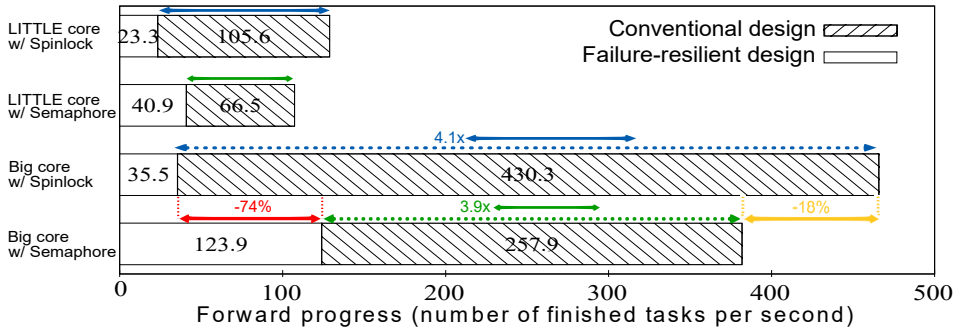


Fig. 2. Computation progress achieved by different designs using different waiting mechanisms on individual cores.

than the LITTLE core given the same amount of CPU time. This suggests that forward progress can be significantly improved if those slow operations leading to high memory access latency on the big core can be *delegated* to the LITTLE core. Second, for a memory operation which is well-suited to one waiting mechanism but instead adopts the other one, the conventional design suffers from a progress reduction of up to 18%, as opposed to 74% for the failure-resilient design. In other words, using an inappropriate waiting mechanism for a memory operation may lead to a more considerable progress reduction in intermittent systems than in traditional systems. This result agrees with the reason for adaptive synchronization (Section 2.2), which is particularly necessary for intermittent systems that feature high variations in the waiting time due to heterogeneous memories. However, the considerable difference in progress reduction would also suggest selecting the waiting mechanism according to the type of memory (instead of prior waiting time assessment) is sufficient to significantly compensate for the progress reduction.

The above-mentioned results could be observed in any intermittent-aware design which requires *alternate* and *frequent* accesses to heterogeneous memories to advance and preserve progress, and will be more evident when more tasks are concurrently executed or the system features a higher level of heterogeneity.

4 HETEROGENEITY-AWARE MULTICORE CONCURRENCY CONTROL

4.1 Design Overview

Motivated by the observations in Section 3, we propose a heterogeneity-aware design which leverages heterogeneous CPU cores to reduce the wasted computation resources due to heterogeneous memory accesses, thereby improving the forward progress of concurrent task execution on intermittent systems. A major challenge of realizing such a design is how to enable *delegable* and *adaptive synchronization*, which is sufficiently lightweight for intermittent systems, while guaranteeing task serializability across multiple cores.

Figure 3 illustrates our design which manages tasks executed concurrently on heterogeneous cores, as well as their data accesses to hybrid memories. To increase the flexibility of concurrency control, *two-version data management* allows two data versions respectively in VM and NVM for each data object, so that multiple tasks can simultaneously access the same data object during execution and then commit their data modifications from VM into NVM after execution. Because tasks on different cores can simultaneously access the same object, their data access operations must be synchronized to ensure an exclusive update on the object. To this end, *adaptive synchronization* is employed to avoid conflict operations while adapting to diverse waiting times incurred by different operations. Moreover, *delegable commitment* allows commit operations to be delegated between

CPU cores and merged with other commit operations, thereby mitigating the CPU time wasted by the high-speed big core on NVM updates without overloading the low-speed LITTLE core. Although task synchronization avoids simultaneous conflict operations, the operations performed by concurrently executed tasks may not maintain a serializable order under *optimistic* concurrency control. Therefore, before a task can actually perform its commit operation, its execution must be validated via *two-phase validation* to ensure serializability across multiple cores.

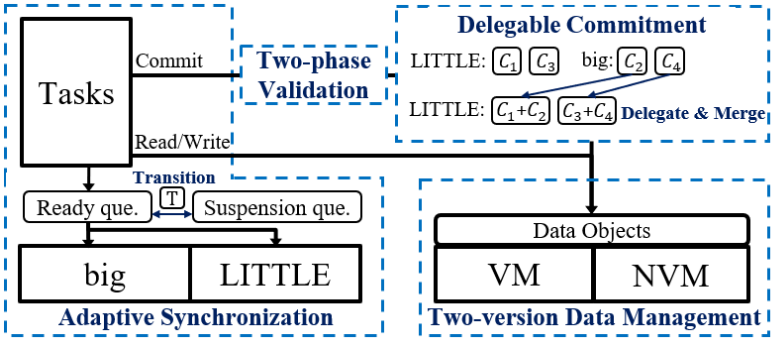


Fig. 3. Our Heterogeneity-aware design for Concurrency Control and Synchronization.

Two-version data management (detailed in Section 4.2) allows multiple tasks to simultaneously *read* and *write* the same data object during their execution while achieving instant recovery under intermittent power. Specifically, at the end of execution, a task atomically *commits* its modifications on data objects in VM into the corresponding objects in NVM, and the task is deemed *finished* once its commit operation is complete. Power failures may occur during task execution. Upon power resumption, only those unfinished tasks, which were lost in VM, are recreated to rerun, thereby preventing the data objects in NVM from being repeatedly modified by finished tasks. This allows data in NVM to be always kept consistent with the progress of finished tasks, thereby allowing instant system recovery upon power resumption.

Delegable and Adaptive Synchronization (detailed in Section 4.3) prevents tasks from simultaneously performing *conflict* data access operations while reducing the wasted CPU time. Specifically, a task must acquire the corresponding lock before it can read, write, or commit a data object which is also accessible by other tasks; otherwise, the task will wait until the lock is available. To adapt to high variations in memory access latency, *adaptive synchronization* allows frequent waiting mechanism transitions during *context switches*, with negligible transition overhead. Moreover, considering the high latency of NVM writes, *delegable commitment* enables the commit operations of tasks executed on the big core to be delayed and delegated to the LITTLE core. Meanwhile, to reduce the number of NVM writes, individual commit operations are allowed to be merged and *atomically* performed as a batch.

Two-phase validation (detailed in Section 4.4) ensures that only tasks which maintain serializability across multiple cores can commit their data modifications from VM to NVM. Specifically, when multiple tasks running on different cores attempt to commit their modifications simultaneously, the first phase allows tasks to be validated on different cores *in parallel* to reduce the validation time, while the second phase is protected by a global critical section to force all tasks under validation to be validated against one another. For tasks that violate serializability, they can be simply aborted and rerun without violating idempotence, because their data modifications have yet to be committed into NVM and will become invalid in VM.

4.2 Two-version Data Management

4.2.1 Version Allocation and Transition. To provide flexibility for concurrency control, our design allows two data versions, namely *consistent* version and *working* version, for each data object. A data object has only one consistent version in NVM but can have multiple working versions in VM. The consistent version represents the latest value committed by *serializable* tasks and is preserved permanently in NVM. On the other hand, working versions store intermediate values written by unfinished tasks and will be lost in VM upon a power failure. For each consistent version, we allow a *temporary* copy in VM to provide high speed data access. Working versions of a data object are allocated in a copy-on-write manner. Specifically, once a task first attempts to write a data object, a working version *dedicated* for the task will be allocated in VM. The working and consistent versions, as well as temporary copies, are manipulated by read, write, and commit operations, and thus allow tasks to simultaneously perform different operations on the same data objects via different versions and copies.

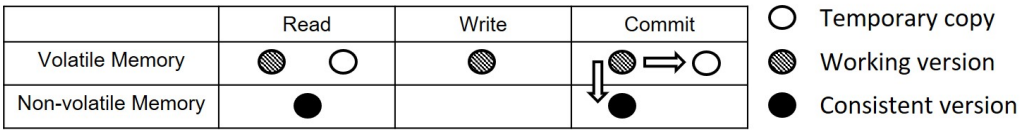


Fig. 4. The allocation and transition of data accessed by read, write, and commit operations [9].

Figure 4 shows the allocation and transition of data versions/copies accessed by the three operations. When a task attempts to read a data object, the temporary copy will be read by default to utilize high speed VM. If the temporary copy is not available, the consistent version in NVM will be read instead. When the task attempts to write the data object, a working version dedicated for the task will be immediately allocated in VM. Afterward, subsequent read and write operations conducted by the task on the data object will be made on its own working version. At the end of execution, the task will commit its modifications via a commit operation. Our design uses a backward validation procedure to validate whether the task is serializable (Section 4.4). If serializability is violated, the task is simply aborted and rerun. Otherwise, for each data object modified by the *validated* task, the commit operation is granted to update the consistent version in NVM as the corresponding working version, and the working version is directly transited into the temporary copy in VM without extra data movement. Once the consistent versions of all data objects modified by the validated task are successfully updated, the task is deemed *finished*. In short, for a data object, our design allows a working version dedicated for each task, and the temporary copy (resp. consistent version) is always transited (resp. updated) from the working version of the most recently validated (resp. finished) task and accessible by all tasks. Whenever an operation is to be performed on a shared data object, our design uses a task synchronization protocol to prevent the data object from being simultaneously accessed by conflict operations (Section 4.3).

4.2.2 Atomicity and Durability. A task cannot directly write on the consistent version but on a working version dedicated for the task in VM. The consistent versions in NVM will never be updated during task execution and can only be atomically updated after a task is validated as serializable. To enforce atomicity, our design implements atomic commit operations to prevent the consistent versions in NVM from being partially updated due to a power failure. When a power failure occurs during task execution, the working versions and temporary copies are directly lost in VM, as if those unfinished tasks had never been executed. Therefore, after power resumption, unfinished tasks can simply be recreated and rerun based on the consistent versions in NVM. Note

that finished tasks will not be recreated to prevent the consistent versions from being repeatedly updated, thus protecting idempotence. By maintaining the consistency between data objects and task progress at all times, our design not only enforces durability but also achieves instant system recovery.

4.3 Delegable and Adaptive Synchronization

4.3.1 Adaptive Synchronization. Multiple data versions allow tasks executed on different CPU cores to simultaneously access a data object to maximize forward progress, yet a task synchronization protocol is needed to prevent *conflict* operations from being simultaneously performed on the same data object. Two operations are deemed to conflict if both access the same data object and at least one of them updates the object. Thus, two read operations are not in conflict because they do not update data. Also, two write operations are not in conflict because each task writes on its own working version. In contrast, a commit operation cannot be simultaneously performed with a read or another commit operation on the same data object. This is because when a task is updating the temporary copy or consistent version of a data object, this object cannot be simultaneously accessed (i.e., read or committed) by another task. Thus, before reading or committing a data object shared with other tasks, a task must acquire the corresponding lock to obtain an exclusive access to the object.

If a task cannot acquire the desired locks due to some ongoing operations, it needs to enter either a busy-waiting or sleep-waiting mode until these operations are complete. Whenever a task attempts to access an occupied data object, our design determines how the task waits for the corresponding lock to reduce the wasted CPU time. Figure 5 shows the transition policy for a task to adapt to diverse waiting times incurred by different data access operations. If the task is waiting for the completion of any slow operations (i.e., NVM write), it enters sleep-waiting and releases the CPU core. In contrast, if the task is only waiting for fast operations (i.e., VM read, VM write, and NVM read) to complete, it enters busy-waiting to provide a quick response. Afterward, the task can dynamically transit between busy-waiting and sleep-waiting, with a change of those remaining memory operations it is waiting for.

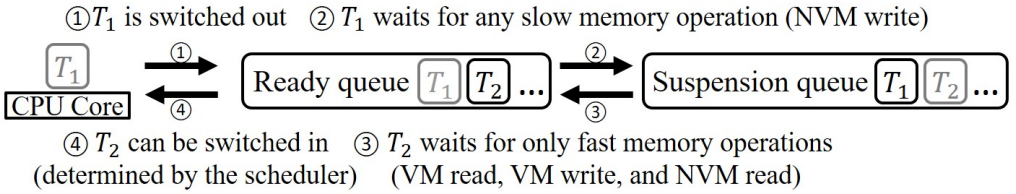


Fig. 5. A transition policy of adaptive synchronization.

As observed in Section 3, intermittent systems should be able to frequently transit between waiting mechanisms to adapt to alternate accesses of heterogeneous memories. To accommodate lightweight transitions, our design realizes adaptive synchronization by allowing tasks to be suspended to enter sleep-waiting and be resumed to enter busy-waiting during *context switch*, which avoids the additional overhead required to suspend and resume tasks for waiting mechanism transitions. As illustrated in Figure 5, once the task currently executed on a core is switched out by the scheduler (Step 1), those busy-waiting tasks in the ready queue and sleep-waiting tasks in the suspension queue are swapped according to the above-mentioned transition policy (Steps 2 and 3). Note that a task that has acquired all desired locks in the suspension queue will also be

moved into the ready queue, but it must acquire all desired locks in an all-or-none manner to avoid a *hold-and-wait* deadlock. Lastly, one of the tasks (either waiting or not) in the ready queue will be switched in by the scheduler to occupy the CPU core (Step 4). More implementation issues will be detailed in Section 5.

4.3.2 Delegable Commitment. Once a task is validated as serializable, its commit operation is granted. However, even with adaptive synchronization, the slow commit operation will incur an inevitable waste of CPU time. To transfer the wasted CPU time between cores, if the commit operation does not conflict with any ongoing operations, our design immediately transits the working versions of the task into temporary copies in VM but allows the updates of consistent versions in NVM to be delayed. Consequently, other unfinished tasks can still read temporary copies to obtain the latest values of data objects, while delayed updates on consistent versions can be delegated between cores and merged with other updates to reduce the NVM access cost.

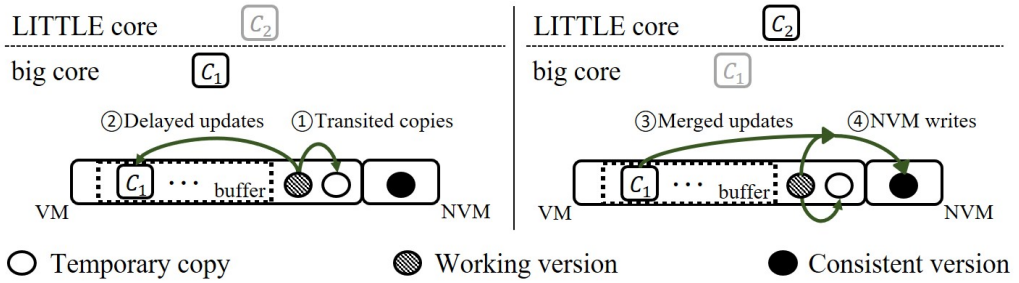


Fig. 6. Delayed and merged NVM updates via delegable commitment.

For ease of presentation, we use a heterogeneous dual-core CPU as an example. To improve forward progress, any commit operations invoked on the big core will be delayed by default and delegated to the LITTLE core. The reason is that the extent of reduced forward progress incurred by conducting NVM updates on a CPU core increases with the computing capability of the core, as observed in Section 3. In contrast, none of the commit operations on the LITTLE core will be delayed. Figure 6 illustrates how a commit operation is delegated from the big core to the LITTLE core, where its working versions are first transited into the temporary copies in VM (Step 1) while its NVM updates are delayed (Step 2). Then, delayed updates can be merged with another commit operation (Step 3) carried out by the LITTLE core when any task executed on the core attempts to update some of the same consistent versions (Step 4). Note that delegable commitment is compatible with adaptive synchronization, in that other tasks can still wait for ongoing conflict operations by adaptively using an appropriate waiting mechanism.

To this end, our design stores all delayed updates in a VM buffer shared by both cores. For each consistent version, only the latest delayed update is stored in the buffer because preceding delayed updates on the same consistent version will eventually be overwritten by the latest one. Delaying the updates of a validated task on consistent versions can reduce the waiting time of other unfinished tasks that attempt to access the corresponding data objects, while merging several updates into one update reduces the amount of CPU time wasted by the LITTLE core to perform slow NVM updates. However, if the buffer which stores delayed updates is full, the big core needs to perform all delayed NVM updates to preserve forward progress in a timely manner. Delegable commitment can also be applied to a multicore CPU featuring more high-speed and low-speed cores. Specifically, each high-speed core is a producer of delayed updates into the buffer, while each

low-speed core consumes some delayed updates from the buffer along with its commit operations. Note that simultaneous accesses from multiple cores to the buffer can be synchronized with global semaphores, in a way similar to the typical *producer-consumer synchronization* protocol [38].

To prevent the consistent versions from being corrupted due to partial updates, the updates of a commit operation must be performed in an all-or-none manner. This implies that, for a commit operation, all its updates on consistent versions must be either delayed or performed together. Thus, if some updates of a commit operation are merged with some updates of other commit operations, all updates of these operations must be performed together (and a cascade is possible). Moreover, once performed, all the updates must be *atomically* completed to prevent the consistent versions in NVM from being partially updated, and once all the updates are completed, the corresponding tasks must be successfully marked as finished to protect idempotence. The implementation issues will be detailed in Section 5.

4.4 Two-phase Backward Validation

4.4.1 Multicore Validation. Whenever a task attempts to perform a commit operation, our design ensures serializability across multiple CPU cores via a *two-phase* validation procedure. The two-phase validation procedure shares a similar idea with a one-phase validation approach [9, 10] to ensure serializability. Differently, the previous approach is intended for single-core systems and thus can validate only one task at a time, which results in a long validation procedure when multiple tasks running on different cores attempt to commit their modifications simultaneously. Hence, our procedure allows multiple tasks to be *simultaneously* validated on different cores in the first phase to reduce the waiting time incurred by the validation procedure, and protects the second phase by a *global critical section* to ensure that all the tasks under validation are validated against one another.

To maintain a serializable order, for each task to be validated, the one-phase validation procedure will try to derive a *validity time interval*, in which the task can be viewed as having been executed logically in isolation. The validity time interval is derived according to the read and write operations performed by the task on data objects during its execution, in an attempt to serialize the precedence relationship between the task and all the validated tasks which have ever accessed some of the same data objects. If a non-empty validity time interval can be derived for the task, the task is serializable and its commit operation is allowed; otherwise, it is aborted and rerun. However, the validation procedure of a task needs to be protected by a critical section to prevent other tasks from being validated during the procedure, ensuring that the task is validated against all tasks that have been previously validated as serializable.

Our two-phase validation procedure extends the one-phase procedure to avoid using a global critical section across multiple cores during the whole procedure. Specifically, each phase uses the one-phase procedure to validate the task under validation against some previously validated tasks that share data objects with the task. In the first phase, the task is validated against all tasks which have been previously validated *before* the procedure starts. While the task is being validated, other tasks can be validated on different cores in parallel and may be validated as serializable ahead of the task. In the second phase, which is protected by a global critical section, the task is validated against those newly validated tasks *after* the procedure starts. Consequently, multiple tasks may pass *partial* validation in the first phase, but only one at a time can enter the second phase to finish the *complete* validation, forcing multiple tasks under validation to be validated against one another. Note that the second phase is usually much shorter than the first phase because the number of tasks involved in the second phase is much smaller than in the first phase.

4.4.2 Serializability. Previous work [10], where one task is validated at a time, has proven that any serializable task validated by the one-phase procedure is *conflict-serializable* [16, 23], by constructing

a precedence graph and showing that the graph is acyclic. Specifically, each node of the graph represents a validated task, and a directed edge between two nodes represents the precedence relationship between two tasks due to their data access orders. The idea is to prove that the to-be-validated task has been validated against all previously validated tasks that share data objects with the task.

Similarly, the serializability of tasks validated by our two-phase procedure, where multiple tasks are validated simultaneously, can be proven in the same way. Specifically, each task is validated at the first phase against all tasks that have previously been validated before it enters the first phase, as well as at the second phase against those newly validated tasks that complete their validation after the task enters the first phase. Moreover, during the second phase, which is protected by a global critical section, no other tasks are allowed to complete their validation. Thus, those tasks validated by our two-phase procedure are also conflict-serializable because every task is validated against all tasks that have been validated as serializable before the task completes its validation.

5 A MULTICORE INTERMITTENT OPERATING SYSTEM

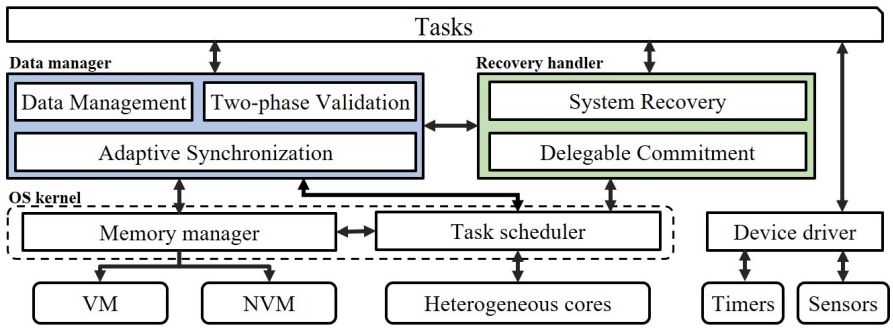


Fig. 7. FreeRTOS-extended system architecture.

We integrated our heterogeneity-aware design into FreeRTOS [3], a real-time operating system supporting many kinds of commercial microcontrollers, to realize an intermittent multicore operating system. Our implementation follows the system architecture of a FreeRTOS-extended operating system [2] developed according to a failure-resilient design intended for single-core intermittent systems [9, 10], by extending two modules, namely a *data manager* and a *recovery handler*, to support heterogeneity-aware multicore synchronization, as shown in Figure 7, with additional 1218 lines of C code scattered into 13 files. Our FreeRTOS-extended operating system is then deployed on a Cypress CY8CKIT-062-WiFi-BT device, which features heterogeneous dual cores and hybrid memories to enable intermittent-aware multicore concurrency control.

5.1 Data Manager

Our data manager, which improves on the data manager developed for single-core intermittent systems in [9], is responsible for data management, adaptive synchronization, and two-phase validation. Specifically, the data management is extended to allow tasks on multiple cores to concurrently access data objects, the adaptive synchronization is added to synchronize data access operations across heterogeneous cores, and the two-phase validation extends the one-phase validation to simultaneously validate multiple tasks on different cores.

To this end, the data manager maintains memory space used by data versions and copies in hybrid memories via the functions, `pvPortMalloc()` and `vPortFree()`, provided by the memory

manager to allocate data and reclaim invalid data. A data structure is maintained in NVM to record the location and size of the consistent version of each data object, whereas the corresponding information of temporary copies and working versions are maintained in VM. To avoid data corruption, tasks must access data objects through our read, write, and commit operations, which replace the original implementations provided by FreeRTOS. Unlike single-core systems where only one task can perform its data access operation at a time, multicore systems allow multiple tasks to simultaneously perform their operations on different cores while ensuring an exclusive update on any data object.

Conflict operations on shared data objects must be synchronized. We implement semaphores and spinlocks with a hardware locking mechanism supported by the microcontroller to allow tasks executed on different cores to atomically acquire a lock. To read or commit (but not write) a data object, a task must acquire permission to set the corresponding hardware lock, and other tasks that invoke conflict operations will wait for the lock occupied by the task. To enable tasks to transit between busy-waiting and sleep-waiting, the data manager uses the functions, `vTaskSuspend()` and `vTaskResume()`, provided by the task scheduler to respectively suspend running tasks and resume suspended tasks for adaptive synchronization. To accommodate frequent transitions, the data manager allows waiting mechanism transitions only during context switch. In addition, instead of moving tasks between the ready and suspension queues, our implementation uses only one queue to store all the tasks and modify the scheduler to prevent sleep-waiting tasks from being selected for execution.

To validate whether a task is serializable, the data manager records its read and write operations on data objects in VM during task execution, and tries to derive a non-empty interval which does not overlap with the validity time interval of any validated task that has ever accessed data objects shared with the task. For ease of validation, each data object is associated with the validity time interval of the most recently validated task which transits its working copy as the temporary copy. In our two-phase validation procedure, the first phase need not to be mutually exclusive, yet the second phase is protected by a global critical section, which is usually short and thus implemented with a spinlock. Moreover, the time granularity of validity time intervals is set as a single system time tick, which is triggered every 2 ms in FreeRTOS, to exempt our implementation from the extra overhead of maintaining the system time.

5.2 Recovery Handler

Our recovery handler, which improves on the recovery handler in [9], is responsible for delegable commitment and instant system recovery. Specifically, our recovery handler additionally enables commit operations to be delayed and delegated between CPU cores, and also extends the atomic commit that can finish only one task at a time to finish multiple tasks together.

Once a commit operation is delayed, the data objects modified by the corresponding task and the delayed updates will be stored in a VM buffer, and they will be removed from the buffer after the operation is performed by some CPU core to actually update the consistent versions in NVM. Note that delegable commitment is orthogonal to how NVM updates are implemented, regardless of synchronous or asynchronous I/O. In our implementation, NVM updates are conducted asynchronously via a *direct memory access* (DMA) controller to reduce the CPU time wasted during data movement. However, the CPU still requires some time to interact with the DMA controller, and delegable commitment can transfer the wasted CPU time from the big core to the LITTLE core.

To accommodate frequent power failures, the recovery handler maintains the attributes of unfinished tasks in a data structure in NVM, and uses the function, `xTaskCreate()`, provided by the scheduler to directly recreate unfinished tasks upon power resumption. Instant system recovery is possible because the consistent versions are always kept consistent with the progress of finished

tasks and available in NVM. To this end, those commit operations with merged NVM updates must be able to be performed *atomically*. Moreover, once all the updates are completed, the corresponding tasks must be successfully marked as finished. In [10], a *shadowing mechanism* is used to atomically update none or all of the data objects modified by a task and, meanwhile, to update the data structure of unfinished tasks as if it were also a data object. Our implementation extends the shadowing mechanism so that multiple commit operations, along with marking multiple tasks as finished, can be finalized by one single CPU instruction.

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

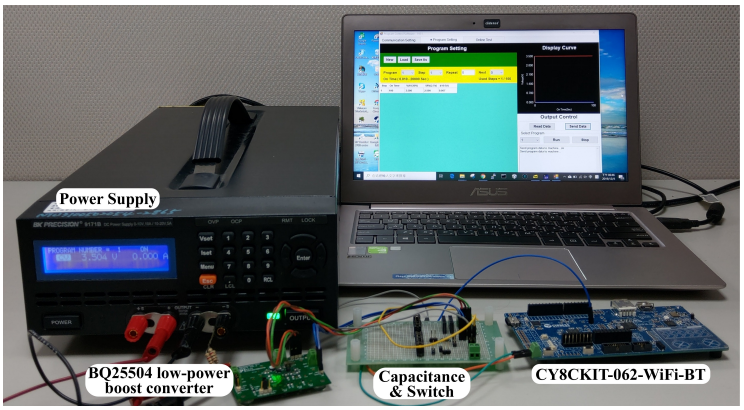


Fig. 8. The experimental environment.

To evaluate the proposed design, we installed our FreeRTOS-extended intermittent operating system on a Cypress device, as described in Section 5. Figure 8 shows the experimental environment. The device is equipped with heterogeneous cores and memories. Moreover, the operating frequency of both the cores can be scaled to up to 100 MHz[2], whereas the throughput of NVM writes is only 256 kBps. We powered the device by an energy harvesting management (EHM) module composed of a BQ25504 low-power boost converter, a 1mF capacitor to store the harvested energy, and a switch to turn on (resp. off) the power supply when the voltage of the capacitor rises above 2.8 V (resp. drops below 2.4 V). Table 1 details the hardware and software specifications.

We used a programmable power supply made by B&K Precision to emulate energy-harvesting sources. As many intermittent applications like environment monitoring [48] adopted solar power, we manufactured strong (70 mW) and weak (24.5 mW) power sources to respectively emulate the average magnitude of power produced by a small (5cm^2) solar cell during daylight and evening hours [30]. Each power source lasted 100 seconds, which was sufficient to mitigate experimental variances while making experiments reproducible. Note that neither power source was sufficient for the device to operate continuously, leading to repeated yet unpredictable power failures at runtime. The resultant power-on and power-off periods depended on the capacitor size and the amounts of energy harvested and consumed during system operation. Given the used capacitor and power sources, power-on periods were approximately dozens of milliseconds.

[2]The frequency of the LITTLE core can only be set at lower or equal to that of the big core, and the big core can execute more instructions per cycle than the LITTLE core even when their operating frequencies are the same.

Table 1. Specifications of the experimental platform.

Hardware	
MCU	PSoC 62
CPU	Cortex M4 up to 100 MHz
	Cortex M0+ up to 100 MHz
Memory	288KB SRAM & 1MB Flash
Software	
OS	FreeRTOS V9.0.0
Energy	
Capacitance	1 mF
Switch on/off voltage	2.8 V/2.4 V
Strong power	70 mW = 3.5 V × 20 mA
Weak power	24.5 mW = 3.5 V × 7 mA

Considering that energy-harvesting devices are typically intended for lightweight applications such as sensing to detect events and simple data processing, with privacy usually considered, we ported two sensing tasks provided by Cypress and six computational tasks from the Texas Instruments MSP430 Competitive Benchmarking. To emulate different levels of task concurrency, we investigated two respective scenarios, namely low and high concurrency workloads. Specifically, in one scenario, the LITTLE core executes the two sensing tasks which respectively sense temperature and detect touch events from the on-board capacitive touch sensor, and the big core executes two computational tasks which respectively perform discrete Fourier transform and SHA-256 hashing algorithm on the collected data. In the other scenario, apart from the four tasks, the LITTLE (resp. big) core additionally executes two computational tasks which respectively performance one-dimensional (resp. two-dimensional) matrix multiplication and integer (resp. floating-point) arithmetic operations. All tasks were run repeatedly, and forward progress (defined as the number of finished tasks per second) was adopted as the performance metric.

We conducted three sets of experiments. First, we evaluated the system costs incurred by our design, where the data manager and the recovery handler require additional computation time and memory space. Then, we compared the performance achieved by our design (*OURS*) with that achieved by *system-wise checkpointing* (*SYS*) [19] and *failure-resilient non-checkpointing* (*NON*) [9] designs, under different combinations of concurrency workloads and core frequencies, allowing us to observe how different heterogeneity levels can improve forward progress under different concurrency levels. Note that all the three designs were respectively integrated into FreeRTOS for a fair comparison[3]. Finally, to further analyze the causes behind performance differences, we explored the runtime overheads incurred by different designs. In particular, we measured the wasted CPU time, system recovery time, and data recency achieved by each design. These runtime overheads affect the forward progress and data quality.

6.2 System Costs

Our design ensures the serializability of concurrent task execution and achieves system recovery in multicore intermittent systems at the cost of additional computation time and memory space. At runtime, the data manager manages multi-version data objects, synchronizes accesses to shared objects, and validates the serializability of concurrent task execution. The recovery handler records

[3]NON was integrated into FreeRTOS and publicly available [2]. We integrated SYS, similar in concept to the design proposed in [19], into FreeRTOS.

attributes of unfinished tasks and atomically updates data objects committed by finished tasks. To evaluate the costs, we measured the average computation time required by each data access operation, as well as the memory space used by the data manager and recovery handler to maintain data versions and copies, attributes of unfinished tasks, and the precedence relationship between finished tasks.

Table 2. Operation execution time and memory space required by our design.

	Read	Write	Validation	Commit
Average execution time	39.7 μs	49.5 μs	63.2 μs	1181.1 μs
	Data manager		Recovery handler	
Additional memory usage	1458 bytes		1376 bytes	

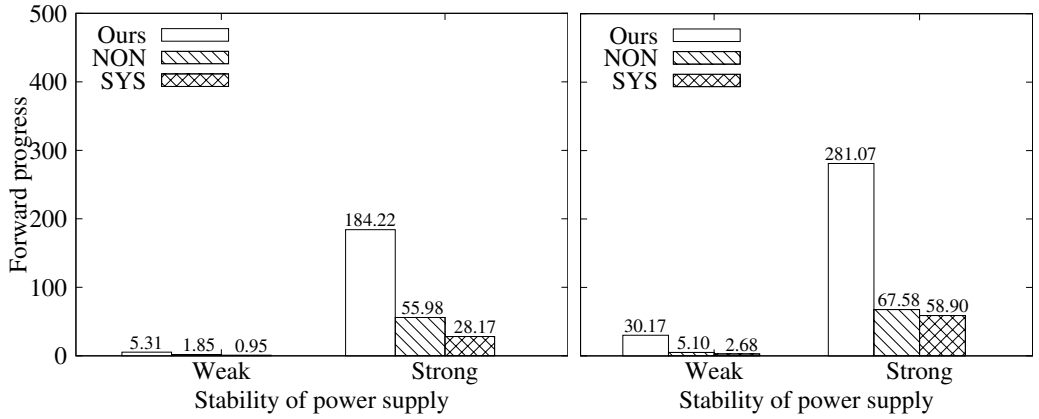
Table 2 lists the average execution time required by each data access operation and memory space used by the data manager and the recovery handler. To ensure serializability, the data manager requires extra execution times for synchronization and two-phase validation. The average time required to execute the validation procedure is 63.2 μs, while the average execution times required by the read, write, and commit operations are respectively 39.7, 49.5, and 1181.1 μs. Relatively, the commit operation requires much more time than the other operations because it involves NVM writes. However, compared to the high access latency of NVM (i.e., a dozen milliseconds on the Cypress platform), the execution time required by the commit operation is insignificant because NVM writes are conducted via DMA and thus in parallel to task execution on the CPU in our implementation. Considering that the execution time of a task (even without waiting for other tasks) is from a few to dozens of milliseconds, the execution times of these operations are marginal. In addition, the data manager and the recovery handler respectively use only 1458 bytes and 1376 bytes over the 1MB + 288KB memory space to maintain multi-version data objects and record task attributes in VM and NVM. Thus, both the time and space costs of our design are justifiable.

6.3 Forward Progress

We compared the forward progress respectively achieved by different designs. SYS enables intermittent computing by checkpointing a snapshot of VM and CPU registers. To preserve forward progress across power cycles, SYS checkpoints a system snapshot every 20 ms. By contrast, NON enables intermittent computing by allowing a task to commit its data modifications only after its execution. SYS was originally intended for serial task execution. To enable concurrent task execution, SYS borrows from NON an intermittent-aware concurrency control protocol with a one-phase validation procedure [9], as described in Section 2.1. Both SYS and NON were originally designed for single-core systems. We assume that they adopt semaphores when run on a multicore system, because semaphores are relatively efficient compared to spinlocks in an intermittent system with heavy NVM updates, as shown in Section 3.

To fully investigate the efficacy of our design, which leverages core heterogeneity to improve forward progress, we measured the forward progress under three different heterogeneity levels. First, the operating frequencies of both cores are set identically (50 MHz). Then, we measured the improved forward progress when the operating frequency of the big core is scaled up from 50 to 100 MHz with the LITTLE core kept at 50 MHz. Finally, the frequency of the LITTLE core is scaled down from 50 to 25 MHz with the big core kept at 100 MHz. The three settings are respectively denoted as low, medium, and high heterogeneity levels, allowing us to investigate the impact of different core heterogeneity on the proposed dynamic delegation. Moreover, we measured the forward progress achieved under two different concurrency levels, which allows us to evaluate

the proposed adaptive synchronization when the CPU resource is utilized by different numbers of concurrently executed tasks.

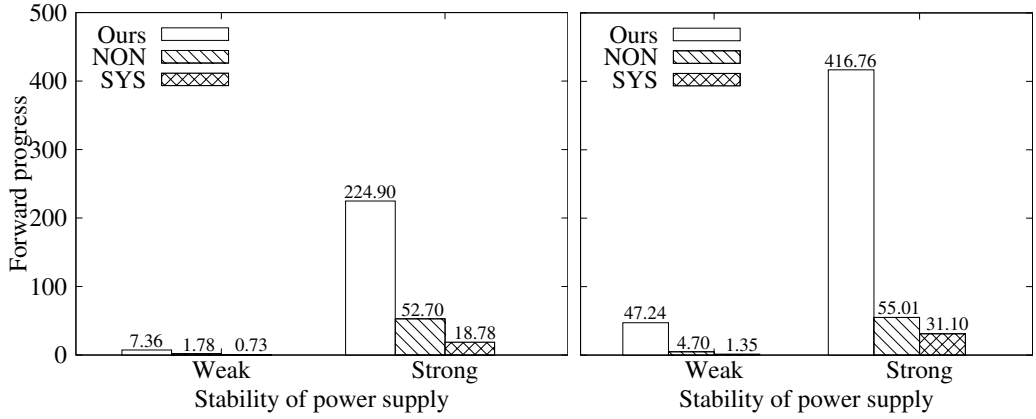


(a) Low concurrency workload (b) High concurrency workload

Fig. 9. Forward progress achieved under low and high concurrency workloads with low CPU heterogeneity.

Figures 9(a) and 9(b) respectively show the forward progress achieved under the low and high concurrency workloads when both cores are set at 50 MHz. In general, OURS and NON achieve more forward progress than SYS by eliminating the long suspension and recovery times incurred by system checkpointing. OURS can further achieve 2.87 to 5.92x forward progress achieved by NON by allowing the waiting mechanism to adapt to high variations in memory access latency. Moreover, the progress improvement increases with the concurrency level, because the efficacy of adaptive synchronization becomes more apparent when more concurrently executed tasks wait for a data access operation. If the operation is fast, OURS allows waiting tasks to provide quick responses while preventing additional context switches. In contrast, if the operation is slow, OURS allows waiting tasks to release CPU cores, preventing a waste of CPU time on spinning. Additionally, as shown in Figure 9(b), compared to NON and SYS, OURS respectively increases forward progress by 3.22 to 5.92x and 4.77 to 11.25x when the power supply is relatively weak. This is because OURS can make progress even given short power-on periods by committing data objects and finishing tasks in a timely fashion. The results imply that, compared with NON and SYS, OURS is more suitable for use in intermittent systems with high concurrency workloads and frequent power failures.

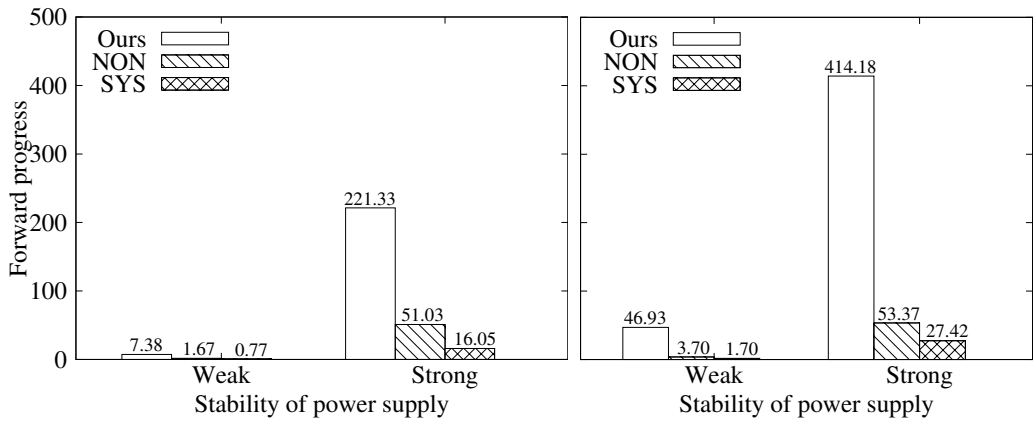
Figures 10(a) and 10(b) respectively show the forward progress achieved under the low and high concurrency workloads when the operating frequency of the big core is scaled up from 50 to 100 MHz. The results are similar to those found in Figure 9. Interestingly, as the operating frequency of the big core increases, the forward progress achieved by NON and SYS is reduced, whereas that achieved by OURS is still increased. By comparing the results in Figures 9(a) and 10(a), we observe that as the big core frequency increases under the weak (resp. strong) power supply, the forward progress achieved by OURS is improved by 38.6% (resp. 167.1%), whereas that achieved by NON and SYS is respectively reduced by 3.7% and 23% (resp. 5.8% and 33.3%). Comparing Figures 9(b) and 10(b) finds similar results. The reduced progress is because, as the frequency increases, the waiting times incurred by commit operations under SYS or NON are not correspondingly reduced, yet the power consumption is increased, thus increasing the cost of using the big core to perform a



(a) Low concurrency workload (b) High concurrency workload

Fig. 10. Forward progress achieved under low and high concurrency workloads with medium CPU heterogeneity.

memory access. In contrast, OURS enables commit operations on the big core to be delayed and delegated to the LITTLE core, thereby allowing the big core to make more forward progress as its frequency increases.



(a) Low concurrency workload (b) High concurrency workload

Fig. 11. Forward progress achieved under low and high concurrency workloads with high CPU heterogeneity.

Figures 11(a) and 11(b) show the forward progress achieved when the frequency of the LITTLE core is scaled down from 50 to 25 MHz. Although the results are similar to those found in Figure 10, the reasons for different approaches to maintain forward progress are different. For OURS, the forward progress is mainly made by the big core and remains similar as the big core operates at the same frequency. On the other hand, the LITTLE core, on which multiple commit operations are

merged and performed as a batch, is not overloaded by delegated commit operations even with a lower frequency. In contrast, for NON and SYS, as the forward progress is already limited by the wasted CPU time due to extensive memory access, the reduced computation resource does not lead to a significant decrease in progress. The experimental results above show that OURS can leverage the huge performance gap between the CPU and memory, and also explains why OURS can respectively achieve up to 11.6x and 33.9x progress improvements compared to NON and SYS. The result also implies that OURS is particularly suitable for use in heterogeneous intermittent systems with asymmetric operating frequencies.

6.4 Runtime Overheads

To analyze the causes behind the performance differences among different designs, we measured runtime overheads, including the wasted CPU time (defined as the percentage of the CPU time spent by waiting tasks over the total CPU time), the recovery time (defined as the time required for the first unfinished task to be ready to run after power resumption), and the data recency (defined as the time difference between the latest data update and the completely recovered system).

Table 3. Runtime overheads incurred by different designs with low CPU heterogeneity.

	OURS		NON		SYS	
	LITTLE	big	LITTLE	big	LITTLE	big
Wasted CPU time (%) - low	0.1	0.5	2	15.4	57.2	57.3
Wasted CPU time (%) - high	0.2	1.1	4.6	22	59.1	59.4
Recovery time (ms)	0.62		0.63		2.53	
Data recency (ms) - low	7.3		7.3		48.5	
Data recency (ms) - high	7.4		7.3		49.1	

Table 3 shows the runtime overheads when the frequencies of both CPU cores are set at 50 MHz. Overall, OURS reduces the CPU times wasted under NON and SYS by one to three orders of magnitude by adaptively transiting waiting mechanisms and merging several NVM updates. For the LITTLE and big cores under different concurrency levels, the wasted CPU times under OURS are only 0.1 to 1.1%, whereas the wasted CPU times under NON and SYS vary from 2 to 22% and 57.2 to 59.4%, respectively. The reduction in the wasted CPU time is consistent with the significant forward progress improvement achieved by OURS, as shown in Section 6.3. Moreover, the system recovery times required by OURS, NON, and SYS are respectively 0.62, 0.63, and 2.53 ms. OURS and NON recreate unfinished tasks based on task attributes stored in NVM after power resumption and thus require about the same amount of recovery time. By eliminating memory restoration during recovery, OURS and NON reduce the recovery time required by SYS by 75%. Also, by committing data modifications immediately before tasks are finished, OURS and NON can significantly improve the data recency achieved by SYS, which reverts the data back to the latest checkpoint after system recovery. Note that although OURS allows commit operations on the big core to be merged and delegated to the LITTLE core, the data recency is not adversely affected. This is because when shared data objects are frequently accessed by tasks on both cores, delayed commit operations on the big core will be merged with others and timely performed by the LITTLE core.

Table 4 shows the runtime overheads when the frequency of the big core is scaled up to 100 MHz with the LITTLE core kept at 50 MHz. Compared to NON and SYS, OURS still achieves a significant reduction in the wasted CPU time. Under OURS, the increased frequency does not have an obvious impact on the wasted CPU time of the big core. In contrast, under NON, the wasted CPU time of the big core increases accordingly. This is because the increased frequency allows the big core to finish

Table 4. Runtime overheads incurred by different designs with medium CPU heterogeneity.

	OURS		NON		SYS	
	LITTLE	big	LITTLE	big	LITTLE	big
Wasted CPU time (%) - low	0.1	0.7	1.3	19.3	50.4	50.4
Wasted CPU time (%) - high	0.2	1.6	2	29.7	60.1	60.1
Recovery time (ms)	0.33		0.33		1.89	
Data recency (ms) - low	7.4		7.3		50.8	
Data recency (ms) - high	7.4		7.2		48.6	

tasks more quickly and thus perform data access operations more frequently. Thus, the efficacy of delegable and adaptive synchronization becomes more manifest. The wasted CPU time under SYS does not increase with the CPU frequency because the frequency of checkpointing, which dominates the wasted CPU time due to long runtime suspension, does not increase accordingly. With the increased frequency, the recovery times required by OURS, NON, and SYS are respectively reduced by 47.8, 47.6, and 25.2%. By contrast, the data recency is not evidently affected because the latency of NVM updates is not reduced. Similar results can also be observed in Table 5, where the frequency of the LITTLE core is scaled down to 25 MHz with the big core kept at 100 MHz. Notably, even if the LITTLE core operates at a lower frequency, OURS can still maintain the data recency because OURS allows commit operations to be merged so that the LITTLE core can still timely perform delegated commit operations despite slow NVM writes.

Table 5. Runtime overheads incurred by different designs with high CPU heterogeneity.

	OURS		NON		SYS	
	LITTLE	big	LITTLE	big	LITTLE	big
Wasted CPU time (%) - low	0.3	1.15	1.6	20.4	60	58.6
Wasted CPU time (%) - high	0.2	1.5	1.3	33.6	56.8	56.1
Recovery time (ms)	0.33		0.33		1.89	
Data recency (ms) - low	7.4		7.3		50.4	
Data recency (ms) - high	7.4		7.3		47.8	

To sum up, extensive experiments based on a prototype system running benchmark tasks demonstrate that OURS not only improves concurrent task execution but also enables instant system recovery. By allowing waiting tasks to adaptively transit between different waiting mechanisms, OURS reduces the wasted CPU times of both CPU cores, thereby improving forward progress. Furthermore, by allowing slow memory operations to be dynamically delegated between cores, OURS trades the CPU time of the LITTLE core for allowing the big core additional time to improve forward progress. The experimental results also suggest that OURS is particularly suitable for heterogeneous multicore devices which may suffer from frequent power failures.

7 CONCLUDING REMARKS

This paper advocates that intermittent systems should move toward heterogeneous multicore architectures, so that core heterogeneity can be leveraged to mitigate the forward progress reduction caused by memory heterogeneity. To demonstrate the efficacy of this proposition, we developed a heterogeneity-aware multicore synchronization protocol, which allows respective cores to be primarily engaged in CPU computations and slow memory operations. Our design was integrated

into FreeRTOS [3] to realize an intermittent-aware multicore operating system capable of concurrent task execution and instant system recovery. We installed the operating system on a Cypress device featuring heterogeneous dual cores and hybrid memories to conduct a series of experiments. Compared to failure-resilient non-checkpointing [9] and system-wise checkpointing [19] designs, our design can respectively improve forward progress by 4.9x and 12.7x on average, with justifiable system costs and runtime overheads. Our design is also found to be particularly effective for high concurrency workloads under frequent power failures.

The source code of our multicore intermittent operating system has been made open [1], facilitating programmers in developing applications on energy-harvesting multicore devices regardless of power stability, while exempting them from the responsibility of handling task concurrency and synchronization.

REFERENCES

[1] A Heterogeneity-aware Multicore Intermittent Operating System. Available: https://github.com/EMCLab-Sinica/Intermittent-Multicore.

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

[3] The FreeRTOS™ Kernel. Available: https://www.freertos.org.

[4] 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.

[5] 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.

[6] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-Level NUMA Systems. In *Proc. of ACM PPoPP*, pages 215–226, 2015.

[7] M. Chabbi and J. Mellor-Crummey. Contention-Conscious, Locality-Preserving Locks. In *Proc. of ACM PPoPP*, pages 1–14, 2016.

[8] W.-M. Chen, Y.-T. Chen, P.-C. Hsiu, and T.-W. Kuo. Multiversion Concurrency Control on Intermittent Systems. In *Proc. of IEEE/ACM ICCAD*, pages 1–8, 2019.

[9] W.-M. Chen, P.-C. Hsiu, and T.-W. Kuo. Enabling Failure-resilient Intermittently-powered Systems Without Runtime Checkpointing. In *Proc. of IEEE/ACM DAC*, pages 1–6, 2019.

[10] 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.

[11] 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.

[12] A. Colin and B. Lucia. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. of ACM OOPSLA*, pages 514–530, 2016.

[13] J. de Winkel, V. Kortbeek, J. Hester, and P. Pawełczak. Battery-Free Game Boy. *In Prof. of ACM IMWUT*, 4(3), 2020.

[14] Z. Ghodsi, S. Garg, and R. Karri. Optimal Checkpointing for Secure Intermittently-powered IoT Devices. In *Proc. of IEEE/ACM ICCAD*, pages 376–383, 2017.

[15] G. Gobieski, N. Beckmann, and B. Lucia. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proc. of ACM ASPLOS*, pages 199–213, 2019.

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

[17] J. Hester and J. Sorber. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proc. of ACM SenSys*, pages 1–6, 2017.

[18] M. Hicks. Clank: Architectural Support for Intermittent Computation. In *Proc. of ISCA*, pages 228–240, 2017.

[19] 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.

[20] 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 IEEE/ACM ISLPED*, pages 29:1–29:6, 2018.

[21] 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.

[22] P. Kansakar and A. Munir. Selecting Microarchitecture Configuration of Processors for Internet of Things (IoT). *IEEE TETC*, 8(4):973–985, 2018.

[23] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM TODS*, 6(2):213–226, 1981.

[24] 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.
[25] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 28(5):25–35, 1994.
[26] Y. Lin, P. Hsiu, and T. Kuo. Autonomous I/O for Intermittent IoT Systems. In *Proc. of IEEE/ACM ISLPED*, pages 1–6, 2019.
[27] 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.
[28] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, S. John, Y. Xie, J. Shu, and H. Yang. Ambient Energy Harvesting Nonvolatile Processors: From Circuit to System. In *Proc. of IEEE/ACM DAC*, pages 150:1–150:6, 2015.
[29] B. Lucia and B. Ransford. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proc. of ACM PLDI*, pages 575–585, 2015.
[30] 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.
[31] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent Execution Without Checkpoints. *Proc. of ACM OOPSLA*, pages 96:1–96:30, 2017.
[32] K. Maeng and B. Lucia. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proc. of USENIX OSDI*, pages 129–144, 2018.
[33] K. Maeng and B. Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proc. of ACM PLDI*, pages 1101–1116, 2019.
[34] H. R. Mendis and P.-C. Hsiu. Accumulative Display Updating for Intermittent Systems. *ACM TECS*, 18(5s):72:1–72:22, 2019.
[35] S. Nayar, D. Sims, and M. Fridberg. Towards Self-Powered Cameras. In *Proc. of IEEE ICCP*, pages 1–10, 2015.
[36] B. Ransford and B. Lucia. Nonvolatile Memory is a Broken Time Machine. In *Proc. of MSPC*, pages 5:1–5:3, 2014.
[37] 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.
[38] M. Raynal and D. Beeson. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, MA, USA, 1986.
[39] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. ThyNVM: Enabling Software-transparent Crash consistency in Persistent Memory Systems. In *Proc. of IEEE/ACM MICRO*, pages 672–685, 2015.
[40] E. Ruppel and B. Lucia. Transactional Concurrency Control for Intermittent, Energy-Harvesting Computing Systems. In *Proc. of ACM PLDI*, pages 1085–1100, 2019.
[41] 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.
[42] V. Talla, B. Kellogg, S. Gollakota, and J. R. Smith. Battery-Free Cellphone. *In Prof. of ACM IMWUT*, 1(2), 2017.
[43] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M. F. Chiang, Y. Yan, B. Sai, and H. Yang. A 3us Wake-up Time Nonvolatile Processor Based on Ferroelectric Flip-flops. In *Proc. of IEEE ESSCIRC*, pages 149–152, 2012.
[44] 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*, pages 38:1–38:25, 2018.
[45] 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 IEEE/ACM DAC*, pages 1–6, 2015.
[46] M. Xie, M. Zhao, C. Pan, H. Li, Y. Liu, Y. Zhang, C. J. Xue, and J. Hu. Checkpoint Aware Hybrid Cache Architecture for NV Processor in Energy Harvesting Powered Systems. In *Proc. of IEEE/ACM CODES+ISSS*, pages 1–10, 2016.
[47] T. Xu and M. Potkonjak. Energy-efficient Fault Tolerance Approach for Internet of Things Applications. In *Proc. of IEEE/ACM ICCAD*, pages 62:1–62:8, 2016.
[48] 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), 2020.
[49] 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*, 36:1804–1816, 2017.