Cache-aware Task Decomposition for Efficient Intermittent Computing Systems

Shuo Xu
Shandong University
shuoxu@mail.sdu.edu.cn

Wei Zhang*
Shandong University
sduzhangwei@sdu.edu.cn

Mengying Zhao
Shandong University
zhaomengying@sdu.edu.cn

Zimeng Zhou
Shandong University
Quan Cheng Laboratory
zhouzimeng@sdu.edu.cn

Lei Ju
Quan Cheng Laboratory
sr-julei@qcl.edu.cn

ABSTRACT

Energy harvesting offers a scalable and cost-effective power solution for IoT devices, but it introduces the challenge of frequent and unpredictable power failures due to the unstable environment. To address this, intermittent computing has been proposed, which periodically backs up the system state to non-volatile memory (NVM), enabling robust and sustainable computing even in the face of unreliable power supplies. In modern processors, write back cache is extensively utilized to enhance system performance. However, it poses a challenge during backup operations as it buffers updates to memory, potentially leading to inconsistent system states. One solution is to adopt a write-through cache, which avoids the inconsistency issue but incurs increased memory access latency for each write reference. Some existing work enforces a cache flushing before backups to maintain a consistent system state, resulting in significant backup overhead. In this paper, we point out that although cache delays updates to the main memory, it may preserve a recoverable system state in the main memory. Leveraging this characteristic, we propose a cache-aware task decomposition method that divides an application into multiple tasks, ensuring that no dirty cache lines are evicted during their execution. Furthermore, the cache-aware task decomposition maintains an unchanged memory state during the execution of each task, enabling us to parallelize the backup process with task execution and effectively hide the backup latency. Experimental results with different power traces demonstrate the effectiveness of the proposed system.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Hardware → Analysis and design of emerging devices and systems**.

*Corresponding author.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3657382

KEYWORDS

intermittent computing, cache analysis, abstract interpretation.

ACM Reference Format:
Shuo Xu, Wei Zhang, Mengying Zhao, Zimeng Zhou, and Lei Ju. 2024. Cache-aware Task Decomposition for Efficient Intermittent Computing Systems. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657382

1 INTRODUCTION

As the deployment of IoT devices continues to grow, it is expected that the number of such devices will exceed 20 billions by 2025 [2]. This presents a significant challenge in terms of powering such a vast number of IoT devices. Energy harvesting, which harvests the energy from the ambient environment, has emerged as a promising solution due to its ability to eliminate the need for frequent battery replacements or recharging. However, since harvested energy is generally weak and unstable, the system may experience very frequent and unpredictable power failures, leading to non-progress problem—the system may repeat to reboot from the very beginning[9].

Intermittent computing has been proposed as a solution to address the non-progress problem by decomposing tasks into smaller program segments [8, 14]. When resuming execution after a power failure, the system starts from the beginning of the most recently executed task prior to the power failure and gradually progresses towards completion. To prevent the loss of system state during power failures, commercial off-the-shelf (COTS) MCUs often adopt non-volatile memory (NVM), such as FRAM [1], as the main memory. However, directly operating data on NVM can lead to memory inconsistency due to the write-after-read (WAR) problem [15], which occurs in the presence of power failures (detailed in Section 2.1). To solve the problem, the system needs to back up the system state at the beginning of each program segment. Furthermore, utilizing NVM as the main memory introduces a longer main memory access latency.

The write-back cache is extensively adopted in modern processors to bridge the speed gap between the fast processor and the slow main memory. However, this cache architecture introduces challenges when it comes to system state backup. Since the most recent data may reside in the cache instead of the main memory, directly backing up the system state solely on the main memory can lead to an incorrect backup. One possible solution to solve the

aforementioned problem is to adopt a write through cache, which leads to a long memory access latency on every write reference [7], resulting in potential performance degradation. Some existing methods enforce a cache flushing before backups. However, this approach prolongs the backup latency, impacting system performance. While some research work [7, 20, 22] has proposed new cache architectures and replacement policies to tackle this problem, our focus in this paper is on the use of realistic COTS cache architecture, specifically the write-back volatile cache with the Least Recently Used (LRU) replacement policy.

While the cache presents challenges to system state backup due to its buffering of the most up-to-date data, it also provides an opportunity to preserve a recoverable system state in memory. If we assume that no dirty cache lines are evicted during the execution of a program segment, it becomes safe to resume execution from the beginning of that program segment using the system state reserved in the main memory. Based on this observation, we in this paper propose a cache-aware task decomposition aimed to decompose a program into several program segments (i.e., task), where zero dirty cache line evictions occur. By doing so, when suffering a power failure, the system can seamlessly resume the execution from the beginning of the task without the need for system state restoration.

However, identifying dirty cache line eviction at compile-time is challenged as the cache behavior of each memory reference is path-dependent. Exhaustively enumerating all program paths to identify such evictions is computationally intractable. We in this paper adopt the abstract-interpretation method based cache behavior analysis [3] to efficiently identify the memory references that may evict a dirty cache line and then decompose the task accordingly. At task boundaries, we manually flush dirty cache lines to create a recoverable system state for the succeeding tasks. Moreover, since the main memory remains unchanged during the execution of each sub-task, we can parallize the task execution with the system state back up. This enables the task execution in our proposed system to only wait for a lightweight cache flush, reducing execution delays compared to existing methods where task execution must wait for both cache flush and backup completion. Experimental results using both intermittent and constant power supplies demonstrate the effectiveness of our proposed system.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the intermittent computing system in section 2.1, and then show the motivation of the proposed method in section 2.2

2.1 Intermittent Computing Systems

Energy harvesting systems generally adopt a capacitor to buffer the harvested energy and operate intermittently—operate when the capacitor is fully charged, and stop when the energy is depleted [6, 18]. However, frequent power failures can cause a long-running program to enter an endless cycle of system reboots, hindering progress. Intermittent computing decomposes the program into small tasks that require less energy than what can be buffered in the capacitor and resumes the execution from the most recently executed task, so that the program can be finished incrementally across power failures [8, 14]. As shown in Fig. 1 (a), assuming the

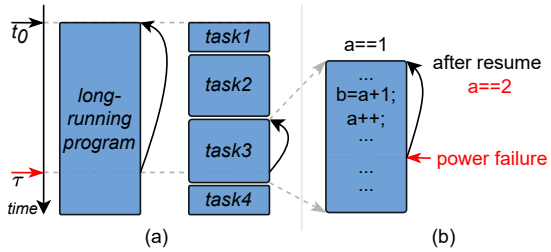


Figure 1: Task decomposition in intermittent computing systems (a) and the WAR problem (b).

system experiences power failures every τ, the long-running program will repeatedly reboot to the beginning and can not forward progress. By decomposing the long-running program into small tasks, each task can be finished in τ, and thus the long-running program can be finished.

To ensure a consistent system state, intermittent computing backs up the system state at the beginning of each task. Most existing ultra-low power MCUs adopt the non-volatile memory (e.g., FRAM) as the main memory [1], which may result inconsistent system state before and after power failures.Consider the example depicted in Fig. 1 (b), where we assume the initial value of variable a is 1 at the beginning of task 3. During execution, a is first read and then written to 2. If a power failure occurs at this point and the system resumes execution from the beginning of task3, the value of a will be read as 2, which is incorrect. To address this issue, intermittent computing systems backup the write-after-read (WAR) variables at the start of each task. When the system resumes the execution from power failures, the backup system state is recovered to ensure a correct execution [16, 17, 21].

2.2 Motivation

In intermittent computing systems, ensuring forward progress in the face of frequent power failures often requires decomposing tasks into smaller tasks. However, this approach introduces a significant backup overhead, which can account for up to 90% of the overall execution time [13, 19]. This backup overhead negatively impacts system performance, and how to reduce it is a crucial design issue in intermittent computing systems. Moreover, the use of NVM, while providing persistent storage, can impact the overall speed of program execution due to the increased time required for memory access.

The write-back cache is extensively adopted in modern processors to bridge the speed gap between the processor and the main memory. However, it introduces challenges in maintaining consistent backups. For the example shown in Fig. 2, let's assume that both variables a and b have an initial value of 1. During the execution of the task, a and b are both updated to 2 in the cache. However, the values stored in the main memory remain as 1. If we back up the system state from the main memory, the backed-up system state will be incorrect. To address the issue of inconsistent backups, one possible solution is to adopt a write-through cache approach. In this approach, every write operation is immediately propagated to the main memory, ensuring consistency between the cache and the main memory. However, this approach incurs a longer main

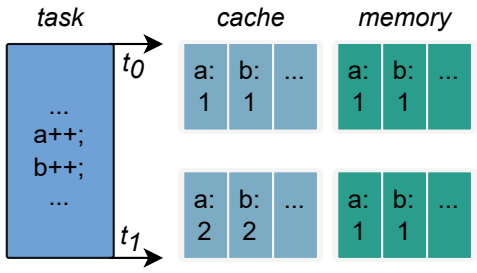


Figure 2: Cache leads to inconsistent backup states and unchanged main memory.

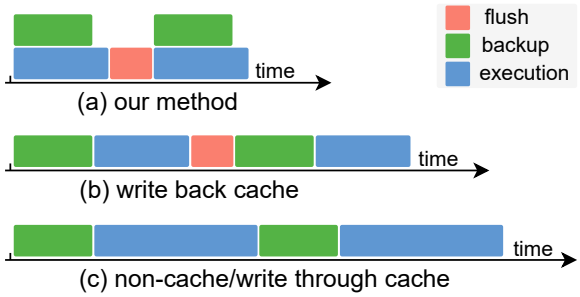


Figure 3: Intermittent computing systems with different backup policies.

memory write latency, as illustrated in Fig. 3 (c). Alternatively, some approaches [7] enforce a cache flush before performing backups, which can increase the backup overhead, as shown in Fig. 3 (b).

Indeed, the write-back cache introduces challenges in maintaining consistent backups. However, it also provides an advantage by preserving an unchanged main memory state, which enables correct system resumption from power failures. In the example depicted in Fig. 2, let's assume that the main memory states at time $t0$ and $t1$ are same. If a power failure occurs during the execution of the task, the system can safely resume execution from the beginning of the task without recovering the system state.

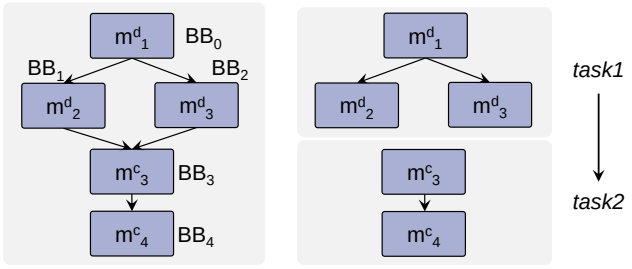
In this paper, we propose a cache-aware task decomposition technique that takes advantage of the cache's ability to delay updates to the main memory. The objective is to decompose a program into a set of tasks in such a way that no dirty cache lines are evicted during their execution. By ensuring that memory remain unchanged during task execution, the system can seamlessly resume execution from the beginning of each task. Furthermore, we introduce a parallel backup mechanism to ensure the correctness of the backup state, while mitigating backup latency. This mechanism significantly reduces the backup latency, as illustrated in Fig. 3 (a). As a result, the efficiency of intermittent computing systems is significantly improved.

3 DESIGN

We in this section first present the cache-aware task decomposition in section 3.1, and then show the run-time system in section 3.2.

3.1 Cache-aware task decomposition

In this paper, we propose a cache-aware task decomposition technique that focuses on preserving a recoverable system state on the main memory by preventing the eviction of dirty cache lines during the execution of each task. The cache behavior of a memory reference is influenced by the specific program path taken, leading to different cache behaviors across various program paths. Consequently, identifying the memory references that could potentially lead to the eviction of dirty cache lines typically needs exhaustive enumeration of all program paths. To avoid the exhaustive path enumeration, we incorporate abstract interpretation-based cache behavior analysis [3] into our approach to efficiently identify memory references that have the potential to evict dirty cache lines.



(a) CFG of the analyzed program (b) cache-aware task decomposition

BB	IN	OUT
BB_0	{⊥, ⊥,⊥}	{ m^d_1,⊥,⊥}
BB_1	{ m^d_1,⊥,⊥}	*update*(gen_1,IN_1)= { m^d_2,m^d_1,⊥}
BB_2	{ m^d_1,⊥,⊥}	{ m^d_3,m^d_1,⊥}
BB_3	*join*(OUT_1,OUT_2)= { $m^d_2$$m^c_3$,$m^d_1$,⊥}	{m^d_3,m^d_2,m^d_1}
BB_4	{m^c_3,m^c_2,⊥}	{m^c_4,m^c_3,m^c_2}

(c) abstract cache state computation

Figure 4: The cache-aware task decomposition. We assume that all the memory blocks map to the same cache set, and the associativity is 2.

The cache behavior analysis is performed on the control flow graph (CFG), which provides a graphical representation of a program's control flow as depicted in Fig. 4 (a). In the CFG, each node represented by BB_i corresponds to a basic block consisting of a sequence of consecutive assembly instructions without branches. The edges in the CFG illustrate the flow of control between the basic blocks based on conditional and unconditional branches. Within the CFG, we identify all write and read data memory references. If a memory block is accessed by a write reference, it is marked as dirty denoted by m^d; otherwise, it is marked as clean denoted by m^c. We in this paper consider the widely adopted set-associative separated L1 data and instruction cache. In the following we only discuss the analysis on each cache set, as the analysis for different cache sets are independent. To represent the abstract cache state, we define $\hat{s}$ as:

$$\hat{s} = l_1, l_1, l_2, ..., l_n$$

to capture the upper bound of the positions (the relative ages) of the memory blocks that can possibly reside in the corresponding concrete cache set, where n denotes the associativity, l_i contains memory blocks that has age i. If a memory is reside in l_n, it indicates that it has been evicted from the cache.

To detect whether dirty cache lines are evict are not during the execution of each basic block, we define two abstract cache states IN_i and OUT_i to represent the abstract cache state before and after executing BB_i, respectively. If a dirty memory block is present in IN_i but not in OUT_i, the dirty memory block is considered to be evicted from the cache. The computation of these abstract cache states is performed using the equations *update* and *join*:

$$OUT_i = update(IN_i, gen_i)$$

$$IN_i = \bigcup_{BB_j \in Pred(BB_i)} join(OUT_j)$$

where $Pred(BB_i)$ returns all the preceding basic blocks of BB_i, and *update* returns the abstract cache states after executing BB_i based on the initial cache state IN_i. For example, when computing OUT_1, since m_2 is accessed in BB_1, it is involved in OUT_1 with age 0, while the age of m_1 is updated to 1.

The *join* operation combines multiple abstract cache states via different input paths into a single abstract cache state to produce a conservative approximation of the overall cache state. This approximation ensures the identification of all possible evicted dirty cache lines, guaranteeing the soundness of our method. Specifically, when a memory block exists in different OUT_j states, *join* retains only the instance with the maximum age. This means that the oldest version of the memory block is considered in the abstract cache state. Additionally, if a memory block exists in different OUT_j states and at least one of those instances is marked as dirty, the reserved memory block is always considered dirty. This ensures that if any version of the memory block is marked as dirty, the abstract cache state reflects this dirty status. For example, consider m_3 which is marked as dirty in IN_3, but it is also accessed by a read reference in BB_3. When computing OUT_3, only the dirty state of m_3 is reserved in order to maintain soundness.

In our method, we initialize the *IN* of the entry basic block as empty and then traverse the CFG to iterativaly compute the *OUT* and *IN* of each basic block. until a fixed point is reached (the *OUT* and *IN* values of all the basic blocks remain the same as in the previous iteration). This iterative computation is illustrated in Fig. 4 (c). During the computation, if the age of a dirty memory block grows to n, we consider it as an indication that a dirty cache line has been evicted. In such cases, we identify the preceding basic blocks as stask boundaries, and the corresponding basic block becomes the entry basic block of a new task. For example, during the computation of OUT_3, the age of m_1^d reaches 2. We consider it to be evicted from the cache. As a result, we decompose the task at the beginning of BB_3, treating BB_1 and BB_2 as the task boundary of the current task (i.e., task1), while BB_3 becomes the head of a new task (i.e., task2).

During run-time, we implement a mechanism to flush all the dirty cache lines to the main memory at task boundaries. This ensures the creation of a recoverable system state in the main memory. Consequently, when reaching the entry basic block of a sub-task, all the cache lines are guaranteed to be clean. As a result, we can safely consider all the memory blocks in the *IN* state of each entry basic block as clean. This allows us to continue the computation of the abstract cache states. Referring to Fig. 4 (c), m_3^d is initially dirty until reaching BB_3 (i.e., OUT_3). However, since BB_3 becomes a new entry basic block, m_3^d transitions to a clean state (i.e., IN_4). By iteratively performing this process, we can identify all the task boundaries and task heads. Additionally, we can group the instructions between the task head and its corresponding task boundary to define the sub-tasks. In cases where the energy required by a task exceeds the energy buffering capacity of the device, we further decompose the task into several smaller tasks based on the energy buffer's capacity to prevent tasks from never completing [9, 12].

3.2 Run-time system

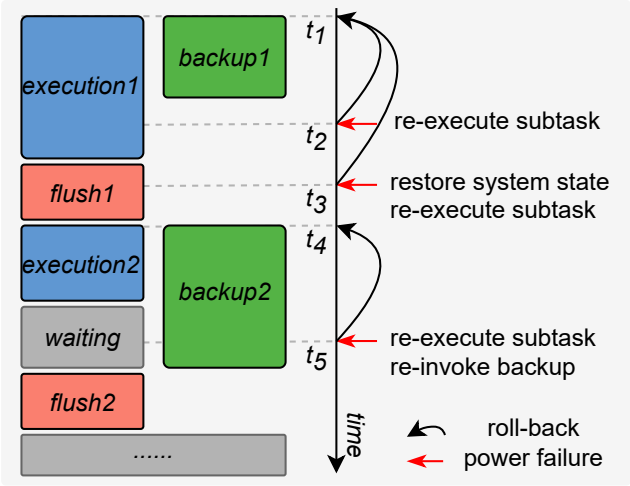


Figure 5: The execution flow of the proposed system.

In this section, we present the workflow of the proposed system. The system utilizes a task decomposition method that maintains a recoverable system state in the main memory and flushes the cache to create a new recoverable system state for the subsequent task. However, there is a potential issue in the system if a power failure occurs during the cache flush process. In such a scenario, the system state stored in the main memory may become corrupted, leading to an inconsistent system state. More specifically, during a power failure, only a portion of the newest system state is successfully flushed to the main memory, while the remaining portion is lost. This incomplete flushing results in an inconsistent system state stored in the main memory.

To address this issue, we create a system state buffer in the main memory to store a copy of the system state. If a power failure occurs during cache flushing, we recover the system state from the buffer and resume execution from the beginning of the most recently executed task prior to the power failure. As a result, each task in the system consists of three phases: execution, backup, and flush. However, since the proposed cache-aware decomposition guarantees that no dirty cache lines are evicted during the execution

of each task, we can parallelize the backup stage with the execution stage. This parallelization significantly reduces the backup latency and improves overall performance as illustrated in Fig. 3 (a). Once both the execution and backup stages are completed, the system proceeds to the flush stage, which creates a new recoverable state for the subsequent task. It is important to note that after cache flushing, all the dirty cache lines remain in the cache, but their states are changed to clean, which creates more space in the cache to accommodate new dirty cache lines in subsequent tasks.

The execution flow of the run-time system is illustrated in Fig. 5. Initially, the execution stage and the backup stage are performed in parallel. If the harvested energy is depleted during either of these stages (e.g., at t_2), the system resumes the execution from the beginning of the task once sufficient energy is available. The flush stage is initiated when both the execution stage and the backup stage have been completed (e.g., at *flush1* and *flush2*). During this stage, the system flushes cache and creates a new recoverable system state for next task. Otherwise, the system can simply resume the execution from the from the latest executed task prior to the power failure. During run-time, we utilize a flag on the Non-Volatile Memory (NVM) to indicate whether the system is in the flush stage or not. This flag helps us determine whether a power failure occurred during the flush stage or at another point in the execution. If a power failure occurs during the flush stage, the system first restores the system state from the buffer and then resumes the execution from the latest executed task prior to the power failure, ensuring a consistent state. Otherwise, the system can simply resume the execution from the latest executed task prior to the power failure.

4 EVALUATION

This section evaluates the proposed method, where the experimental setting is shown in section 4.1 and the experimental results are shown in section 4.2.

4.1 Experimental Setting

We perform our experiments using the gem5 [4], which simulates a single-core in-order ARM processor with a 16KB L1 D-cache and 16KB L1 I-cache. The CPU frequency is set to 16 MHz, which is same with that of the commonly used processors in intermittent computing systems such as the MSP430 [1]. In order to demonstrate the applicability of the proposed method on the realistic COTS cache architecture, we set up the I-cache and D-cache with 2-way associativity and a cache line size of 64 bytes, using the LRU (Least Recently Used) replacement policy. In our configuration, the main memory is implemented based on the specifications of FeRAM, a nonvolatile random access memory [10]. The memory access latency of FeRAM is approximately 12 times that of a cache hit. Benchmarks are selected from Mibench[11], which are representative applications in embedded systems. All benchmark programs are decomposed into small tasks based on the proposed cache-aware task decomposition method and compiled with the default O3 flag.

We evaluate our method by comparing it with an intermittent computing system that utilizes a write-through cache (referred to as WT), as well as a method that employs a write-back cache but sequentially performs backup with cache flushing (referred to as WB).

4.2 Experimental Results

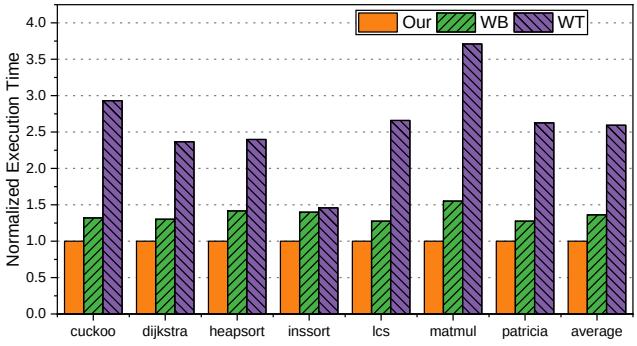


Figure 6: Normalized execution time of different benchmarks in different intermittent computing systems under a constant power supply.

We frist evaluate our method in a non-power failure scenario to investigate the system performance improvement of the proposed system. Experimental results are shown in Fig. 6, where all execution times are normalized relative to the WB-P method. Comparing our method with WT, we observed a significant average reduction of 62% in execution time. This is because that each write reference in a write-through cache experiences a long main memory access latency, leading to longer program execution times. For instance, the *matmul* benchmark, which involves extensive memory writes while computing the multiplication of two matrices, achieved a 71% reduction in execution time with our approach. Even for benchmarks like *inssort* which has less write references, our proposed method still achieved a 29% reduction in execution time compared to WT.

When comparing our method with WB, we observed an average performance improvement of 27% with different benchmark programs. The improvement varies from 22% to 36% depending on the ratio between the task execution time and the backup time with different benchmark program. For instance, in the case of the *matmul* benchmark, the average backup time is 1.3 times the average task execution time, which is very close. Since our method parallelize the backup with task execution, we achieve a significant reduction of 36% in execution time for *matmul*. These results highlight the benefits of our method's parallel backup approach, as it effectively reduces execution times by leveraging the parallelism between task execution and backup operations.

We also evaluate the proposed method in the context of intermittent power supplies, where power failures occur periodically with a specified power cycle duration. We conduct experiments using three different power cycles: 1ms, 5ms, and 10ms, which are typical ranges for power cycles in intermittent systems powered by capacitors [5]. The experimental results show the similar trend to those observed under a constant power supply. Our proposed method consistently outperforms both WB approach and the WT approach. However, due to space limitations, we only present the breakdown

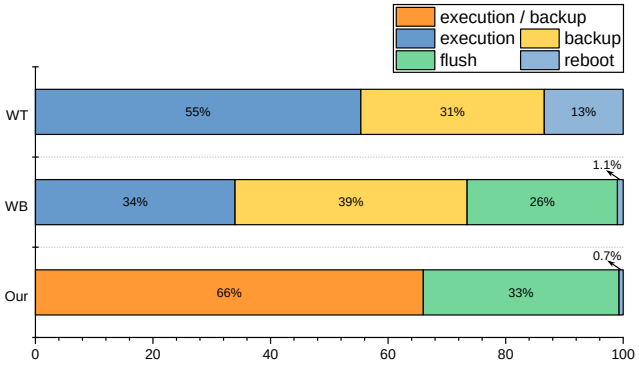


Figure 7: Time breakdown for all benchmark programs in different systems under *1ms* power cycle.

of the execution time for all benchmark programs under different approaches in Fig. 7 with a power cycle of *1ms*. The time breakdown analysis demonstrates that our method achieves the shortest overall execution time while also allocating the largest proportion of time to task execution. This indicates the efficiency of our approach in effectively utilizing the limited harvested energy. Furthermore, the experimental results indicate that both our method and the WB approach experience less reboot overhead, which refers to the time spent on recovering the system state after a power failure. In contrast, the WT approach exhibits longer task execution times, leading to more frequent power failures. This highlights the crucial role of adopting a write-back cache in intermittent computing systems to minimize system reboots and enhance overall performance.

5 CONCLUSION

Intermittent computing systems periodically save the system state to non-volatile memory (NVM) in order to ensure progress during power failures. However, the presence of cache, which buffers the latest system state, introduces additional complexities in system state backup. Existing methods enforce a cache flush before backup to maintain a consistent backup system state, leading to a prolonged backup latency. We emphasize that while a cache may leave a potentially outdated system state in the main memory, it also presents an opportunity to remain an unchanged main memory state which can be used to resume the execution from power failures. Building on this observation, we propose a cache-aware task decomposition method that divides a program into small tasks that do not evict any dirty cache lines during their execution. Furthermore, since the main memory remains unchanged during task execution, we can parallelize task execution with the system state backup process to hide the backup latency. Experimental results demonstrate that the propose system outperforms other systems across different power supply conditions.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (Grant No.62302270, 62102230),Shandong Provincial Natural Science Foundation (Grant No.ZR20220F003, ZR2021019),Taishan Scholars Program (No.tsqn202211281), and Qilu Young Scholar Program of Shandong University.

REFERENCES

[1] 2023. MSP430 microcontrollers. http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html.
[2] Sep. 2022. iot-analytics. https://iot-analytics.com/.
[3] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache behavior prediction by abstract interpretation. In *International Static Analysis Symposium*. Springer, 52–66.
[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.
[5] J. Choi, H. Joe, Y. Kim, and C. Jung. 2019. Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[6] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 399–412.
[7] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2023. Write-Light Cache for Energy Harvesting Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.
[8] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 514–530.
[9] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*. 116–127.
[10] William Goh, Andreas Dannenberg, and Johnson He. 2014. MSP430 FRAM technology–how to and best practices. *Technical report, Texas Instruments* (2014).
[11] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 3–14.
[12] Maeng Kiwan and Lucia Brandon. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the 40th ACM Conference on Programming Language Design and Implementation, PLDI*.
[13] Songran Liu, Wei Zhang, Mingsong Lv, Qiulin Chen, and Nan Guan. 2020. LATICS: A Low-Overhead Adaptive Task-Based Intermittent Computing System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3711–3723.
[14] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM Conference on Programming Language Design and Implementation, PLDI*.
[15] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.
[16] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 129–144.
[17] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks (TOSN)* 16, 1 (2020), 1–24.
[18] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 17–32.
[19] Kasim Sinan Yildirim, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah D. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems, SenSys*. 41–53.
[20] Jianping Zeng, Jungi Jeong, and Changhee Jung. 2023. Persistent processor architecture. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1075–1091.
[21] Wei Zhang, Qianling Zhang, Mingsong Lv, Songran Liu, Zimeng Zhou, Qiulin Chen, Nan Guan, and Lei Ju. 2022. Adaptive Task-based Intermittent Computing System with Parallel State Backup. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).
[22] Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, and Changhee Jung. 2023. Sweepcache: Intermittence-aware cache on the cheap. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1059–1074.