Intermittent Computing with Efficient State Backup by Asynchronous DMA

Wei Zhang[1], Sonran Liu[2,1], Mingsong Lv[1,*], Qiulin Chen[3], and Nan Guan[1]
[1]The Hong Kong Polytechnic University, Hong Kong
[2]Northeastern University, China
[3]Huawei Technologies Co., Ltd., China

***Abstract*—Energy harvesting promises to power billions of Internet-of-Things devices without being restricted by battery life. The energy output of harvesters is typically weak and highly unstable, so computing systems must frequently back up program states into non-volatile memory to ensure a program will progress in the presence of frequent power failures. However, state backup is a time-consuming process. In existing solutions for this problem, state backup is conducted sequentially with program execution, which considerably impact system performance. This paper proposes techniques to parallelize state backup and program execution with asynchronous DMA. The challenge is that program states can be incorrectly backed up, which may further cause the program to deliver incorrect computation. Our main idea is to allow errors to occur in parallel state backup and program execution, and detect the errors at the end of the state backup. Moreover, we propose a technique that allows the system to tolerate backup errors during execution without harming logical correctness. We designed a run-time system to implement the proposed approach. Experimental results on an STM32F7-based platform show that execution performance can be considerably improved by parallelizing state backup and program execution.**

***Index Terms*—intermittent computing, asynchronous DMA, state backup, execution performance**

I. Introduction

It is predicted that the number of Internet-of-Things (IoT) devices will exceed 20 billion by 2025 [1]. A challenge is how to power such a huge amount of IoT devices. As such devices are typically deployed in complex working environments, it is almost impossible to charge or change their batteries. Energy harvesting is a promising approach which allows a device to rely on energy harvested from the ambient environment. As energy output of harvesters is typically weak and unstable, the computing system must ensure software programs will make progress in the presence of frequent power failures.

To this end, a new computing paradigm called intermittent computing is proposed [2]. In intermittent computing systems, a program progresses incrementally with the granularity of program segment. At the end of each program segment, the program states are backed up to non-volatile memory (NVM), such as FLASH or FRAM, and then the system continues to execute the next program segment. Once a power failure occurs and later the system recovers, the backup states on NVM are reloaded so that the execution can resume from the beginning of the failed segment, instead of conducing a complete restart from the very beginning of the program. State backup is very time-consuming and is frequently conducted during the progress of a program. Although Direct Memory Access (DMA) is used in most systems to accelerate data copying, the time overhead of state backup is still very large.

To the best of our knowledge, all existing intermittent systems, with documented method on state backup, conduct state backup sequentially with program execution, i.e., during state backup the program has to stop and wait. However, if state backup can be conducted in parallel with program execution, then the state backup latency will be hidden in program execution. As a result, the program has a much smaller total execution time, and thus can make better progress.

In this work, we seek to parallelize state backup and program execution by asynchronous DMA. The main challenge is data racing may cause incorrect program states that are polluted by program execution to be backed up into NVM. In our solution, we parallel state backup with program execution, and at the end of the state backup detect whether an error has occurred. Moreover, even if an error is detected, we do not immediately handle the error, but let the program continue to execute, as it is highly probable that the erroneous backup will be covered by a future correct backup. We have implemented an intermittent system based on the proposed approach. Experiments conducted on an STM32F7-based platform show that the proposed method can efficiently detect and cover state backup errors, and by parallelizing state backup with program execution, system performance is considerably improved.

II. Related Work

Intermittent computing [3], [4] is recently proposed to enable a system to keep progress and produce correct computation in the presence of frequent power failures. The main principle is to let the system timely back up program states to NVM so that when the system recovers from a power failure, it can resume from the most recent backup states instead of experiencing a complete restart. With the increasing number of energy harvesting IoT devices [1], intermittent computing is an enabler for such devices to provide sustainable services.

This work was supported by Huawei Innovation Research Program "Collaborative Research Project on Computational Sensing", the Research Grants Council of Hong Kong (GRF 15204917 and 15213818), and Natural Science Foundation of China (grant No. 61772123 and 61672140).

*Corresponding author: Mingsong Lv. Email: mingsong.lyu@polyu.edu.hk

TABLE I: A survey of state backup methods

Methods	Intermittent systems
CPU	Cotai [7], Chain [8], Mementos [9], Alpaca [10], QUICKRECALL [11], Ratchet [12], Chinchilla [13]
DMA	InK [14], Daulby [15], Chen [16], TICS [17], Coala [18], CoSpec [5], ELASTIN [19], CatNap [20]
Unknown	Hibernus++ [6], HarvOS [21]

As state backup is a major overhead in intermittent computing, we did a survey of existing intermittent systems on their state backup methods. A classification of state backup methods is given in Table I (For intermittent systems that are not named, we use the surname of the first author to represent the proposed system or method). All systems with documented state backup method adopt either CPU or DMA to conduct state backup. First, in all systems that use CPU, state backup is sequentially conducted with program execution. Second, systems that use DMA also conduct state backup sequentially with program execution. In CoSpec [5], a specialized hardware, which is not widely available in embedded MCUs, is required in state backup. This paper works towards reducing state backup overhead by parallelizing state backup and program execution. Note that in intermittent systems that adopt Just-in-Time checkpointing [6], system execution is firstly stopped and then the system states are saved to NVM, so state backup is always serially conducted with system execution.

III. The Parallel State Backup Problem

This section shows the correctness problem caused by parallel state backup and program execution. In sequential state backup using either CPU or DMA, a program segment is allowed to start only after the program states at its starting point are totally copied to NVM. Now consider to conduct state backup simultaneously with the execution of a program segment. We will show by the example in Fig. 1 that a critical correctness problem may occur.

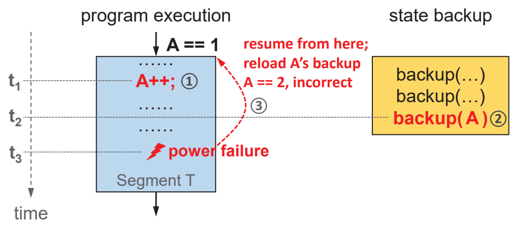


Fig. 1: An example of the parallel state backup problem

Assume A is one of the variables to back up. The execution of program segment T operates A during execution, and state backup copies state variables including A to NVM. A correct state backup at the beginning of T should copy A's value 1 to NVM. The execution of T modifies A's value to 2 by an increment operation at time t_1, while the backup of A is conducted at a later time t_2. As a result, the NVM stores an incorrect value of A (i.e., 2). If a power failure occurs at time t_3, after the system recovers, the program resumes from the beginning of T by reloading the backup states from NVM. Now an incorrect value of A is reloaded. We call this phenomenon the "*parallel state backup problem*". The problem will further result in incorrect computation of T.

Due to the lack of techniques to solve the parallel backup problem exemplified by Fig. 1, to the best of our knowledge, existing intermittent computing systems pessimistically serialize task execution and state backup (by either CPU or DMA). The consequence is that a large portion of the system execution time is spent on frequent state backup [22].

IV. Overview of Our Approach

The objective of this work is to parallelize state backup and program execution for intermittent systems without destroying logical correctness. Such a design allows to hide state backup latency in task execution time, and thus can improve system performance. In this section, we will give an overview of the main idea. Key design details are provided in the next section.

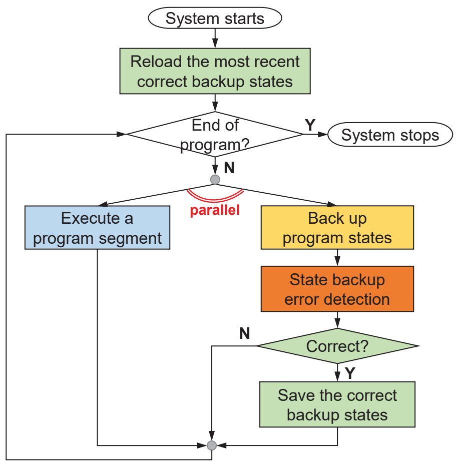

System starts
Reload the most recent correct backup states
End of program?
Y
System stops
N
parallel
Execute a program segment
Back up program states
State backup error detection
Correct?
N
Y
Save the correct backup states

Fig. 2: Overview of the proposed state backup approach

The main idea is carried by the work flow shown in Fig. 2. The system behavior mainly contains a loop to execute program segments one after another until the program is finished. In our approach, we conduct state backup simultaneously with the execution of a program segment. As this may cause the problem shown in Fig. 1, we propose a technique to detect potential backup errors at the end of the state backup (The error detection technique will be detailed in Sec. V).

Once a state backup error is detected, we know the current backup data is not trustworthy and can not be used for system restoration. A common method is to completely restart the system. However, a complete restart will incur too much re-execution overhead, so we try to avoid it. Note that state backup is performed timely and frequently in intermittent systems. It is highly probable that an incorrect backup can be overwritten by a future correct backup. Motivated by this observation, we propose *fault-tolerant backup management* to allow the system to continue execution even if the backup states are temporarily incorrect. We will explain the idea with the example in Fig. 3.

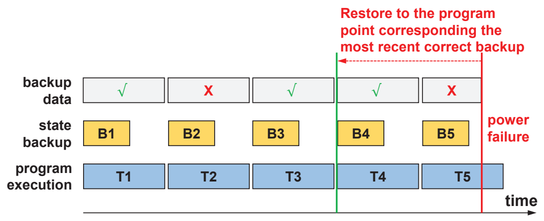


Fig. 3: Fault-tolerant backup management

In Fig. 3, T_i represents program segments, and B_i represents the state backup conducted in parallel with T_i. When backup B_2 is done, a backup error is detected, leading the backup data to an inconsistent state. However, we do not immediately solve the problem, and simply allow the system to continue executing T_3. As the next state backup B_3 is correct, the incorrect backup states are covered by new correct backup states that can be used for system restoration. Note that state backup B_5 is also incorrect, and unfortunately during the execution of T_5, a power failure occurs. As the backup states now are incorrect, the system is not able to restore to the beginning of T_5. Since backup B_4 is correct, the system can roll back to the beginning of T_4, i.e., the program point corresponding the most recent correct backup. The fault-tolerant backup management, as part of the whole solution, is depicted by the green blocks in Fig. 2.

The next section will present the detailed design for state backup error detection and fault-tolerant backup management.

V. Design

In this section, we first introduce the task-based software model [14] which is used to explain the proposed techniques Then state backup error detection is presented in detail. At last, we present a heuristic to explore how memory layout can affect the occurrence of state backup errors.

A. Software Model

By task-based model, a program is coded as a collection of *tasks*, where each task is essentially a program segment implemented as a function. The tasks are connected by the control flows specified by the programmer. During the execution, state backup is conducted at the beginning of each task. If a power failure occurs, and later the system recovers, it first restores the program states from the backup, and then continues to execute the last unfinished task before power failure. Program states in task-based model refer to the set of variables that are shared by multiple tasks and whose lifetimes span across task boundaries, but not the local variables allocated on the stack and accessed inside the task. To distinguish data copies, we say task execution accesses task-shared variables stored in a *working buffer* in main memory, and in state backup, task-shared variables are copied from the working buffer to a *backup buffer* on NVM using DMA.

B. State Backup Error Detection

1) Target for Error Detection:

In the general sense, any task-shared variable in the working buffer may experience the problem in Fig. 1, if state backup is conducted in parallel with task execution. However, not all backup errors will eventually lead to incorrect program execution. Error detection only needs to observe those backup errors that affect the correctness of task execution.

Whether the backup error of a variable will cause incorrect execution depends on the access behavior of the variable. Without loss of generality, we consider a task-shared variable A accessed in task T. The behaviors of accessing A can be classified into the following cases which have different results on the correctness of task execution.

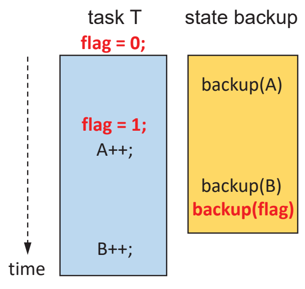
- *CASE 1: A is only read in* T. No backup error will occur as the execution of T does not change A's value.
- *CASE 2: A is only written in* T. A write to A may cause incorrect state backup and assume such a case occurs. If power fails in T, the system resumes from the beginning of T. Although an incorrect value of A will be reloaded, the incorrect value will be overwritten by the write operation to A in T, so task execution remains correct. Otherwise, if power failures do not occur, at the end of T, the correct value of A remains in the working buffer. The execution of the next task will execute on the working buffer and the task sees a correct value of A.
- *CASE 3: A is first written and then read in* T. This case is the same as CASE 2.
- *CASE 4: A is first read and then written in* T. T's execution will be incorrect if an incorrect value of A is backed up and power fails in T. After system resumes, the incorrect backup value of A is reloaded from the backup buffer. The first read to A loads the incorrect value, which may cause incorrect computation. Variable A in Fig. 1 is an example of this case.

With the above analysis, an error detection method only needs to observe the backup results for those variables that fall into *CASE 4*. The access properties of variables are evaluated within each task. A variable, evaluated to be in *CASE 4* in any task, will be observed. *In the next sub-sections, when talking about variables, we mean the variables of CASE 4.*

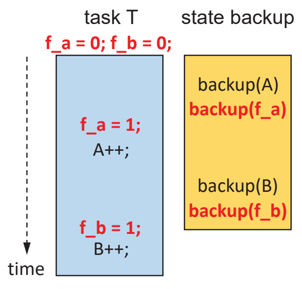
2) Error Detection:

To detect the backup error for a variable, we need to observe the timing relation between the write to the variable by program execution and the read to the variable by state backup. To this end, we instrument both the task code and the backup data with flag variables to record the progress of both CPU and DMA, so as to provide information for error detection at the end of state backup. We will use the examples in Fig. 4 to explain our method.

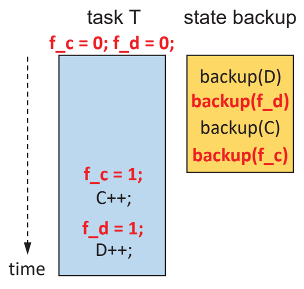
Task T modifies variables A and B during its execution. We need to know whether before such modifications the backup to the two variables is finished or not. We introduce a flag variable "$flag$" and initialize it to 0 before T starts. In task code, we insert a flag instruction $\mathsf{Fl}(flag)$ before the first write to A or B in the task. For example, in Fig. 4(a), $flag = 1$ is a flag instruction. On the state backup side, the variable $flag$ is also backed up, immediately after B. When state backup is finished, we check the value of $flag$ in the backup buffer



(a) Inserting one flag variable to detect error, failed



(b) Inserting two flag variables to detect error, succeeded



(c) Reducing redundant flag variables

Fig. 4: State backup error detection exemplified (principle and optimization)

to detect error. If the value of $flag$ in the backup buffer is 0, it is clear that after both A and B are backed up, none of the modifications to A or B is executed, then we can safely conclude that no backup error occurred. Otherwise, if the value of $flag$ in the backup buffer is 1, it indicates at the time when A and B finish backup, task execution has passed $flag = 1$. In such case, it is possible that the task has executed $A{+}{+}$ or even $B{+}{+}$. Thus, we are not sure whether an error actually occurred during state backup, and have to report an error for safety. In the example in Fig. 4(a), the value of $flag$ in the backup buffer is 1 as $backup(flag)$ executed after $flag = 1$, so a backup error is reported.

Actually, in this example, there is no backup error, since the backup operation to A did occur before the modification to A (by $A{+}{+}$), and so does B. This behavior can not be captured with only one flag variable. Then we introduce two flag variables f_a and f_b to observe the access conflicts on A and B, respectively. The code instrumentation is shown in Fig. 4(b). This time, when we check the values of f_a at the end of state backup, we find that f_a's value in the backup buffer is 0 as $backup(f\_a)$ occurs before $f\_a = 1$, so A is correctly backed up. Similarly, $f_b = 0$, so B is also correctly backed up. Only if the value of *all* flag variables in the backup buffer are 0, we can safely conclude that there is no backup error caused by the execution of the task. By inserting more flag variables in a finer-grained way, we are able to find out correct state backups that would be classified incorrect by a coarser flag insertion.

3) Removing redundant flags:

To maximize the precision of error detection, one can choose to use an independent flag for every variable. However, this will insert redundant flags. We will show what is a redundant flag with the example in Fig. 4(c).

Let us consider two variables C and D observed by two flag variables f_c and f_d respectively. We use the notion $e1 \Rightarrow e2$ to represent event $e1$ occurs earlier than event $e2$. In this example, if at the end of state backup f_c's value in the backup buffer is 0, i.e., C is correctly backed up, we know that $backup(f\_c) \Rightarrow f\_c = 1$. Now we can conclude that D is also correctly backed up, because $backup(f\_d) \Rightarrow backup(f\_c) \Rightarrow f\_c = 1 \Rightarrow f\_d = 1$. Otherwise, if C is not correctly backed up, of course one can still leverage f_d to observe whether D is correctly backed up. Note that we can conclude the state backup is correct only if all flag variables are 0 in the backup buffer. Even if D is found to be correctly backed up, as an error is already detected for C, the detection result for D does not affect the final conclusion. Thus, we can safely remove flag variable f_d without sacrificing the precision of error detection. We say f_d is a redundant flag. The property described above is formulated with Lemma 1. As the proof is straightforward, we omit the proof in the paper.

Lemma 1: Suppose variable A has a higher address than variable B in the buffers, which means $backup(B) \Rightarrow backup(A)$, and in a task the first write to A occurs before that to B, i.e., $write(A) \Rightarrow write(B)$. If the backup of A is correct, the backup of B must be correct.

Assume flag variables and corresponding flag instructions have been inserted to observe all variables. Redundant flag variables that satisfy *Lemma 1* can be found by exploring the flag instructions on the control flow graph (CFG) of a task according to *Definition 1*. Any identified redundant flag variable and its flag instructions will be removed.

Definition 1: A flag instruction $\mathsf{FI}(f)$ is a redundant flag instruction of task T iff: on every path from the starting node of the CFG of T to $\mathsf{FI}(f)$, there is a flag instruction operating a flag whose memory address is no lower than f.

To summarize, our proposed state backup error detection technique works as follows:

1) **Off-line instrumentation**.
   a) Identify task-shared variables that needs observation.
   b) For each such variable, insert a flag into the backup data behind the variable it observes, and insert a corresponding flag instruction into the task before the first write on the variable.
   c) Remove all redundant flags from the backup data and corresponding flag instructions in the program.
2) **Run-time Support.** All the flag variables are set to 0 before each task starts execution. After state backup is finished, if all the flag variables in the backup buffer are 0, the backup is concluded correct; otherwise, the backup is considered incorrect.

C. Buffer Design for Fault-tolerant Backup Management

In Sec. IV we introduced fault-tolerant backup management which allows the system to continue execution even if the backup states are temporarily incorrect. The technique is implemented by the buffer design.

For now, we have a working buffer for program execution and a backup buffer to store backup states. As the backup buffer can be polluted by incorrect state backup, we introduce a new buffer called *safe buffer* to maintain the most recent correct backup copy. When state backup is finished, the correct backup data is in the backup buffer. We do not copy the data from the backup buffer to the safe buffer. Instead, we swap the role of the two buffers. In the implementation, there is a pointer to each buffer. The role swap can be done by a pointer swap. If the backup buffer is polluted by an erroneous backup, the safe buffer still maintains correct program states corresponding an earlier program point. If a power failure occurs, the system will always resume by reloading the states in the safe buffer and roll back to the most recent consistent program point. A power failure may also occur during the data copying from the working buffer to the backup buffer, making the backup buffer inconsistent. For such a case, the safe buffer still enables the system to restore to a most recent consistent program point.

D. Reorganizing Memory Layout to Reduce Backup Errors

In state backup, the variable at a lower address will be copied earlier and thus has a lower probability to experience backup error. So, changing the layout of the variables in the buffers may affect the occurrence of backup errors. However, an optimal memory layout does not generally exist. First, different tasks of a program may access different variables and they compete to put their own variables in low memory address, in the layout optimization process; second, the execution frequency of different tasks is input-dependent and can not be decided offline, so it is impossible to decide which task should have higher priority in competing lower addresses. Therefore, we developed a heuristic method to re-organize memory layout to investigate its impact on backup errors.

The main idea is to put the "most frequently" accessed variables in the lowest memory address. We compute a weight, W_v, for each variable to model its access frequency by equation (1), where v is a variable, and $C_{\mathsf{T}_i}(v)$ is the execution count of task T_i that accesses v. In our exploration, we run each program a sufficiently large number of times to measure the average execution count of each task.

$$W_v = \forall_{v \in \mathsf{CASE}\ 4} \sum C_{\mathsf{T}_i}(v) \tag{1}$$

VI. Experiments and Evaluation

A. Experimental Setup

We designed a task-based intermittent system to implement the proposed approach, and run the system on a STM32F7-based development board. 8 benchmark programs from related work [14], [18] are used for evaluation[1]. The sizes of shared variables in the benchmarks range from 18 bytes to 622 bytes. A programmable power supply is adopted to generate power traces. In the experiments, we assume that the system suffers a power failure around every 5ms.

[1]Available at https://github.com/IntermittentComputing/TaskBased

New NVM devices are increasingly adopted in new processors [23]. They vary in multiple features including most importantly the access speed. To evaluate the performance of our approach for different NVM devices, we use SRAM to simulate NVM at different speeds. This is implemented by evenly inserting dummy variables into the variables to be backed up. We denote the access speed of SRAM by $\mathsf{v_s}$. For instance, to simulate an NVM with half the speed of SRAM, denoted by $\mathsf{v_s}/2$, we insert a dummy variable before each shared variable with the same size. In the experiments, we simulated four NVM speeds, $\mathsf{v_s}/1$, $\mathsf{v_s}/2$, $\mathsf{v_s}/3$ and $\mathsf{v_s}/4$.

B. Results and Evaluation

To evaluate the performance of the proposed approach, and also how it can be affected by different memory layouts, we measure the execution time of the benchmark programs executed in three different settings:

- ASY-OPT: asynchronous DMA on the memory layout produced by the technique in Sec. V-D
- ASY: asynchronous DMA with initial memory layout
- SYN: sequential state backup (by DMA) and program execution with initial memory layout

The results are given in Fig. 5 with all execution times normalized to ASY-OPT. The performance of the proposed approach (ASY-OPT and ASY) is considerably improved compared to existing approaches (SYN). The performance gain is comparably lower in programs such as CRC and CEM. The reason lies in the ratio between the average execution time of a task and the average latency of state backup. Take CEM for example, as the states to backup is very large, the backup latency is much larger than the average task execution time. Parallelizing state backup and program execution does not help much in reducing total execution time. Program CRC is on the other extreme. CRC has a very small state size, and the backup latency is much smaller than the average task execution time. In such a case, the space for asynchronous DMA to improve performance is reduced. The inserted flag variables cause an average increase in the sizes of backup states by 3%.

We analyze the results regarding different NVM speeds. There is not an identical trend for all programs. In essence, different NVM speeds indicate different backup latency, and thus affect the ratio between the average task execution time and the backup latency. If the ratio becomes too small or too large, as discussed before, the space for performance improvement by asynchronous DMA will be small.

At last, we evaluate the impact of memory layout to the proposed approach. In Table II, for each program, we list the average numbers of total backups (**B**), incorrect backups (**IB**) and uncovered incorrect backups (**UIB**) under different NVM speeds, and compare the results between ASY-OPT and ASY. The results show that the proposed heuristic can reduce the number of incorrect backups, especially for AR, CK and CEM.

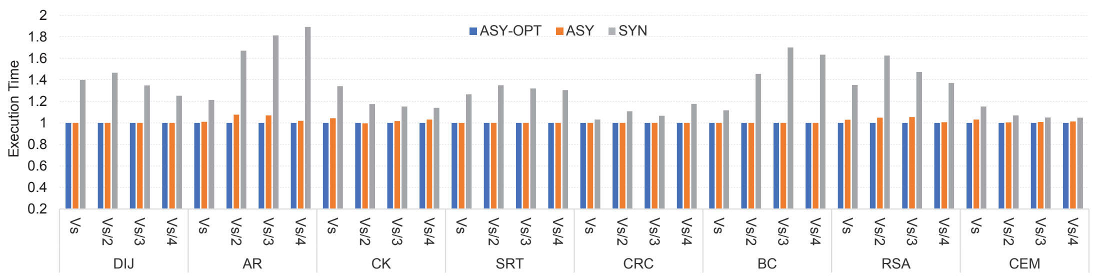


Fig. 5: Execution times of different benchmarks under different NVM speeds (Normalized to ASY-OPT)

TABLE II: Average numbers of uncovered incorrect backups (**UIB**), incorrect backups (**IB**) and total backups (**B**)

		DIJ	AR	CK	SRT	CRC	BC	RSA	CEM
ASY-OPT	UIB	0	0	0	0	0	0	0	0
	IB	0.5	3	24.7	0	0	5.5	8.5	0
	B	167.5	152	260	402	102	87	81	540.5
ASY	UIB	0	0	0.5	0	0	0	0	2.2
	IB	0.5	13.5	49.5	0	0	5.5	10.2	214.2
	B	167.5	152	260.5	402	102	87	81	541.2

If we look at the execution time under ASY-OPT and ASY in Fig. 5, only AR is comparably more sensitive to the memory layout. There are two main reasons. First, even if the better memory layout reduces the number of incorrect state backups, almost all incorrect state backups can be efficiently covered by the proposed fault-tolerant backup management technique regardless of the memory layout (see the UIB row for the two layouts in Table II). Second, in most programs, the state backups are finished before conflicts between state backup and program execution occur, since the write operations to the variables in a task are separated by computation steps.

VII. Conclusion

This paper presents an approach to improve the performance of intermittent systems by enabling parallel state backup and program execution, and at the same time ensure the system still produces correct computation. The main techniques are an error detection method to precisely identify state backup errors and a backup management method that allows the system to tolerate state backup errors at run time. Experimental results show that the proposed approach can considerably improve execution performance compared to the existing approaches with sequential state backup and program execution.

References

[1] https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/, referenced in Sep. 2020.

[2] G. Gobieski, A. Nagi, and et al., "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in *Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2019.

[3] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *Summit on Advances in Programming Languages, SNAPL*, 2017.

[4] K. Ma, Y. Zheng, and et al., "Architecture exploration for ambient energy harvesting nonvolatile processors," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[5] J. Choi, Q. Liu, and C. Jung, "Cospec: Compiler directed speculative intermittent computation," in *Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2019.

[6] D. Balsamo, A. S. Weddell, and et al., "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[7] E. Ruppel and B. Lucia, "Transactional concurrency control for intermittent, energy-harvesting computing systems," in *ACM Conference on Programming Language Design and Implementation, PLDI*, 2019.

[8] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," in *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2016.

[9] B. Ransford, J. Sorber, and K. Fu, "Mementos: system support for long-running computation on rfid-scale devices," in *International conference on Architectural support for programming languages and operating systems, ASPLOS*, 2011.

[10] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent Execution without Checkpoints," in *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2017.

[11] 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 *International Conference on VLSI Design*, 2014.

[12] J. van der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016.

[13] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.

[14] K. S. Yildirim, A. Y. Majid, and et al., "Ink: Reactive kernel for tiny batteryless sensors," in *ACM Conference on Embedded Networked Sensor Systems, SenSys*, 2018.

[15] T. Daulby, A. Savanth, G. Merrett, and A. S. Weddell, "Improving the forward progress of transient systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[16] W.-M. Chen, P.-C. Hsiu, and T.-W. Kuo, "Enabling failure-resilient intermittently-powered systems without runtime checkpointing," in *ACM/IEEE Design Automation Conference, DAC*, 2019.

[17] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2020.

[18] A. Y. Majid, C. D. Donne, and et al., "Dynamic task-based intermittent execution for energy-harvesting devices," *ACM Transactions on Sensor Networks ,TOSN*, 2020.

[19] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving stagnation-free intermittent computation with boundary-free adaptive execution," in *IEEE Real-Time and Embedded Technology and Applications Symposium ,RTAS*, 2019.

[20] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *ACM Conference on Programming Language Design and Implementation, PLDI*, 2020.

[21] N. A. Bhatti and L. Mottola, "Harvos: efficient code instrumentation for transiently-powered embedded sensing," in *ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2017.

[22] J. San Miguel, Ganesan, and et al., "The eh model: early design space exploration of intermittent processor architectures," in *IEEE/ACM International Symposium on Microarchitecture ,MICRO*, 2018.

[23] A. Chen, "A review of emerging non-volatile memory (nvm) technologies and applications," *Solid-State Electronics*, 2016.