REMARK: Reliable and Efficient Multi-device Recovery Framework for Transiently Powered Computers

ABSTRACT

Transiently Powered Computers (TPCs) powered by energy harvesting are becoming popular in today's IoT applications. Although energy harvesting enables the sensor nodes with smaller size and longer lifetime, unstable energy sources frequently break the system execution, posing new design challenges. Existing techniques enable continuous processor executions under intermittent power supply using non-volatile memories and checkpointing techniques. However, few addresses the unique challenges associating checkpoint and recover a syetem with multiple I/O and volatile peripherals devices, which causes major performance degradation. This paper fills this gap by introducing REMARK, a HW/SW co-designed framework enabling reliable and efficient operations in a TPC with multiple peripherals. A REMARK-enabled non-volatile processor chip has been fabricated, based on which a wireless sensor node is implemented to demonstrate reliable and efficient wireless transmission operations under intermittent power supply. Results show that the data transmission efficiency are improved by as much as 13× compared with state-of-the-art. Furthermore, the sensitivity studies on REMARK are also carried out, resulting a program design guideline that improves the performance by as much as 36.5%.

1. INTRODUCTION

Transiently Powered Computers (TPCs) [1, 2, 3] are a new class of battery-less embedded systems that depend solely on energy harvested from ambient environment. TPC systems are especially attractive to many IoT applications since it is often challenging to employ traditional batteries where it is inconvenient, costly or even dangerous to replace or service batteries. Energy harvesting can eliminate the need for batteries or wires and enable long-term operation of these systems with little maintenance. However, design of these systems faces unique challenges since the power supply is intrinsically unstable and affected by the ambient environment conditions [1, 2, 4]. For example, the reliability of solar powered systems largely depends on the weather, location and environmental changes of different applications [5, 6, 7, 8].

Considering the intermittent power supply, how to bridge TPC executions across power failures becomes essential. It is known that a volatile processor will lose its state after power failure, and frequent rollbacks will significantly slow down the system forward progress [1]. Several checkpoint and rollback strategies have been proposed to save the processor state and improve the forward progress. Mementos [3], Hibernus [9] and QuickRecall [10] provide automatic software checkpointing mechanisms for volatile processors (VPs). Meanwhile, non-volatile processor (NVP) [11] and NVIO [12] are proposed to automatically store the processor and I/O state more efficiently based on hardware.

However, these schemes mainly aim at recovering computation tasks on a processor. None of them considers the unique challenges associated with the recovery of multiple peripheral devices. Firstly, peripherals contain analog circuits and interactive operations with environment, making it impossible to backup the device state during peripheral operations. Second, there are a wide variety of volatile peripherals produced by different manufacturers and it is difficult to apply nonvolatile memory to all peripherals. Third, a complete TPC system runs computation tasks on the processor and peripheral operations on peripheral devices simultaneously, where consistency issue arises. Therefore, recovering these peripherals faces more reliability and efficiency challenges. Previous work [13, 14, 15] have noticed the data consistency problem between processor and off-chip NVM, a special case of peripherals. Since peripherals are generally volatile and their states cannot be checkpointed anywhere like a processor, a systemically recovery framework is critical for a multi-device TPC system.

Targeting the recovery problem of TPC with multiple devices, this paper proposes REMARK, a Reliable and Efficient Multi-device Recovery FrAmewoRK, containing novel hardware components on top of NVP and corresponding software optimizations. REMARK enhances the functionality of NVP with four new hardware modules to support efficient multi-device checkpointing. The software optimizations include an offline program transformer and an online recovery mechanism, which enable efficient and reliable resumption of multiple peripherals. The major contributions are listed as follows.

- To the best of authors' knowledge, REMARK is the first work that systematically addresses reliable and efficient peripheral operations in multi-device TPC systems.
- REMARK hides the complexity associated with multi-device recovery by efficient hardware architecture and software/runtime supports. It maintains easy-to-configure interfaces for TPC designer such that they can focus on

the functionality instead of peripheral implementation details.

- We implement a REMARK-enabled NVP chip and the supporting hardware platform. A real application is evaluated on this platform and the results show that REMARK can speed up the data transmission tasks by 13× compared with state-of-the-art solutions.
- We further analyze the impact of different design parameters of REMARK on a sofware simulator, NVnodeSim. Based on these analyses, program design guidelines are proposed to improve the resistance against deadlock and reduce the timing overhead by as much as 36.5%.

The rest of this paper is organized as follows. Sec. 2 illustrates the challenges to design a recovery strategy for TPCs with multiple peripherals. Sec. 3 presents system overview of REMARK, which contains the hardware architecture, the offline multi-device checkpointing strategy and the online recovery mechanism. The details of these three components are presented in Sec. 4, 5, and 6, respectively. Sec. 7 presents the fabrication and the performance of a REMARK-enabled NVP chip. Sec. 8 explores the impact of different design parameters in REMARK with a software simulator and summarizes design rules for TPC systems. . The remaining sections introduce the related works and conclude this paper.

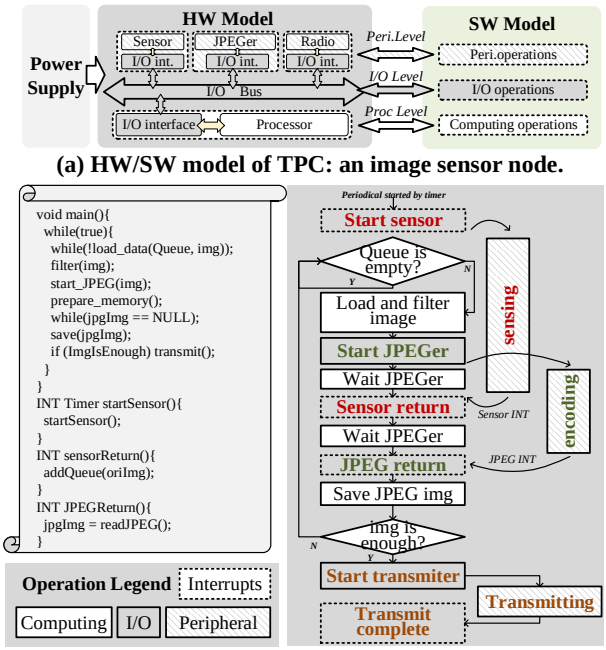
2. MOTIVATION

In this section, we first present the system model of TPCs in Sec. 2.1. Then, we use an image sensing example to show the needs for hardware support in Sec. 2.2 and challenges of recovering a TPC with multiple peripherals in Sec. 2.3.

2.1 TPC System Model

Fig. 1 (a) uses an image sensing example to show the HW/SW models of a multi-device TPC system. The devices contain a processor and multiple peripherals. The sensor node utilizes a processor to execute *computing operations* and controls peripherals. Peripherals, such as sensors and wireless transceiver, are used to perform specific *peripheral operations* in parallel. These peripherals can be divided into three categories. The input-type collects the environmental information, such as the image sensor. The output-type transmits data to other systems, such as the wireless transceiver. The compute-type accelerates computing intensive tasks, such as a JPEG encoder. Processor and the peripherals are mounted to an I/O bus via on-chip I/O interfaces, where I/O operations are executed. With *I/O operations*, the processor initializes, configures and accesses the peripherals by reading and writing their embedded memory.

Fig. 1 (b) shows the program and the work flow of this image sensor node. The image sensor is periodically started by a timer to collect and store images in a queue. In the loop of main function, the system first load and filter an original image from the queue. Then, the JPEGer is called to encode the image. Both the sensor and the JPEGer return their results through hardware interrupts. The processor will finally store the encoded image in memory. When the stored images reach a threshold, the transceiver send these images to the server.



(a) HW/SW model of TPC: an image sensor node.

(b) Program and work flow of image sensor node.

Figure 1: An image sensing example used to present the HW/SW model of TPC system.

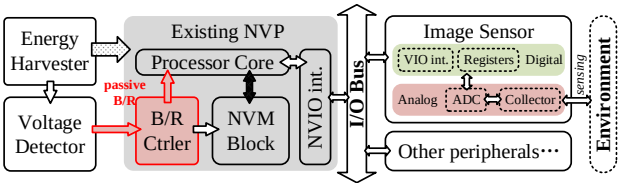


Figure 2: Hardware challenges of multi-device checkpointing. NVP is limited by passive B/R function. Peripherals require efficient reconfigurations and restarts.

In such a system, the processor can be efficiently recovered with existed techniques. However, peripherals will lose their states after power failure, which leads to reliability and efficiency issues. Based on the type of peripherals, different faults will occur if the devices are not recovered: **1. Input-type Failure:** When a sensor loses power, the current collection will be crashed and data is lost. Even worse, one data lost may jeopardize the integrity of the whole data set in a multi-sensor system. **2. Output-type Failure:** When a transceiver loses power, the current transmitting package as well as the data in peripheral buffer will be lost. Although costly software-level re-transmission can be used in some cases, it may not be feasible for resource-constraint energy-harvesting sensors. **3. Compute-type Failure:** When an accelerator loses power, processor cannot receive its returned results and halts all subsequent operations. It may cause system deadlock when program relies on the returned data. Therefore, a systematical peripheral recovery mechanism is in desperate needs.

Targeting on the multi-device TPC, two challenges arise. (a) How to design an efficient and reliable architecture to recover the states of the peripherals in a multi-device system;

(b) How to devise a reliable automatic checkpointing mechanism for a TPC with multiple devices. The rest of this paper addresses these two challenges.

2.2 Hardware Needs for Peripheral Recovery

Fig. 2 shows the NVP [11] achieving efficient recovery by backing up and restoring (B/R) the processor state, when the voltage detector is triggered by a power failure. The B/R operation of processor is passively activated by the voltage detector. However, consistency problems may exist when a power failure occurs during a peripheral operation. This is because peripherals can only back up their states at certain safe positions. Therefore, multi-device TPC requires flexible B/R interfaces instead of the passive approach based on voltage detector in existing NVPs.

Furthermore, an efficient recovery monitor for volatile peripherals besides the NVP is needed. Fig. 2 shows the structure of an image sensor, a typical volatile peripheral. Its digital part uses registers to store the data and device state, and the analog part collects images from environment. When the system recovers a peripheral after power failures, not only the digital part should be restored, but also the analog part. On the other hand, not all peripherals should be recovered in a multi-device TPC especially those are not used. As the number of peripherals increases, the recovery overhead becomes significant. Thus, a peripheral status monitoring module is needed to avoid unnecessary peripheral reconfiguration overhead.

Last but not least, once a peripheral device is determined to be restarted after power failure, e.g. the sensing and data transmission task in the image sensor node, a specific start function should be re-executed. In existing NVPs, those operations are restarted by rolling the entire system back to where the broken peripheral operations just started. Extra rollback overhead takes place because the program cannot return correctly after executing the start program. If a hardware module can simply re-execute the peripheral start program and return correctly, the peripheral restart can be more efficient.

Therefore, to efficiently and reliably recover multi-device TPCs, the enhanced NVP should support flexible checkpointing, peripheral status monitor as well as the peripheral restart modules, which are all missing in existing NVPs.

2.3 Multi-device Checkpointing Challenges

Even with the hardware described in Sec. 2.2, it is still nontrivial to recover multi-device system efficiently and reliably. Fig. 3 shows the recover flows of the image sensor when a power failure occurs. There are three processes in each figure. The bottom is processor process, which executes the main program given in Fig. 1. The middle is sensor process and the top is JPEGer process. These two modules are started by the processor, and return their results via interrupts after completing the sensing and encoding tasks. When power fails and crashed both of the peripherals, traditional single-device approaches [3, 9, 10] roll back the entire system to a homogeneous checkpoint. If *cp1* is selected, all peripherals can be correctly recovered but nontrivial rollback overhead is introduced for both processor and the JPEGer. The processor rolls back for 135ms and the JPEGer has to wait 79ms to restart. If *cp3* is selected, the rollback overhead is smaller but inconsistency arises. The image sensor is not recovered, and the processor receives wrong data.

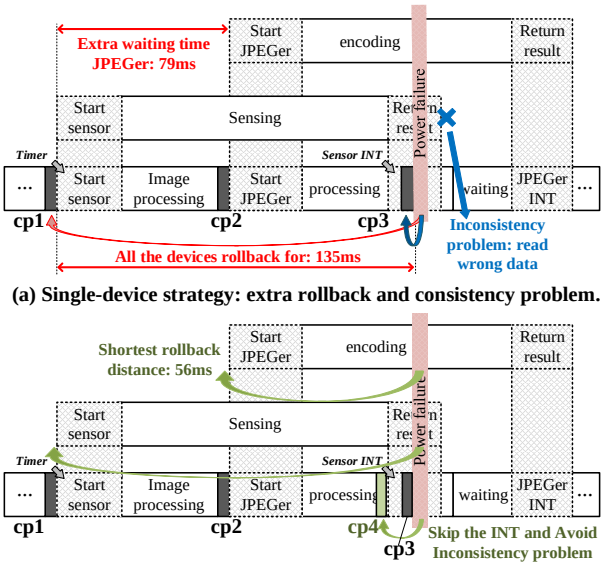


Figure 3: Challenges using single-device checkpointing strategies in a multi-device system. A distributed and reliable checkpointing framework is needed.

An ideal multi-device recovery should support different devices to roll back separately with minimum overheads while maintaining system consistency. In Fig. 3 (b), if the processor is restored to the interrupt position without rollback and the JPEGer is also restarted instantly, the recovery overhead can be reduced by 58.5%. However, *cp3* is still not the correct position because of the inconsistency problem. *cp4* is a more reliable checkpoint located before the interrupt, which prevents the processor from receiving erroneous data. Therefore, an automatic checkpointing tool supporting individual and reliable recovery of multi-device TPCs is also needed even with the enhanced hardware.

3. SYSTEM OVERVIEW OF REMARK

To address the two challenges raised in Sec. 2, this paper proposes REMARK, an efficient and reliable multi-device checkpointing framework. Fig. 4 shows the overview of REMARK framework, which contains an enhanced NVP architecture (Sec. 4), an offline program transformer (Sec. 5) and an online recover procedure (Sec. 6). The offline transformer pre-sets checkpoints to convert the original program to recoverable program for online execution. The online recover procedure will adjust the checkpoints dynamically for further efficiency. The rest of this section presents the overview of each component. The details will be covered in the rest of this paper.

Enhanced NVP Hardware Support.
As shown in Fig. 4, on top of existing NVP and NVIO that can instantly backup and restore the system state, REMARK adds four new modules to satisfy the requirements for flexible B/R functions and efficient peripheral recovery in the multi-device checkpointing strategy.

Two of these modules are used to help implement flexible and reliable processor checkpointing. **B/R Manager** is designed based on the traditional B/R controller enabling both active and passive B/R operation. Moreover, bootstrap is also added in this module to control the processor starting procedure. **INT Recognizer (IRec)** is used to place safe checkpoint when power failure crashes a hardware interrupt where interactions between processor and peripherals locates. The other two modules are used to realize efficient peripheral configuration and restart. **Peripheral State Registers (PSRs)** is a real-time monitor of the peripheral state to support the efficient selective peripheral configuration. **Peripheral Restart Module (PRM)** accelerates the peripheral restart procedure by automatically locating the start function in the program and correctly returning. With the help of PRM, extra rollback of processor is removed. To access these new modules, five new instructions are proposed as shown in Table 1. Design details of the hardware architecture are presented in Sec. 4 and Fig. 5.

Offline Program Transformer.
REMARK addresses the multi-device checkpointing issue by pre-set the checkpoints according to checkpointing rules to ensure the recovery efficiency and reliability. The transformer is used to transform the original program into recoverable program where reliable multi-device checkpoints are inserted. The transformer first insert recoverable checkpoints by scanning the peripheral related operations. Then the interrupts are processed to realize efficient and reliable interrupt recovery with the support of IR. Details are presented in Sec. 5.

Online Recover Procedure.
With the support of hardware architecture and the offline program transformer, an online recover procedure is proposed containing two parts, peripheral configuration and restart. Based on PSRs, REMARK adopts an 'init-used' strategy which only initializes and configures the invoked peripherals to avoid redundant reconfiguration overheads. A config function queue (CFQ) is used to track and stores the peripheral configuration information. The peripheral restart is realized by Initiator where peripheral checkpoints are stored. Initiator is supported by the bootstrap in B/R Manager to control the system restart work flow. After power failure, Initiator restarts all the checkpointed peripherals instantly and individually. Considering the interactions of devices, the reliability of the entire recover procedure is guaranteed by the flexible B/R functions. Details are explained in Sec. 6.

4. ENHANCED NVP ARCHITECTURE

To impletement REMARK, NVP is enhanced by four hardware modules to support flexible B/R functions and realize efficient peripheral recovery.

Backup/Restore (B/R) Manager.
Existing NVP only supports passive B/R functions, which limits the flexibility of NVP. Passive backup operation can only be triggered when the supply voltage is below the threshold. When power recovers, passive restore operation is activated automatically and no initiator program is allowed. Target on these challenges, B/R Manager extends active B/R functions to support flexible checkpointing. Moreover, a Bootstrap module is also added to support the initiator program. The structure of the B/R Manager is shown in Fig. 5.

Table 1: The extended instructions to the instruction set.

Instructions	Operators	Specifications
RSR	SRaddr oper2	Read PSRs with address *SRaddr* to register *oper2*.
WSR	oper1 SRaddr	Write the value of *oper1* to PSRs with address *SRaddr*.
WER	oper1	Write the value of *oper1* to *EndAddr* register in PRM.
ABR	oper1	Enable the active backup function, if $oper1 = 1$; enable the active restore function, if $oper2 = 0$.
EBR	oper1	Enable/Disable the backup and restore function when $oper1 = 1/oper1 = 0$.

The B/R controller is enhanced by control logics which are used to select the trigger signal of B/R functions. Two instructions, *ABR* and *EBR* (Table 1), are added to enable the active B/R functions. Active backup is realized by *ABR 1*, and active restore is *ABR 0*. In addition, *EBR 0* and *EBR 1* are used to disable and enable the B/R function, respectively.

Bootstrap is a module used to control the system clock and the reset signals. Its work flow is shown in Fig. 7. After the system is powered on, Bootstrap selects whether to enter the restart procedure or to start the program from the very beginning according to the 1^{st}*_start* signal. 1^{st}*_start* is an external signal set by users. If 1^{st}*_start* is set, the processor is started from the very beginning. Otherwise, Initiator is started to recover the peripherals and restore the processor.

Interrupt Recognizer (IRec).
Interrupt is an interaction between processor and the peripherals where careful checkpoints should be placed. IRec provides hardware support for reliable interrupt recovery. It recognizes the interrupt request and decides whether to recover the interrupt. Fig. 5 shows the hardware diagram of this module. Firstly, IRec contains a nonvolatile special register to store a programmable control bit, IR, which indicates whether an interrupt is recoverable. If the interrupt is recoverable, IR is set and the system will recover the interrupt as a normal processor task. Otherwise, IR is reset. When an unrecoverable interrupt request arrives, the control logic triggers B/R Manager to execute a backup operation and set a checkpoint before the interrupt. In this way, the system will resume from the checkpoint and skip the interrupt after power failure.

Peripheral Status Registers (PSRs).
PSRs are used to monitor the peripheral status in real-time to support efficient peripheral reconfiguration. PSRs in Fig. 5 is a group registers, each bit of which indicates whether a peripheral is ready or not. These states will be set when the peripherals are configured and cleared after power failures. Thus, a group of volatile registers are the ideal choice. Since the internal registers are all non-volatile and are resumed after power failures, PSRs are added as external registers.

PSRs can be accessed by the processor with the proposed instruction *RSR* and *WSR* as shown in Table 1. *RSR* reads the value from *SRaddr* into the register *oper2*. *SRaddr* is the

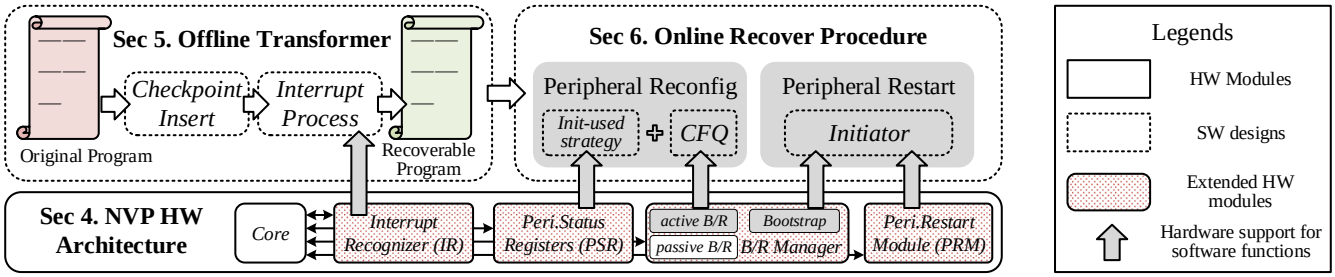


Figure 4: The HW/SW co-designed system diagram of REMARK.

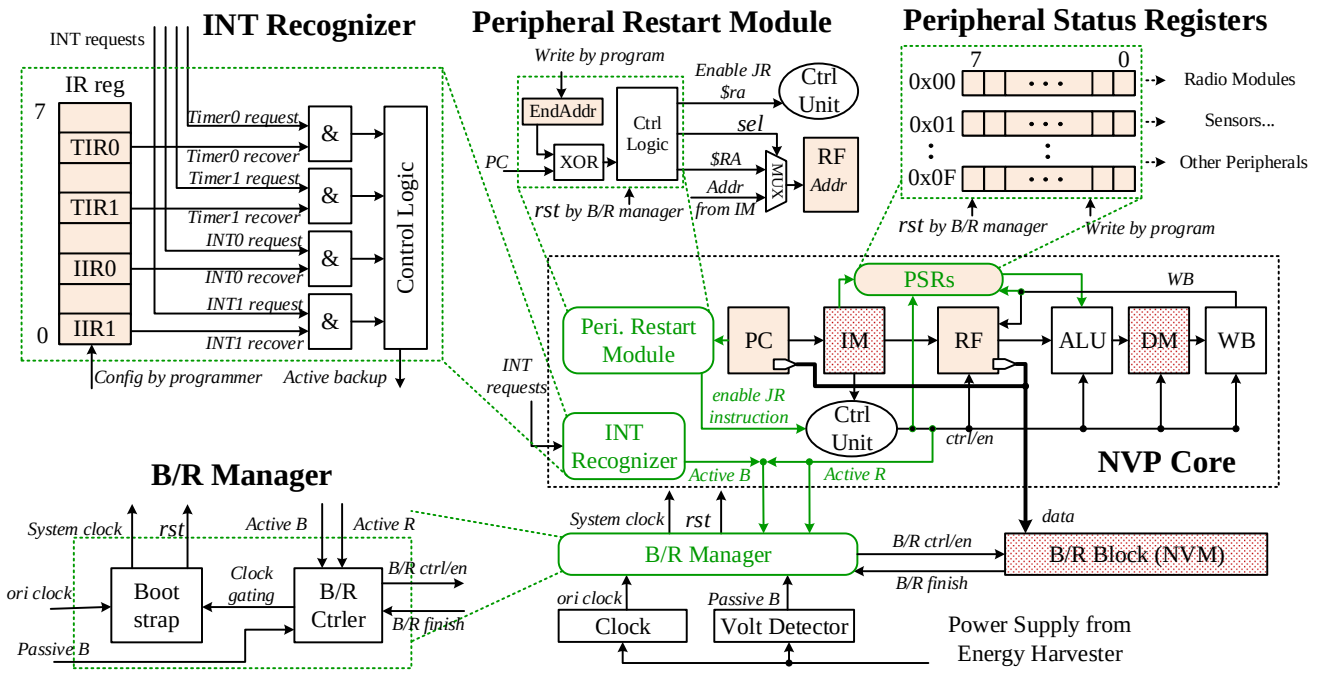
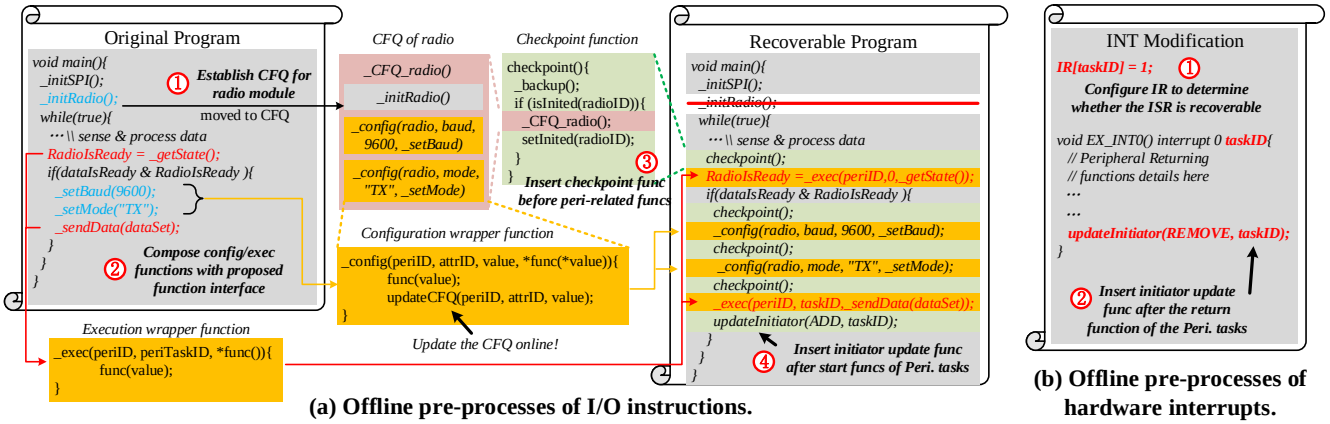


Figure 5: The hardware architecture of REMARK and its main modules.



(a) Offline pre-processes of I/O instructions.

(b) Offline pre-processes of hardware interrupts.

Figure 6: The program pre-processes during the software transformation stage.

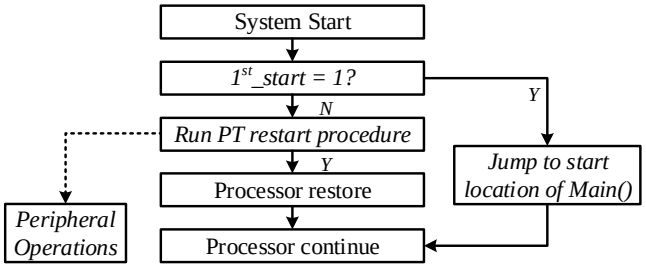


Figure 7: The flow chart of Initiator.

address of PSRs and has an independent address space from the register file. *oper2* can be the address of a normal register, a direct address or an indirect address. Similarly, *WSR* writes the value of second operand *oper1* into *SRaddr*. *oper1* can be the address of a normal register, a direct address, an indirect address or an immediate operand.

Peripheral Restart Module (PRM).
PRM is used to locate the start function of peripherals and realize correct returning during the restarting procedure. As shown in Fig. 5, PRM contains a special register *EndAddr*, an XOR gate and the control logic. When a peripheral needs to be restarted, PRM is called by the processor with two instructions. Firstly, the end address of the peripheral start function is stored to *EndAddr*. Then, the program jumps to the peripheral start function with a *JAL* instruction. When the program counter reaches the end address, the XOR gate generates a positive trigger and the control logic enables a *JR $ra* instruction automatically. In this way, the peripheral is successfully restarted and the program returns correctly to restart the next peripheral.

5. MULTI-DEIVCE CHECKPOINTING

With the enhanced NVP, REMARK will process the original program with the offline program transformer to pre-set safe checkpoints to collaborate with the new hardware modules to ensure correct online execution.

5.1 Checkpoint Rules for Multi-device TPC

In multi-device systems, we should realize reliable checkpoints considering the processor, the peripherals and the interactions between devices.

As discussed above, a peripheral cannot keep its state when it is interacting with the processor or the environment. Therefore, *the peripheral checkpoints can only be placed when the peripheral is idle.* For example, *cp1* and *cp2* in Fig. 8 are two available checkpoints to store the state of the sensor.

In contrast, NVP allows checkpoints at any position in the program and can recover safely. However, the interaction between processor and peripherals may cause inconsistency. When power fails during these interactions, the processor may keep its state while the peripheral needs to roll back to a previous position, which leads to inconsistency. Therefore, *the processor checkpoint should be placed out of the processor-peripheral interaction operations.*

Fig. 8 shows two types of processor-peripheral interactions: I/O operations and interrupts. First type is I/O operations started by processor to write configurations or commands to peripherals. Here, if power fails when the processor is configuring or starting the sensor, the processor should rollback to *cp1* or *cp2* to restart the I/O operation. Second type is the external interrupt activated by peripherals. Sensor triggers an interrupt when the sensing task is completed, then, the processor read the operation result from the sensor. If power failure occurs during an interrupt, the processor should not recover the interrupt service routine (ISR). Instead, it should rollback to *cp3*, the entrance before the interrupt in the main process. In this way, when the power resumes, ISR is skipped to avoid processing corrupted data in ISR.

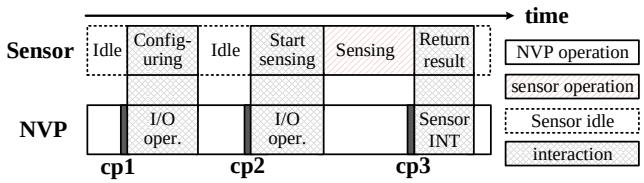


Figure 8: Interactions between processor and peripheral.

Based on these rules, the offline program transformer scans the whole program and inserts safe checkpoints according to the positions of the I/O instructions and the interrupts.

5.2 Offline Program Transformer

The offline program transformer contains two steps to handle the I/O instrucitons and the hardware interrupts.

Handle I/O Instructions.
REMARK categorizes I/O instructions into two groups: peripheral configuration instructions and peripheral execution instructions. Configuration instructions, including the initialization instructions, are used to configure the internal registers of peripherals. Execution instructions are used to realize the functionalities of peripherals, such as starting a communication, fetching data from a peripheral buffer, etc. Before execution instructions, configuration instructions are required to set the peripherals ready.

The pre-processing for I/O instructions involves three steps. Fig. 6 shows a data transmission example to illustrate the procedure. Firstly, REMARK establishes a configuration instruction queue (CFQ) for each peripheral. CFQ tracks configuration instructions, including the initialization instructions, of a peripheral, which are used to configure the peripheral in case of power failures. Every time a new configuration instruction is executed, CFQ will be updated to include the latest configuration modification.

To allow REMARK to recognize and pre-process the I/O instructions, the programmers have to implement the peripheral configuration and execution instructions with wrapper functions indicated in the orange parts of Fig. 6. Here, the configuration wrapper function contains the peripheral ID, the target attribute, the configuration value and the pointer of the configuration instruction. It will execute the configuration instructions and update CFQ with function *_updateCFQ()*, which will be described in Sec. 6. Execution wrapper function contains the peripheral ID, the peripheral operation ID and the target instruction pointer. A non-zero peripheral operation ID represents that the instruction starts a peripheral operation.

After peripheral related instructions are recognized, two recovering functions are inserted into the program to set checkpoints of processor and peripherals. Firstly, a **checkpoint function** is inserted before each I/O instruction to checkpoint the processor. With the checkpoint function, the processor rolls back to reconfigure the peripherals to avoid inconsistency. Second, **initiator update functions**, *updateInitiator()*, are inserted after the execution instructions of peripheral operations to set peripheral checkpoints. *updateInitiator(ADD, taskID)* records the start position and the end position of the start program of the peripheral operations, *PT(taskID)*, and adds its peripheral checkpoint into Initiator. These checkpoints are used to restart the crashed peripheral operations. Circle 3 and 4 in Fig. 6 (a) illustrates these two functions.

Handle Hardware Interrupts.
Fig. 6 (b) shows the ISR processing procedure with two steps. The first step is to set safe checkpoints considering the consistency of processor-peripheral interactions. Generally, an ISR should not be recovered, if it contains processor-peripheral interactions. Consider the case that, the embedded volatile memory in a sensor loses all the data after power failure. If the processor resumes the ISR, it will receive wrong data from the sensor which leads to system errors. However, ISRs which contain no interactions can be safely recovered after power failure, such as a counter triggered by a timer. To solve this application specific issue, REMARK provides a control bit, *IR*, to the application developers to indicate whether an ISR can be restored according to application requirements. Programmers can reset IR, if an ISR is recoverable, or set IR, if it's unrecoverable. The IR is stored in IRec and the latter can set reliable checkpoints automatically during execution.

The second step is to maintain the peripheral checkpoints. An interrupt started by a peripheral indicates that the peripheral has completed its task and will enter idle mode. The system does not need to restart a task if power fails after its completion. To remove such invalid checkpoint, Initiator update function is inserted at the end of each ISR. *updateInitiator(REMOVE, taskID)* removes the peripheral checkpoint of *PT(taskID)*.

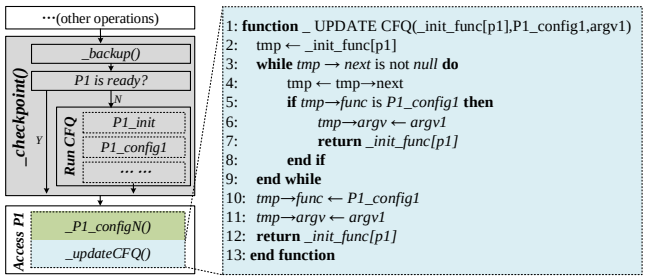
In this way, the program is transformed to be recoverable. It is now ready to realize recoveries on the proposed hardware. Noted that, *programmers can dictate an operation as unrecoverable by not using the wrapper codes according to application-specific reasons, such as data freshness requirements*.

6. ONLINE RECOVER PROCEDURE

With the support of hardware architecture and offline program transformer, this section presents the online recover procedure, including the reconfiguration (Sec. 6.1) and the restart (Sec. 6.2) of peripherals.

6.1 Peripheral Reconfiguration

A peripheral is ready to be called only if it is well configured. Instead of employing 'init-all' strategy in QuickRecall [10], 'init-used' approach is proposed in this paper to reduce large reconfiguration overheads. The main idea is that, instead of reconfiguring all the peripherals, we only need to reconfigure necessary peripherals after power failure.



(a) *checkpoint()* in program. **(b) Pseudo code of _updateCFQ().**

Figure 9: Program when peripheral *P1* is accessed. *_checkpoint()* is used to configure *P1* and *_updateCFQ()* is used to track the latest configurations.

In order to achieve this goal, we need the synergy of both offline inserted checkpoint function and online recover procedure. Fig. 9 (a) shows the program where the checkpoint function, *_checkpoint()*, is inserted before the accesses of *P1*. *_checkpoint()* includes two parts: a backup function and a peripheral configuration part. The backup function places a checkpoint before the I/O instruction in case of power failures. Then, the program checks whether *P1* is ready according to PSRs. If *P1* is not ready, the program will configure *P1* with the configurations in CFQ. Otherwise, the program skips the configuration part. In this way, the 'init-used' approach configures a peripheral only before it is invoked.

During system execution, the configuration of a peripheral is dynamically updated. In order to track the changes of configurations, CFQ is maintained by *_updateCFQ()* function inserted after each configuration function. Fig. 9 (a) shows the program where peripheral *P1* is configured by *_P1_configN()*. To record this configuration, *_updateCFQ()* function is added after *_P1_configN()*. Fig. 9 (b) shows the pseudo code of *_updateCFQ()*. *_updateCFQ()* can either update the parameters of an existing configuration function or add the new configuration function into CFQ.

6.2 Peripheral Operation Restart

After the peripherals are reconfigured, we are ready to restart the peripheral operations. REMARK restarts the peripheral operations from peripheral checkpoints with the help of PRM (described in Sec. 4) and Initiator.

Restart Peripheral Operations by Initiator.
As shown in Fig. 10, Initiator is a program located in the top of instruction memory used to store the peripheral checkpoints. It contains a list of peripheral checkpoints and a processor restore instruction. Each peripheral checkpoint is a restart function, *_restart_PT()*, which can restart a specific peripheral operation. The queue is empty at the beginning and is dynamically updated by *updateInitiator()* described in Sec. 5.2.

Each peripheral checkpoint consists of two instructions, *WER* and *JAL*. Fig. 10 shows how to restart a peripheral operation, *PT1*, with its checkpoint. In this example, the first instruction writes the end address of *PT1* start function, *endAddr1*, into PRM. The second instruction, *JAL*, jumps to the start address of *PT1* start function and links the address of the next instruction in Initiator into register *$ra*. Then,

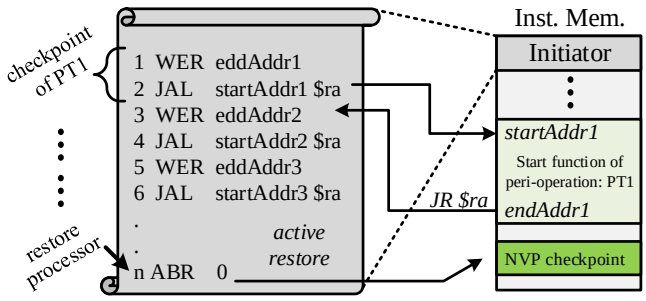


Figure 10: Restart peripheral operations in Initiator.

the peripheral operation *PT1* is restarted. When the start function is finished, the PC counter reaches *endAddr1*. Then PRM automatically enable an *JR $ra* operation to return the program to Initiator and continue restarting the next peripheral operation.

The Restart Flows after Power Failure.
The restart flows are different from each other when power fails during different stages. A peripheral operation can be decomposed into three stages: the starting stage, the execution stage, and the returning stage. During the starting stage, the processor uses I/O instructions to start a peripheral operation. The execution stage lasts until the returning interrupt is triggered. Finally, the ISR returns the result to the processor in the returning stage. The peripheral operation is completed only if the ISR is finished.

When power fails during the starting stage, the parallel peripheral operation has not been started. No peripheral checkpoints are placed, and the processor only needs to roll-back and re-run the starting stage. Once the starting stage is completed, *updateInitiator()* inserted during offline program transformation will place a peripheral checkpoint in Initiator.

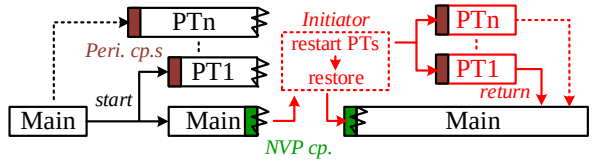
When power fails during the execution stage, as shown in Fig. 11 (a), the processor restores its state and the peripheral operations are restarted from their checkpoints individually.

When power fails during the returning stage, as shown in Fig. 11 (b), the peripherals lose all the data and fail to return the results. Therefore, the peripheral operations still need to be restarted. Meanwhile, the processor needs to restore to the checkpoint located before the ISR, as discussed in Sec. 5.2. When a successful ISR is completed, the peripheral operation is also finished. Therefore, the peripheral checkpoint to restart this operation needs to be removed from Initiator by another *_updateInitiator()* function.

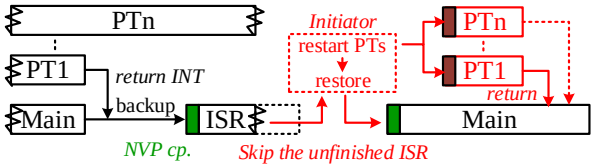
When power fails during the recover procedure, the devices can roll back to their checkpoints if these checkpoints are not crashed. Therefore, completing the backup function is essential to ensure the validity of a checkpoint. To guarantee the success of backup operation, NVP adopts a $4.7\mu F$ on-chip capacitor. In the hardware prototype of this paper, each backup operation consumes $77.69nJ$ in $7\mu s$. In this way, the system can safely backup whenever power fails.

7. HARDWARE IMPLEMENTATION

In order to evaluate the performance and resiliency, we implements REMARK on an Intel 8051 based NVP as well as a hardware platform, *NVnode*. In this section, we first



(a) Power failures during the execution stage.



(b) Power failures during the returning stage.

Figure 11: Power failures during the starting and the returning stage of peripheral operations.

Table 2: The parameters of NVP.

Parameter	Value	Parameter	Value
Process Technology	$0.13\mu m$	Clock frequency	$1MHz$
Backup Time	$5\mu s$	Restore Time	$3\mu s$

present the hardware platform in Sec. 7.1. Then, the improvement of the peripheral recovery and the overall performance are presented in Sec. 7.2 and Sec. 7.3, respectively.

7.1 Hardware Platform

Fig. 12 shows a picture of NVP chip and NVnode platform. The structure of NVnode is presented in Fig. 13. NVnode contains a power supply system, an NVP with proposed REMARK modules and peripherals including ZigBee transceiver and sensors. The energy storage is a $20\mu F$ capacitor which is charged between $1.2V$ and $5V$. The total storage of the capacitor is $212\mu J$ while the energy efficiency is 90%.

The parameters of the proposed NVP are listed in Table. 2. This NVP is designed based on Intel 8051 ISA. It contains a $256B$ register file using NVFF, a $32KB$ FeRAM data memory, a B/R Manager and an NVRF module to enable REMARK.

As shown in Fig. 13, NVRF realizes the functionality of PSRs and PRM, which enable efficient and reliable peripheral configuration and restart operations. NVRF contains a $32B$ register file, an RF controller, and an SPI interface. The register file stores the state of the wireless transceiver. The RF controller can reconfigure the transceiver and restart the data transmission task automatically and efficiently after power failure. The SPI interface is used to mount to SPI bus, which also connect to a ZigBee transceiver and several sensors.

7.2 Device Recovery Improvement of REMARK

To show the effectiveness of REMARK, this subsection first shows the overhead reduction of a single transceiver recovery. Then, it shows the efficiency improvement when there are multiple peripheral devices. REMARK is compared with the traditional single device recovery strategies, such as QuickRecall [10]. Since the hardware platform utilizes NVP, a single device rollback strategy (SDRB) is designed based on QuickRecall in NVnode for comparison purposes. SDRB restarts the peripheral by the processor execut-

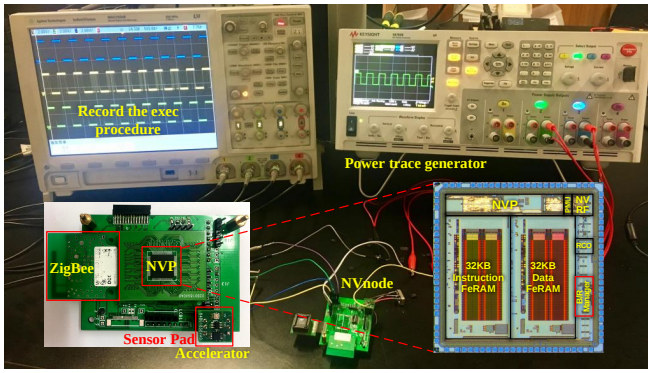


Figure 12: The hardware platform, NVnode. REMARK is implemented with the B/R manager and the NVRF controller in an 8051 NVP chip under $0.13\mu m$ technology.

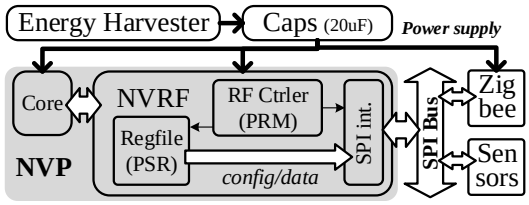


Figure 13: The structure of NVnode. The energy storage is $20\mu F$. NVRF implements the functionality of PSRs and PRM to accelerate the peripheral recovery.

ing configuration and restart functions.

Fig. 14 (a) compares the transceiver recover time between the hardware NVRF and the software recover procedure, SDRB. After power failure, NVRF automatically configures and restarts the transceiver in $1.22ms$. During this procedure, NVRF first restores the register file in $9\mu s$, and then performs the configuration and restart operation in $0.91ms$ and $0.31ms$, respectively. SDRB performs the recovery with three steps. Firstly, the processor recovers the configurations and the data of the transceiver in the NVFFs in $9\mu s$. Then, the transceiver is reset and configured via SPI interfaces in $25.3ms$. Finally, the transmission operation is restarted in $7.7ms$. The total overhead of software recovery is 33ms. Thus, REMARK accelerates the process by $27.0\times$.

Fig. 14 (b) shows the comparison between the multi-device checkpointing strategy in REMARK and the rollback strategy in SDRB. From the figure, we can see that, REMARK places checkpoints of NVP and the transceiver individually and realizes parallel recovery with the help of NVRF. The distributed rollback strategy for multi-device TPC recovers the peripherals and the processor in parallel. The total recovery overhead of REMARK equals to the largest recovery overhead of each device, which is smaller than recovering all the devices in serial. Moreover, the *'init-used'* strategy allows REMARK only to recover a peripheral when it is invoked, while SDRB suffers large re-configuration overhead adopting the *'init-all'* strategy. In addition, SDRB also introduces processor rollbacks, whose overhead is related to the power failure position. With these factors, REMARK accelerates the peripheral recovery procedure by $34.1\times$ than SDRB in this case.

7.3 Overall Performance Evaluation

The improvement in recovery performance leads to better overall system performance. A *'bridge-monitoring'* application, a pure transmission application, and a pure sensing application are used to evaluate the overall performance improvement. *'bridge-monitoring'* is a data collecting application which contains both sensing and transmission tasks. These tasks are repeated periodically. The work flow of *'bridge-monitoring'* is shown in Fig. 15 (a). The power trace is a Wi-Fi power profile, whose average power is $93.3\mu W$. NVnode starts to work, when the capacitor is charged to $5V$. The system fails and the capacitor gets recharged when the voltage of capacitor falls below $1.2V$.

The benchmarks are executed for 2 minutes with REMARK and SDRB. Fig. 15 (b) compares the task completion efficiency. For the benchmarks with pure transmission and sensing tasks, REMARK achieves $5.7\times$ and $5.6\times$ efficiency improvements. For the âĂŸbridgemonitoringâĂŹ benchmark, REMARK completes $13\times$ more data collection and transmission tasks than SDRB. This is because the latter contains more peripherals where the performance of SDRB significantly deteriorates. Due to the improvement in task completion efficiency, the energy consumption is also reduced accordingly.

8. EVALUATION AND ANALYSIS

In addition to the real chip evaluation, we also want to investigate the impact of design parameters in REMARK. To explore larger design space, we develop a simulator, *NVnodeSim* to gain more insights in designing TPC system. In this section, we first present the simulator in Sec. 8.1. Then, we tune the simulator to different settings and evaluate it with more general benchmarks to show how different design options affect the system performance in Sec. 8.2 and 8.3. Based on the observations, we summarize three design rules in Sec 8.4.

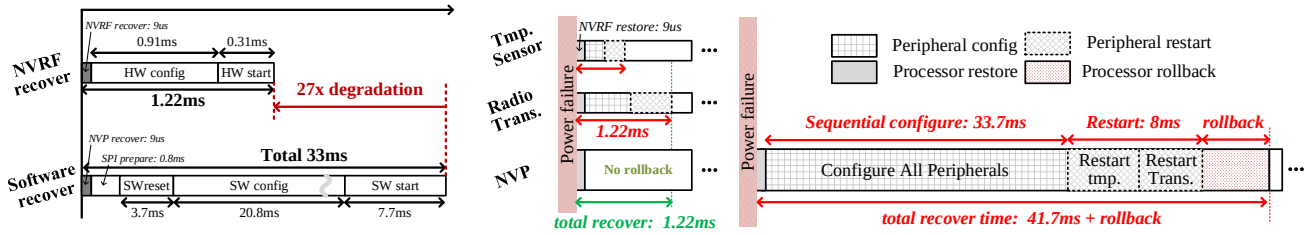
8.1 Experiment Settings

The inputs of NVnodeSim are power traces and benchmark profiles. The power traces are solar power data from an open source database [16]. The benchmarks consist of three kinds of operations as shown in Table 3. All parameters are extracted from datasheets or the cycle-accurate processor simulator Keil C. Three common computing tasks for IoT are adopted. Two I/O interfaces and five peripherals are also used, including sensors, transceivers, and an off-chip memory.

To validate the accuracy of NVnodeSim, Table 4 compares the task execution time between NVnode and NVnodeSim. It shows that NVnodeSim provides reasonable accuracy for architecture-level exploration.

8.2 Evaluation with Multiple Benchmarks

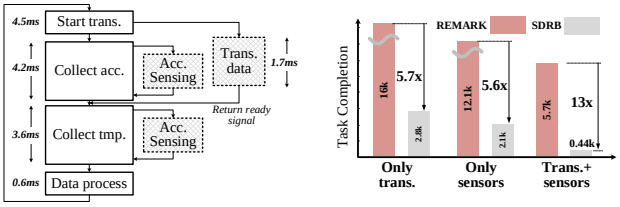
We further evaluate the performance of REMARK with ten additional benchmarks on the simulator. Fig. 16 (a) shows the recovery overhead distribution under ten benchmarks. On average, REMARK reduces total overhead by 64% compared with SDRB. Fig. 16 (b) shows that, the overhead of SDRB rises rapidly when multiple peripherals are included in the program. This is because the re-initialization overhead



(a) Peripheral recover overhead reduction

(b) Recover performance improvement by multi-device checkpointing

Figure 14: Recover overhead and task completeness comparison between REMARK and SDRB.



(a) Work flow of bridge-monitoring.

(b) Task completion efficiency.

Figure 15: *bridge-monitoring* application and task completeness comparison between REMARK and SDRB.

Table 3: Benchmarks: compute, I/O and peri. oper.

compute operation	time /cycles	compute operation	time /cycles	compute operation	time /cycles
crc16	122	rsa	409	fft16	275

I/O bus	init time/cycles	read time/cycles	write time/cycles
I2C	403	$270+90\times N$	$180+90\times N$
SPI	171	$320+80\times N$	$320+80\times N$

Peripheral	Chip	init./ms	I/O bus	exec time/ms
ZigBee	ML7266	531	SPI	$5.7+0.14\times N$
Acc. Sensor	LIS331DLH	2391	I2C	0.451
Image Sensor	LUPA1300	14,952	I2C	118.0
Tmp. Sensor	TMP101	566	I2C	0.283
FeRAM Mem	FM25V20	76	SPI	–

grows rapidly. The system even falls into deadlock when the re-initialization overhead is too large to complete during power failure intervals. Thanks to the efficient *'init-used'* strategy, the re-initialization overhead of REMARK stays modest. From the experiment, we can see that *REMARK can recover the system with multiple peripherals more efficiently and reliably*.

8.3 Analysis the Peripheral Related Factors

An important observation in Fig. 16 is that the re-initialization and rollback overheads dominate the recovery overhead of REMARK. These overheads are caused by peripheral disruptions. The operation distribution in the program profile has a direct effect on peripheral disruptions. Therefore, we explore the impact of three distribution factors of I/O and peripheral operations as below to gain insights on TPC system design.

Peripheral Operation Percentage.
Fig. 17 (a) shows the overheads and the failure times using

Table 4: Accuracy evaluation of NVnodeSim.

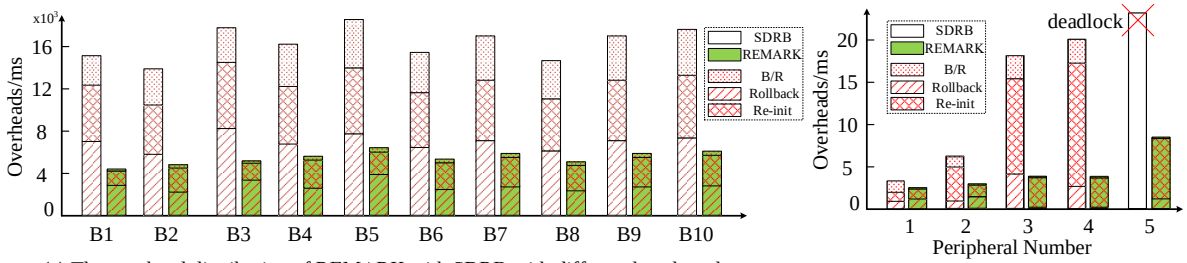
Benchmarks	Failure Times		Execution Time/ms		
	NVnode	Sim	NVnode	Sim	Error
RSA	3	3	3.560	3.512	-1.35%
CRC	1	1	1.316	1.309	-0.53%
MemLS	12	11	12.05	11.99	-0.49%
Sense	42	42	42.95	42.38	-1.33%

benchmarks containing different peripheral operation percentages. When more peripheral operations are used in a benchmark, power failures are more likely to cause peripheral disruptions. According to the results, when the percentage increases, the total overhead increases slower than linear trend. The reason is that recovering peripherals introduces more peripheral operations which increase its actual percentage compared with the original program. Therefore, when a benchmark contains more than 70% peripheral operations, the actual percentage approaches saturation and the total overhead grows slowly.

Peripheral Operation Length.
The length of each I/O operation affects the rollback distance during recovery. Fig. 17 (b) shows the effect of lengths on the peripheral disruption times and the recovery overheads. Rollback overhead increases when the length of peripheral operation increases from 0.5*ms* to 3.5*ms*. This is because longer peripheral operations may cause higher rollback distance. Therefore, *shorten the length of each peripheral operation can reduce the overhead of single recovery*.

Overlap of I/O and Peripheral Operations.
Since the processor and peripherals operate in parallel, the interaction among these devices also affects the performance of REMARK. This part explores the impact of the overlap between I/O and peripheral operations.

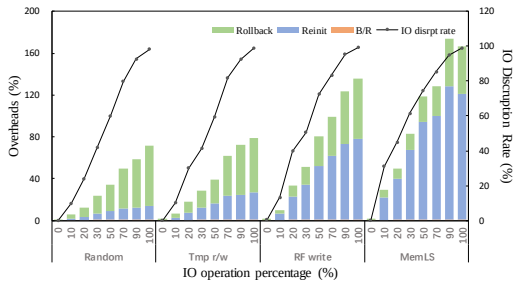
In a wireless sensor node, I/O operations are executed by the processor, including off-chip memory accesses, sensor configurations, etc. Peripheral operations are executed on peripherals, including sensing and data transmission tasks. In such a system, the total recovery overhead of the system is determined by the largest overhead either on the processor or the peripheral. Table 5 shows the recovery overhead under different overlap ratios, where I/O/Peri. overlap is the overlap percentage of the I/O and peripheral operations. The I/O/Peri. overlap ranges from 0 to 100% in two different benchmarks (sensing and data transmission). The breakdown of the recovery overhead of processor and peripherals



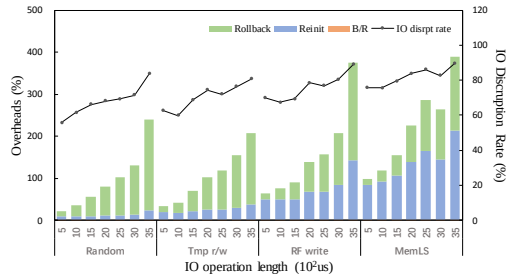
(a) The overhead distribution of REMARK with SDRB with different benchmarks.

(b) Overheads v.s. peripheral/IO number.

Figure 16: Recover overheads comparison between REMARK and SDRB.



(a) Overhead v.s. IO percentage.



(b) Overhead v.s. IO operation length.

Figure 17: The performance overheads with various peripheral operation percentage and length.

are shown. The stall time represent that the processor waits for the peripherals. Overhead reduction shows how much reduction is achieved when there are more overlaps compared with the program with no overlap. Results show that, the overhead decreases as much as 36.5% when the overlap percentage increases.

Since most of sensor nodes work in a periodical way, overlapping I/O operations of previous loops with peripheral operations in current loop could be an effective solution to avoid large recovery overhead. Therefore, *with REMARK, software designer should overlap I/O and peripheral operations as much as possible to reduce the stall time of recovery process*.

8.4 Analysis Conclusion

The evaluation experiments show that REMARK is able to support the reliability and task progress of a TPC system with multiple peripherals. According to these analysis, we summarize three design rules for performance-aware software:

- **Smaller I/O percentage.** Reduce I/O operations as much as possible in TPCs to avoid expensive recovery overhead.
- **Shorter I/O operation length.** Shorter I/O operations have smaller overhead. It is benefit to split a long I/O operation into several shorter ones.
- **More overlap between I/O and peripheral operations.** Overlapping parallel operations can remove the stall time of the recovery process. It is also benefit to overlap the sensing and transmission operations in different cycles.

Table 5: Overheads with different overlap percentages.

Benchmarks	Sensing			Data Transmission		
I/O/Peri. overlap	0%	50%	100%	0%	50%	100%
proc. overhead /%	44.9	43.6	43.4	43.4	42.7	41.7
peri. overhead /%	18.4	16.0	18.3	80.9	74.8	74.2
stall time/%	18.4	10.8	6.1	80.9	62.4	39.2
Total overhead /%	63.3	54.4	49.5	124.3	105.1	78.9
Overhead Reduction/%	–	14.0	21.8	–	15.4	36.5

9. RELATED WORK

To support the task progress under intermittent power supply, both software and hardware recovery mechanisms have been proposed to support automatic recovery of the processor.

Software Mechanism: Checkpointing.
Checkpointing is a software based recovery mechanism for processors. Checkpoints are placed in the program, where the processor state are stored into non-volatile memories. By rolling back to these checkpoints, the processor can keep the task progress after power failure [17, 1]. Bronevetsky et al. propose the application level checkpointing strategy on the shared memory systems to enhance the reliability [18]. Mementos proposes the concept of transiently powered computer and presents a software processing strategy which transforms the general purpose programs into interruptible computations for the general hardware architecture with Flash memory [3]. Balsamo et al. propose Hibernus and Hibernus++ using FeRAM and an interrupt-based checkpointing solution to reduce the performance overhead [9, 19, 20]. Jayakumar et al. design QuickRecall and em-Map to integrate FeRAM into main memory to reduce the backup data

size and lower the failure voltage threshold [10, 21]. In addition to general purpose processor architectures, Mirhoseini et al. target the Application Specific Integrated Circuits (*ASICs*) and propose Idetic and Chime with the help of control data flow graphs (*CDFGs*) [22, 23, 24].

These schemes achieve continuous progress of processors with out peripherals. Large rollback overhead may take place while recovering a multi-device system with these strategies.

Hardware Mechanism: Non-volatile Processor.
Besides the checkpointing mechanisms, researchers also provide solutions in hardware domain. Non-volatile processor draws a lot of attention due to its ability to store the system state and data automatically in hardware. The first processor chip designed by Wang et al. using FeRAM realizes the ability to backup and restore the processor state and data within $3\mu s$ [11]. Bartling et al. propose a non-volatile logic based Cortex-M0 chip with higher performance and lower leakages [25]. Sakimura et al. from NEC propose the non-volatile magnetic flip-flops [26] and a 20MHz non-volatile micro-controller with STT-RAM [27]. Recently, Liu et al. propose an enhanced NVP based on ReRAM which has the highest integration level [28]. In addition, Li et al. propose the non-volatile I/O (NVIO) enabling efficient automatic reconfiguration of I/O interfaces [12].

Compared with software checkpointing strategies, nonvolatile devices enable state recovery with higher speed and fewer rollbacks. However, these nonvolatile processors lack of flexibility on checkpointing and recovery which may cause inconsistency problems in a multi-device system.

Inconsistency and Program Partitioning.
Researchers have noticed the problem of the inconsistency issues while rolling back a system with nonvolatile memories. B. Lucia et al. [15] discover and model the data inconsistency problem caused by improper rollbacks after power failures. After that, more works [14], [29], [13] analyze the scenarios of checkpoint recoveries and propose careful checkpointing strategies to ensure the correctness of all checkpoints. Recently, an energy-interference-free debugger for intermittent powered systems is proposed by A. Colin et al. to provide a more reliable and convenient debugging platform [30]. However, reliabliliy and efficiency challenges still exist when adopting NVP in a TPC system with multiple peripherals and interrupts.

10. CONCLUSION

This paper proposes REMARK, a Reliable and Efficient Multitask Recovery FrAmewoRK, which realizes reliable and efficient TPC recovery. REMARK provides an automatic and efficient peripheral recover hardware architecture supporting the system with multiple peripherals. Moreover, a new checkpointing strategy for the multi-device system is also provided to avoid the inconsistency issues between different devices and incurs the lowest rollback overhead. A REMARK-enabled NVP is fabricated and evaluated, which demonstrates that REMARK enables reliable data transmission with 13× completeness improvements compared with state-of-the-art. In conclusion, REMARK implements a critical component that existing NVP solutions are missing and will greatly improve the applicability of TPCs in the IoT era.

11. REFERENCES

[1] 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 *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 526 – 537, 2015.

[2] B. Ransford, *Transiently powered computers*. PhD thesis, University of Massachusetts Amherst, 2013.

[3] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," *ASPLOS 2011, Acm Sigplan Notices*, vol. 47, no. 4, pp. 159–170, 2012.

[4] C. Wang, N. Chang, Y. Kim, S. Park, Y. Liu, H. G. Lee, R. Luo, and H. Yang, "Storage-less and converter-less maximum power point tracking of photovoltaic cells for a nonvolatile microprocessor," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 379–384, IEEE, 2014.

[5] K. M. Abas, "Solar-powered smart wireless camera network for outdoor monitoring," *Dissertations & Theses - Gradworks*, 2015.

[6] C. Alippi, R. Camplani, C. Galperti, and M. Roveri, "A robust, adaptive, solar-powered wsn framework for aquatic environmental monitoring," *IEEE Sensors Journal*, vol. 11, no. 1, pp. 45–55, 2011.

[7] A. Malaver, N. Motta, P. Corke, and F. Gonzalez, "Development and integration of a solar powered unmanned aerial vehicle and a wireless sensor network to monitor greenhouse gases," *Sensors*, vol. 15, no. 2, pp. 4072–4096, 2014.

[8] A. J. Rojas, L. F. Gonzalez, N. Motta, and T. F. Villa, "Design and flight testing of an integrated solar powered uav and wsn for remote gas sensing," in *Aerospace Conference*, pp. 1–10, 2015.

[9] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *Embedded Systems Letters, IEEE*, vol. 7, no. 1, pp. 15–18, 2015.

[10] 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 *2014 13th International Conference on Embedded Systems*, pp. 330–335, IEEE, 2014.

[11] 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 *2012 Proceedings of ESSCIRC (ESSCIRC)*, pp. 149–152, IEEE, 2012.

[12] Z. Li, Y. Liu, D. Zhang, C. J. Xue, Z. Wang, X. Shi, W. Sun, J. Shu, and H. Yang, "Hw/sw co-design of nonvolatile io system in energy harvesting sensor nodes for optimal data acquisition," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 154, ACM, 2016.

[13] M. Xie, M. Zhao, C. Pan, J. Hu, Y. Liu, and C. J. Xue, "Fixing the broken time machine," *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, pp. 1–6, 2015.

[14] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, p. 17, 2016.

[15] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, pp. 575–585, 2015.

[16] Measurement and I. D. C. (MIDC). `http://www.nrel.gov/midc/srrl_bms/`.

[17] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *Acm Transactions on Architecture & Code Optimization*, vol. 8, no. 2, pp. 510–521, 2011.

[18] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," *ASPLOS 2004, Acm Sigops Operating Systems Review*, vol. 32, no. 5, pp. 235–247, 2004.

[19] D. Balsamo, A. Weddell, A. Das, A. Arreola, D. Brunelli, B. Al-Hashimi, G. Merrett, and L. Benini, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[20] A. Rodriguez Arreola, D. Balsamo, A. K. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, and G. V. Merrett, "Approaches to transient computing for energy harvesting systems: A quantitative evaluation," in *International Workshop on Energy Harvesting and Energy Neutral Sensing Systems*, pp. 3–8, 2015.

[21] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan, "Q uick r ecall: A hw/sw approach for computing across power cycles in transiently powered computers," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 12, no. 1, p. 8, 2015.

[22] A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics," in *IEEE International Conference on Pervasive Computing and Communications*, pp. 216–224, 2013.

[23] A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Automated checkpointing for enabling intensive applications on energy harvesting devices," in *IEEE International Symposium on Low Power Electronics and Design*, pp. 27–32, 2013.

[24] A. Mirhoseini, B. D. Rouhani, E. Songhori, and F. Koushanfar, "Chime: Checkpointing long computations on intermittently energized iot devices," *IEEE Transactions on Multi-Scale Computing Systems*, pp. 1–1, 2016.

[25] S. C. Bartling, S. Khanna, M. P. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. P. Mcadams, "An 8mhz 75ua/mhz zero-leakage non-volatile logic-based cortex-m0 mcu soc exhibiting 100 percents digital state retention at vdd=0v with <400ns wakeup and sleep transitions," in *IEEE International Solid-State Circuits Conference*, pp. 432–433, 2013.

[26] N. Sakimura, T. Sugibayashi, R. Nebashi, and N. Kasai, "Nonvolatile magnetic flip-flop for standby-power-free socs," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 8, pp. 2244–2250, 2009.

[27] N. Sakimura, Y. Tsuji, R. Nebashi, H. Honjo, A. Morioka, S. Fukami, S. Miura, N. Kasai, T. Endoh, H. Ohno, T. Hanyu, and T. Sugibayashi, "10.5a 90nm 20mhz fully nonvolatile microcontroller for standby-power-critical applications," *IEEE International Solid-State Circuits Conference*, vol. 12, no. 4, pp. 184–185, 2014.

[28] Y. Liu, Z. Wang, A. Lee, F. Su, C.-P. Lo, Z. Yuan, C.-C. Lin, Q. Wei, Y. Wang, Y.-C. King, *et al.*, "A 65nm reram-enabled nonvolatile processor with 6x reduction in restore time and 4x higher clock frequency using adaptive data retention and self-write-termination nonvolatile logic," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 84–86, IEEE, 2016.

[29] A. Colin and B. Lucia, "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*, pp. 514–530, ACM, 2016.

[30] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," *ASPLOS 2016, Acm Sigops Operating Systems Review*, vol. 50, no. 2, pp. 577–589, 2016.