Intermittent Energy Harvesting System Architecture: A Concurrent Atomic Peripherals' Perspective

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 continuable processor executions under intermittent power supply using non-volatile memories and checkpointing techniques. However, few addresses the unique challenges associating checkpoint and recover a system with concurrent atomic 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 concurrent operations in a TPC with peripheral devices. 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

By 2020, 50 billion Internet of Things (IoTs) devices are expected to be connected to the Internet [1, 2]. By 2025, these connected IoTs are predicted to generate more than 11000 billion dollars economic value [3]. In some domains, the IoTs have already exhibited tremendous potential to bring new opportunities with it, for instance, the wearable devices [4], the smart home [5], the smart industry [6], and the smart meters [7, 8].

With more application scenarios deployed and more devices connected to the Internet, a vital challenge is observed to limit the wider application of IoTs - the battery life. To begin with, it is inconvenient to charge the batteries in some scenarios. Today, most of the wearable devices needs to be charged once a day or several days []. In addition, renewing batteries in huge amount like those IoTs in smart industry [9] or in very distributed area like environmental and infrastructure monitoring [] can very costly, and sometimes unrealistic. What's more, the existence of batteries significantly frustrates application with volume and / or safety concerns, for example in the implantable devices [].

Energy harvesting techniques emergie as one of the promising techniques to solve problems brought by batteries. By harvesting energy from ambient environment, including solar, vibration, RF and thermal energy, the lifetime of IoTs can be eventually infinite. This can essentially extend the application domain of IoTs, for instance, smart parking [], environment monitor [], smart argriculture [], implantable devices [] etc.

While a coin has two sides, the energy harvesting can provide only unstable power supplies, depending on the application scenarios []. In the processor level, various techniques have been proposed to mitigate the inconsistency problem brought by the unstable power supply. These techniques includes software checkpointing under guidance of programming languages or compliers [], and hardware based architectural supports []. Although the processor for signal processing is in same extend mitigated, the energy and time distribution of processing data actually takes only very small part. For instance, the signal processing on processor takes less than 20% in every wake-up cycles, while peripherals initializations contribute to over 80% []. Similar observations can be found in really deployed systems: WISPcam (image sensor + RF + memory) [], Implant cochlear (acoustic encoder + demodulator + sound sensor) [], Mini-satellite (RF + thermometer + gyroscope + magnetometer + gas sensors) []. Optimizating consistency with processor alone can not solve the system level challenges in energy harvesting systems. The peripheral recovery is another critical problem in energy harvesting scenarios and remaining unsolved, and is an important but last step to bring real energy harvesting systems to life[Cite Brandon-SNAPL17,Maeng-OOPSLA17].

Modifying the architecture of all peripherals in a system to support unstable power supply can mitigate the inconsistency problem within the peripherals, but the data transmission between the processor and the peripherals remains unstable. More importantly, peripheral actuations are mostly concurrent atomic operations [10], which requires static checkpointing, conflicting with the dynamic NVPs based solutions. Directly grafting NVP techniques to peripherals to be nonvolatile is rough and inefficient. In addition, considering the amount of various peripherals in the market, it is not time and cost efficient to make all the peripherals nonvolatile.

Targeting at the recovery of a system with multiple peripherals, we propose REMARK, a Reliable and Efficient Multi-device Recovery FrAmewoRK, for hardware and software co-optimization on top of a NVP to address the system level issues.

The major contributions are listed as follows.

- This paper addresses the problem of concurrent execution of peripherals in intermittent power scenarios, by proposing a hybrid checkpointing strategy, combining static checkpointing for peripherals and efficient dynamic checkpointing in NVP to achieve both reliability and efficiency.

- To accelerate the recovery procedure of peripherals, for the first time REMARK provides a peripheral configuration tracking strategy and according methdology for hardware-assisted recovery modules, which simplifies the peripheral recovery through an easy-to-configure interfaces.

- We analyze the design tradeoffs in REMARK to improve the resistance against deadlock and reduce the timing overhead by as much as 36.5%.

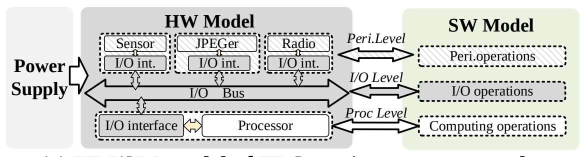
- To verify the architecture, a REMARK-enabled chip is fabricated. Furthermore a real application is evaluated and the results show that REMARK can speed up the data transmission tasks by 13× compared with traditional solutions.

2. MOTIVATION

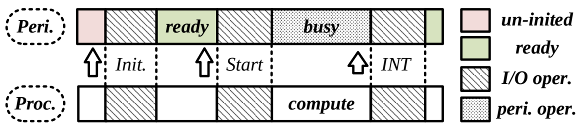
In this section, we present the system model of TPCs in Sec. 2.1 and propose an image sensing application example to show the software model of programs executed on TPC. Based on these two models, we analyze the two software and hardware challenges in Sec. 2.2 and Sec. 2.3.

2.1 TPC System Model

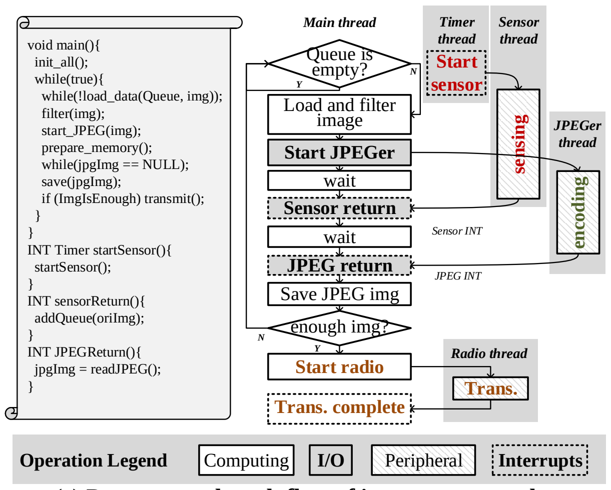
Fig. 1 (a) exhibits the hardware structure of a typical TPC and the corresponding three types of operations on each hardware level. The sensor node contains a processor, multiple peripherals and the I/O bus to connect these hardware component. The processor executes *computing operations* and accesses the peripherals via *I/O operations* on the I/O bus. Peripherals, such as sensor, encoder and radio transceiver, are used to perform specific behaviors, such as sensing, data encoding and transmitting, with *peripheral operations*. These peripherals can be divided into three categories. The input-type collects/receives the external information, such as a sensor or a touch screen. The output-type realizes actuation to outside targets, such as a electromagnetic relay or a wireless transmitter. The compute-type accelerates computing intensive tasks, such as a JPEG encoder. Fig. 1 (b) shows the program model of a concurrent peripheral operation. Before all, a peripheral should be completely initialized via I/O operations by the processor. Then, the processor will write the start command to the peripheral, also via I/O operation, to start the actuation of the peripheral. After that, the peripheral concurrently operates a specific peripheral operation and returns an interrupt to announce the processor when completed. From this procedure, we conclude that, (1) the peripherals requires more preparations to get ready; (2) I/O operations are atomic interacting operations between in the control thread of both processor and peripherals; (3) some



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



(b) Working flow of a typical peripheral.



```
void main(){
  init_all();
  while(true){
    while(!load_data(Queue, img));
    filter(img);
    start_JPEG(img);
    prepare_memory();
    while(jpgImg == NULL);
    save(jpgImg);
    if (ImgIsEnough) transmit();
  }
}
INT Timer startSensor(){
  startSensor();
}
INT sensorReturn(){
  addQueue(oriImg);
}
INT JPEGReturn(){
  jpgImg = readJPEG();
}
```

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

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

peripheral operations are atomic, concurrent operations that only loaded on peripherals.

Fig. 1 (c) shows an image sensing example to explain the model. The processor maintains two threads, the timer thread and the main thread. The timer thread periodically collects and stores the images into buffer. In the main thread, the processor first initializes all the peripherals. Then, in the loop of main thread, the processor processes and encodes the images with a JPEG encoder whenever the buffer is not empty. When the process has stored enough encoded images, the sensor node will transmit all the images back to the server. In this system, the image sensor is an input-type peripheral that collects outside images. The radio transmitter is an output-type peripheral to transmit the collected images to the server for further processes.

In intermittent power supply scenario, such a system can efficiently recover the processor and the computing operations 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** may cause incompleteness and non-determinism. When a sensor loses power, the current collection will be crashed and result is non-deterministic. Even worse, one data lost may jeopardize the integrity of the whole data set in a multi-sensor system. **2. Output-type Failure** may leads to large data loss and affect the application procedure. When the transmitter loses power, the current transmitting package as well as the data in peripheral buffer are lost. Moreover, the subsequent processes in the server are affect since none image arrives. 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** may cause fatal disruptions to the program. When the JPEGer 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 this requirement, two challenges arise. (a) How to design an efficient architecture to recover the execution states in peripherals; (b) How to devise a reliable checkpointing mechanism for a TPC with multiple peripherals. The rest of this paper addresses these two challenges.

2.2 Checkpointing for Concurrent TPC

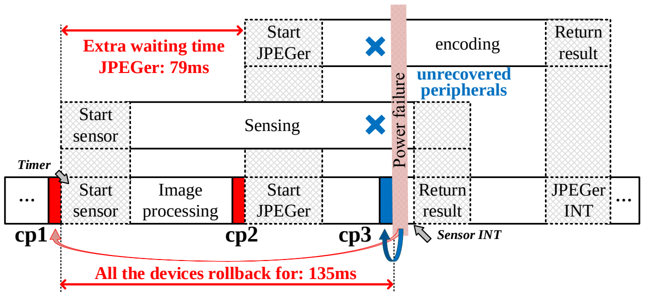
Existing checkpointing strategy faces inefficient and reliability issues in recovering the TPC program where both non-atomic computing operations and atomic concurrent peripherals operations exist.

Static checkpointing strategy is safe for atomic operations but not efficient for non-atomic operations. The red lines in Fig. 2 (a) shows the efficiency issue caused by static checkpointing strategies [11, 12, 13]. Three execution threads are shown, from top to bottom: the JPEGer thread, the sensor thread and the main thread of the processor. The program presets checkpoints in position $cp1$, $cp2$ and $cp3$ to protect the atomic operations. During execution, the program successively starts the sensor and the JPEGer. Once power failure takes place during the I/O and peripheral operations, the program has to rollback to the complete safe position $cp1$ to restart all the devices which leads to extra waiting time and large rollback overheads.

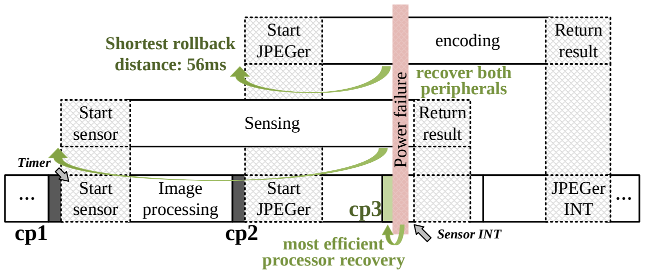
Dynamic checkpointing strategy is efficient for non-atomic computing operations on the processor but not safe for the atomic operations related to peripherals. The blue lines in Fig. 2 (a) shows the reliability issue caused by dynamic checkpointing strategies [14, 15, 16, 17]. When power failure takes, the processor immediately place a checkpoint in position $cp4$. From $cp4$, the main thread can efficiently continue with not rollback overhead, while all the peripheral devices are left un-recovered. Not only the disrupted peripheral operations failed, the peripherals are also left uninitialized which will leads to more errors when the subsequent peripheral-related operations are executed.

Both static and the dynamic checkpointing strategies are needed to achieve safe and efficient recovery. In Fig. 2 (b), if the processor is restored to the interrupt position without rollback and all the peripherals are restarted from the static checkpoints before the atomic operations, the recovery overhead can be reduced by 58.5%. Therefore, an automatic checkpointing tool supporting a dynamic checkpointing strategy for TPCs with multiple peripheral is needed.

2.3 Hardware Needs for Peripheral Recovery



(a) Single-device strategy: extra rollback and consistency problem.



(b) Ideal Multi-device recover: distributed & reliable checkpoints.

Figure 2: Challenges using static or dynamic checkpointing strategies in TPC. A hybrid checkpointing framework is needed.

To support the hybrid checkpointing strategy, we still need hardware modifications based on existing hardware platforms. Two drawbacks are noticed in the existing hardware platform shown in Fig. 3 which contains a nonvolatile processor and sensors.

As is discussed above, NVP is the most efficient dynamic checkpointing strategy which is able to backup and restore (B/R) the processor state within $8\mu s$ once the supply voltage changes across the thresholds. However, both the B/R operations are passively triggered by the voltage detector. NVP do not provide API to support the programmable backup operation for static checkpointing strategy. Moreover, the processor may trigger unwillingly backup when power failure takes place during an atomic I/O operation in the main thread which will cause consistency problems. Therefore, it requires flexible B/R interfaces instead of the passive approach to support the hybrid checkpointing strategy.

On the other hand, checkpointing and recovering a peripheral also poses challenges. Fig. 3 shows the structure of an image sensor, a typical volatile peripheral. The sensor contains a digital part using registers to store the states and data, and the analog part to realize the image collection operation. When power failure takes place during a sensing task, the system has to not only restore the states in the register files, but also restart the analog part to re-execute the uncompleted sensing task to avoid the input-type failure, which is the exact problem beween associate processors and sensors. To efficiently and reliably recover concurrent TPCs, the system should support hybrid checkpointing, peripheral status monitor as well as the peripheral restart modules, which are all missing in existing NVPs.

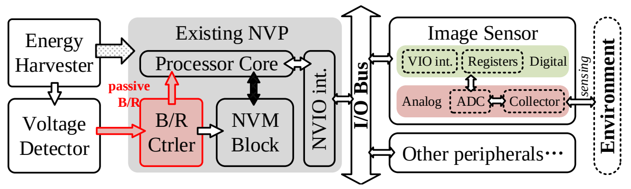


Figure 3: Hardware challenges of multi-device checkpointing. NVP is limited by passive B/R function. Peripherals require efficient reconfigurations and restarts.

3. SYSTEM OVERVIEW OF REMARK

To address the two challenges raised in Sec. 2, this paper proposes REMARK, an efficient and reliable hybrid checkpointing framework.

3.1 Hybrid Checkpointing for TPC

As discussed above, there are three types of operations located in TPC that are executed on different devices and have different atomicity. REMARK divides the program according the operation types into static zones and dynamic zones, as shown in Fig. 4 (a). Static zones are constructed with I/O operations and the dynamic zones are constructed with computing operations. Note that, the source code of a program only contains computing and I/O operations. All the peripheral operations are started by specific I/O operations. Based on this division, REMARK defines three different kinds of checkpoints to recover each type of operations.

Dynamic Checkpoint (DCP). DCPs are used to recover a computing operation by placing checkpoint exactly where power failure takes place to achieve optimal efficiency, as shown in Fig. 4 (b). DCPs do not need to be preset in the program and are enabled only in dynamic zones.

Static Peripheral Recover Checkpoint (SCCP). SCCPs are preset checkpoints that can restore the states of both the processor and the peripheral to re-execute an I/O operation, as shown in Fig. 4 (c). SCCPs are set before each I/O operation in static zones where dynamic checkpoints are disabled.

Static Peripheral Restart Checkpoint (SSCP). SSCPs are preset checkpoints after each I/O operation that starts a peripheral operations. SSCP only recovers and restart the uncompleted peripheral operation and is compatible with the other two checkpoints, as shown in Fig. 4 (d).

In conclusion, with these three kinds of checkpoints, a TPC can recover all the devices and uncompleted operation reliably and efficiently. To realize this strategy, REMARK has to provide hardware and software co-operation.

3.2 HW/SW Architecture Support

To support the hybrid checkpointing strategy, we propose a HW/SW cooperated framework to solve the two drawbacks posed by the inflexibility of NVP and inefficient recovery of peripherals, as discussed in Sec. 2.3. Fig. 5 shows the overview of REMARK framework, which contains an enhanced NVP architecture and an offline program transformer to support peripheral recovering, restarting and the two peripheral-related checkpoints.

The rest of this section presents the overview of each component. The details will be covered in the rest of this paper.

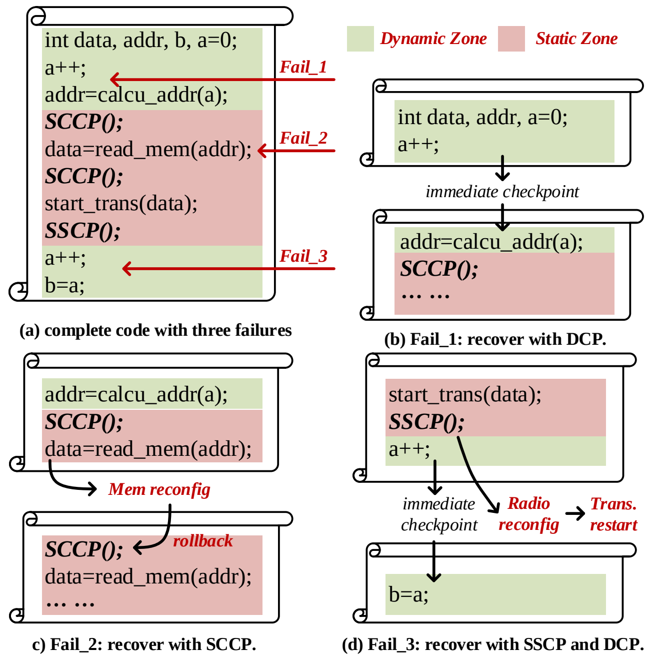


Figure 4: Interactions between processor and peripheral.

Enhanced NVP Hardware Support.
As shown in Fig. 4, on top of existing NVP 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.

To realize flexible control of dynamic checkpointing functions, **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.

Peripheral State Registers (PSRs), **Peripheral Restart Module (PRM)** and **INT Recognizer (IRec)** are used to realize efficient and reliable peripheral recovering and restarting. PSR is a real-time monitor storing the peripheral states to support the efficient peripheral recovering. PRM is used to 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. Moreover, recovering the peripheral interrupts may cause inconsistency problem since the states in the peripherals are ruined by power failures. To guarantee the reliability, **INT Recognizer (IRec)** is designed to place safe checkpoints when outages crash a peripheral interrupt.

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. 6.

Offline Program Transformer.
To preset the static checkpoints in the program, REMARK proposes a program transformer convert a normal program to a recoverable program where safe checkpoints are inserted as shown in Fig.7. APIs are provided to developers to help distinguish operations and divide the program zones.

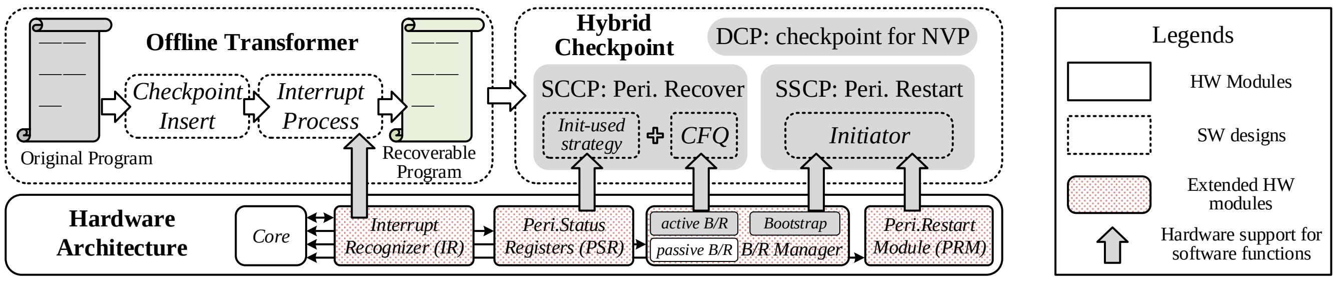


Figure 5: The HW/SW co-designed system diagram of REMARK.

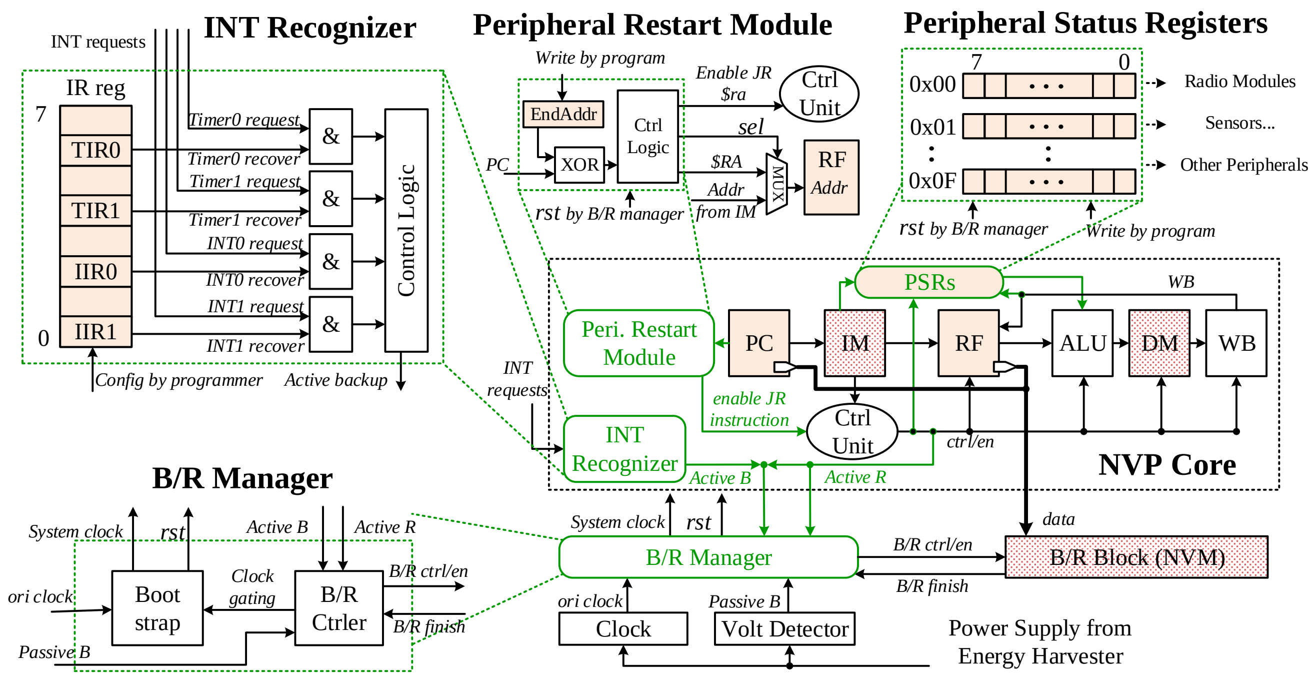
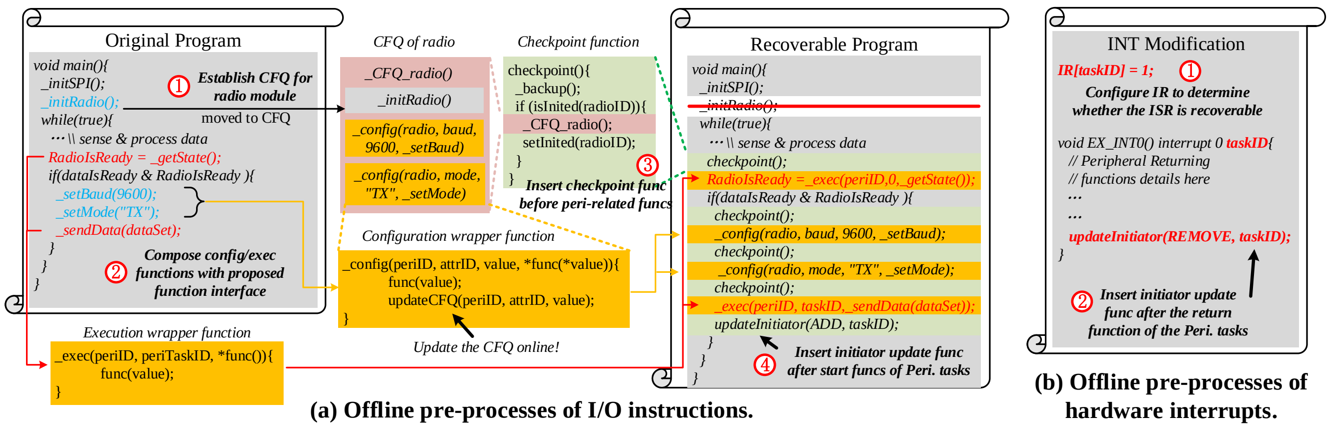


Figure 6: The hardware architecture of REMARK and its main modules.



Original Program
① Establish CFQ for radio module
moved to CFQ
② Compose config/exec functions with proposed function interface
CFQ of radio
Checkpoint function
③ Insert checkpoint func before peri-related funcs
Configuration wrapper function
Update the CFQ online!
Execution wrapper function
Recoverable Program
④ Insert initiator update func after start funcs of Peri. tasks
INT Modification
① Configure IR to determine whether the ISR is recoverable
② Insert initiator update func after the return function of the Peri. tasks

```
void main(){
  _initSPI();
  _initRadio();
  while(true){
    ··· \\ sense & process data
    RadioIsReady = _getState();
    if(dataIsReady & RadioIsReady ){
      _setBaud(9600);
      _setMode("TX");
      _sendData(dataSet);
    }
  }
}
```

```
_CFQ_radio()
  _initRadio()
  _config(radio, baud, 9600, _setBaud)
  _config(radio, mode, "TX", _setMode)
```

```
checkpoint(){
  _backup();
  if (isInited(radioID)){
    _CFQ_radio();
    setInited(radioID);
  }
}
```

```
_config(periID, attrID, value, *func(*value)){
        func(value);
        updateCFQ(periID, attrID, value);
}
```

```
_exec(periID, periTaskID, *func()){
        func(value);
}
```

```
void main(){
  _initSPI();
  ~~_initRadio();~~
  while(true){
    ··· \\ sense & process data
    checkpoint();
    RadioIsReady =_exec(periID,0,_getState());
    if(dataIsReady & RadioIsReady ){
      checkpoint();
      _config(radio, baud, 9600, _setBaud);
      checkpoint();
      _config(radio, mode, "TX", _setMode);
      checkpoint();
      _exec(periID, taskID,_sendData(dataSet));
      updateInitiator(ADD, taskID);
    }
  }
}
```

```
IR[taskID] = 1;

void EX_INT0() interrupt 0 taskID{
  // Peripheral Returning
  // functions details here
  ···
  ···
  updateInitiator(REMOVE, taskID);
}
```

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

(b) Offline pre-processes of hardware interrupts.

Figure 7: The program pre-processes during the software transformation stage.

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 $oper1 = 1/oper1 = 0$.

Together with the support of the hardware framework, REMARK realizes efficient and reliable peripheral recovery and restart. Details are presented in Sec. 5.

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. 6.

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.

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. 6 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. 6 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 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. 6, 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.

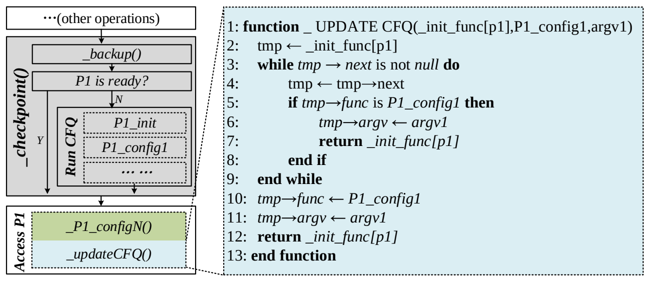
5. SOFTWARE SUPPORT FOR PERIPHERAL RECOVERING

With the enhanced NVP, REMARK will process the original program with the offline program transformer to pre-set safe checkpoints, collaborate with the new hardware modules and ensure realize efficient and reliable peripheral recovering and restarting.

5.1 Peripheral Recover & SCCP

Recovering a peripheral is to restore the configuration states to its internal registers. As is discussed in Sec. 2, neither software-based [13] nor hardware-based [18] 'load-and-backup' recovering approach is usable in TPC. Therefore, REMARK proposes a state tracking and 'init-used' strategy that realizes lightweight, efficient peripheral recovery.

Peripheral State Tracking.
During system execution, the configuration of a peripheral is dynamically updated. Different from the 'load-and-backup' recovering approach, REMARK establishes a **configuration function queue (CFQ)** for each peripheral to track the usage of the configuration instructions, including the initialization instructions, along the execution and record the configuration states in PSRs.. Every time a new configuration instruction is executed, CFQ will be updated to include the latest configuration modification by a pre-inserted *_updateCFQ()* function. Fig. 8 (a) shows the program where peripheral *P1* is configured by *_P1_configN()*. To record this configuration, *_updateCFQ()* function is added after *_P1_configN()*. Fig. 8 (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. In this way, new configurations to an existing



```
function _ UPDATE CFQ(_init_func[p1],P1_config1,argv1)
    tmp ← _init_func[p1]
    while tmp → next is not null do
        tmp ← tmp→next
        if tmp→func is P1_config1 then
            tmp→argv ← argv1
            return _init_func[p1]
        end if
    end while
    tmp→func ← P1_config1
    tmp→argv ← argv1
    return _init_func[p1]
end function
```

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

Figure 8: Program when peripheral *P1* is accessed. *_checkpoint()* is used to configure *P1* and *_updateCFQ()* is used to track the latest configurations.

status will not expand the capacity of CFQ and PSR.

'Initi-used' Reconfiguration Strategy & SCCP.
A peripheral is ready to be called only if it is well configured. Previous works [13, 18] has to 'init-all' the peripherals after power failures in case of subsequent usages. However, if power failure occurs frequently, it leads to large reconfigurations overheads to 'init-all' the peripherals. To reduce the large reconfiguration overheads, REMARK provides an 'init-used' approach, that is, instead of reconfiguring all the peripherals, we only need to reconfigure a peripheral whenever it is accessed. To realizes this strategy, REMARK needs to place a reconfiguration function before each peripheral functions. Considering that the hybrid checkpointing strategy also requires to place SCCPs at the same place, we combine the reconfiguration function with the checkpointing function.

A **checkpoint function**, *_checkpoint()*, is inserted before each I/O operation to realize both the checkpointing and the reconfiguration functions. With *_checkpoint()*, the processor rolls back to reconfigure the peripherals to avoid inconsistency. Fig. 8 (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.

5.2 Peripheral Restart & SSCP

When power failure crashes a peripheral operation, the system has to restart this uncompleted peripheral operation to avoid peripheral failures. REMARK restart the uncompleted peripheral operations from SSCPs with the help of an initiator and PRM.

Initiator & SSCP Pre-placement.
REMARK proposes an **initiator** executed after power recovery to manage the recovery and restart of both the processor and the peripheral operations. Its work flow is shown in Fig. 9. After the system is powered on, Bootstrap selects whether to enter the restart procedure or to start the program

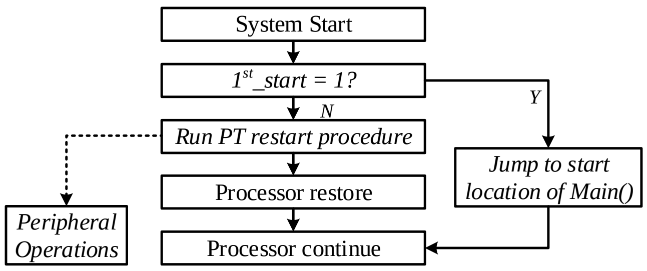


Figure 9: The working flow chart of initiator.

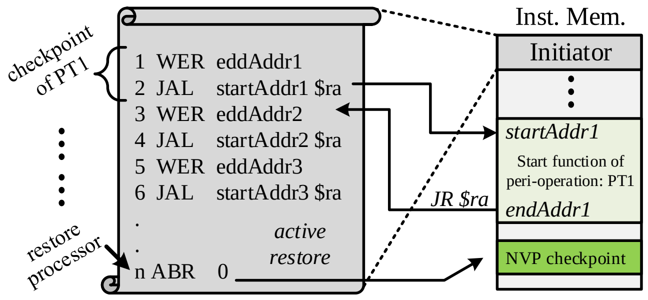


Figure 10: Restart peripheral operations in Initiator.

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.

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 added by *updateInitiator(ADD)* pre-placed after the peripheral operation starting functions.

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)* removes the peripheral checkpoint of the specific peripheral operation.

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, 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.

Handle Hardware Interrupts.
The ISR processing procedure requires safe checkpoints to avoid consistency problem. Generally, an ISR should not be recovered, if it contains processor-peripheral interactions since 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.

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. 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 present the hardware platform in Sec. 6.1. Then, the improvement of the peripheral recovery and the overall performance are presented in Sec. 6.2 and Sec. 6.3, respectively.

6.1 Hardware Platform

Fig. 11 shows a picture of NVP chip and NVnode platform. The structure of NVnode is presented in Fig. 12. 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. 12, 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.

6.2 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 [13]. 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 executing configuration and restart functions.

Fig. 13 (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×.

Fig. 13 (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

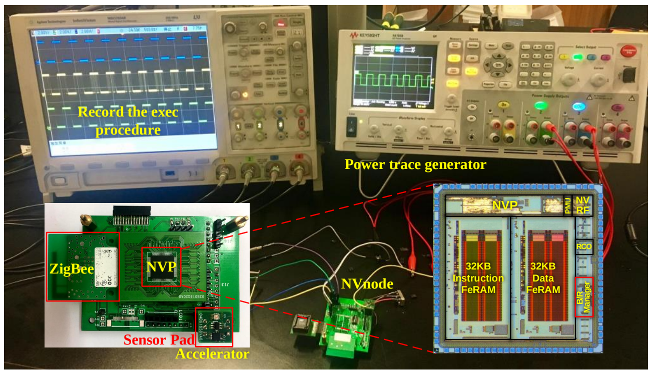


Figure 11: The hardware platform, NVnode. REMARK is implemented with the B/R manager and the NVRF controller in an 8051 NVP chip under $0.13\mu m$ technology.

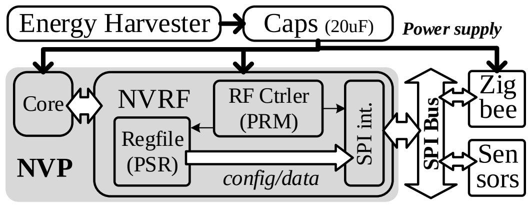
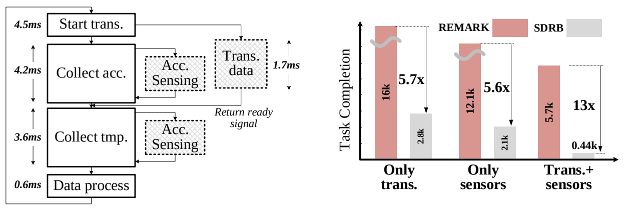


Figure 12: The structure of NVnode. The energy storage is $20\mu F$. NVRF implements the functionality of PSRs and PRM to accelerate the peripheral recovery.

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	$3\mu s$



(a) Work flow of bridge-monitoring. (b) Task completion efficiency.

Figure 14: *bridge-monitoring* application and task completeness comparison between REMARK and SDRB.

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.

6.3 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. 14 (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. 14 (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.

7. 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. 7.1. 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. 7.2 and 7.3. Based on the observations, we summarize three design rules in Sec 7.4.

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	–

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%

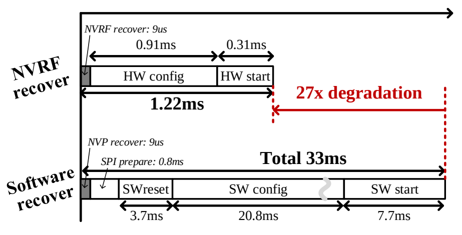
7.1 Experiment Settings

The inputs of NVnodeSim are power traces and benchmark profiles. The power traces are solar power data from an open source database [19]. 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.

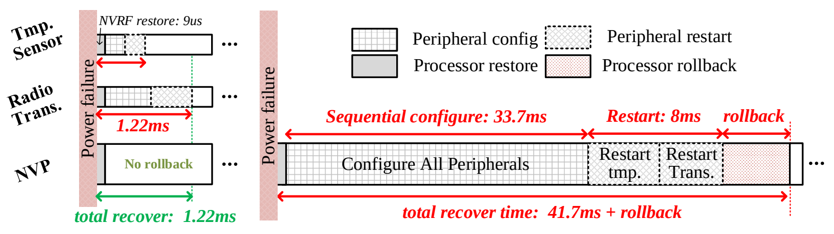
To validate the accuracy of NVnodeSim, Table 4 compares the task execution time between NVnode and NVnodeSim. It shows that NVnodeSim provides reasonable accuracy for architecture-level exploration.

7.2 Evaluation with Multiple Benchmarks

We further evaluate the performance of REMARK with ten additional benchmarks on the simulator. Fig. 15 (a) shows the recovery overhead distribution under ten benchmarks. On average, REMARK reduces total overhead by 64% compared with SDRB. Fig. 15 (b) shows that, the overhead of SDRB rises rapidly when multiple peripherals are included in

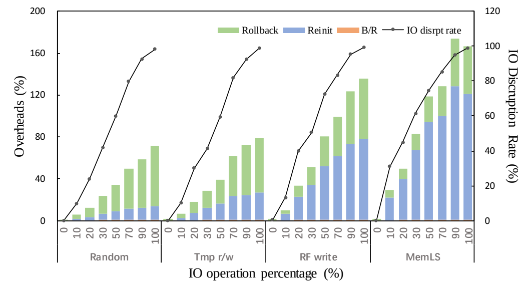


(a) Peripheral recover overhead reduction

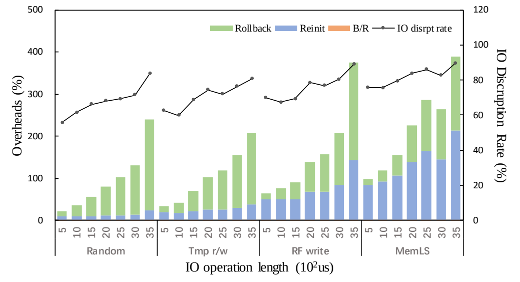


(b) Recover performance improvement by multi-device checkpointing

Figure 13: Recover overhead and task completeness comparison between REMARK and SDRB.



(a) Overhead v.s. IO percentage.



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

Figure 16: The performance overheads with various peripheral operation percentage and length.

the program. This is because the re-initialization overhead 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*.

7.3 Analysis the Peripheral Related Factors

An important observation in Fig. 15 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. 16 (a) shows the overheads and the failure times using 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.

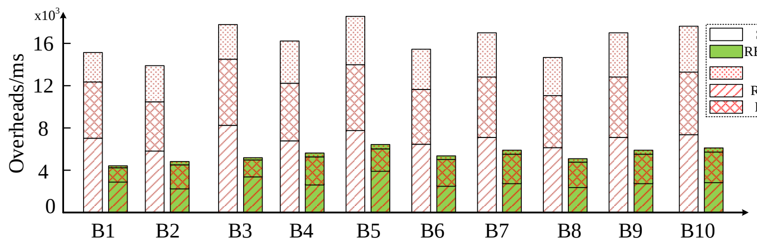
The length of each I/O operation affects the rollback distance during recovery. Fig. 16 (b) shows the effect of lengths on the peripheral disruption times and the recovery overheads. Rollback overhead increases when the length of peripheral operation increases from 0.5*ms* to 3.5*ms*. This is because 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.

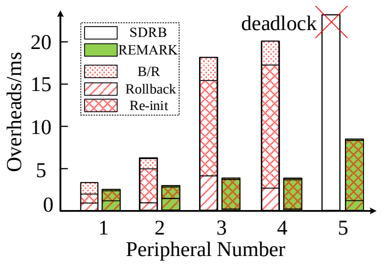
Since the processor and peripherals operate in parallel, the interaction among these devices also affects the performance of REMARK. This part explores the impact of the overlap between I/O and peripheral operations.

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 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, over-



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



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

Figure 15: Recover overheads comparison between REMARK and SDRB.

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

lapping 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*.

7.4 Analysis Conclusion

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

Smaller peripheral occupation. Reduce the percentage of peripheral operations to avoid expensive recoveries.

Shorter peripheral operation. Shorter peripheral operations have smaller overhead. It is benefit to split a long peripheral operation (e.g. large data packet) into shorter ones.

Intensive peripheral operations. Overlapping parallel operations can reduce the stall time of recovery procedures. It is benefit to call peripheral intensively in burst modes.

8. 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 [20, 21]. Bronevetsky et al. propose the application level checkpointing strategy on the shared memory systems to enhance the reliability [22]. 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 [11]. Balsamo et al. propose Hibernus and Hibernus++ using FeRAM and an interrupt-based checkpointing solution to reduce the performance overhead [17, 23, 24]. 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 [13, 25]. 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*) [26, 27, 28].

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 3μs [14]. Bartling et al. propose a non-volatile logic based Cortex-M0 chip with higher performance and lower leakages [29]. Sakimura et al. from NEC propose the non-volatile magnetic flip-flops [30] and a 20MHz non-volatile micro-controller with STT-RAM [31]. Recently, Liu et al. propose an enhanced NVP based on ReRAM which has the highest integration level [15]. In addition, Li et al. propose the non-volatile I/O (NVIO) enabling efficient automatic reconfiguration of I/O interfaces [18].

Compared with software checkpointing strategies, non-volatile 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. [12] discover and model the data inconsistency problem caused by improper rollbacks after power failures. After that, more works [32], [33], [34] an-

alyze 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 [35]. However, reliabliliy and efficiency challenges still exist when adopting NVP in a TPC system with multiple peripherals and interrupts.

9. 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.

10. REFERENCES

[1] A. Nordrum, "Popular internet of things forecast of 50 billion devices by 2020 is outdated," *IEEE Spectrum*, vol. 18, 2016.

[2] C. I. of Everything Connections Counter (Cisco), "How many internet connections are in the world? right. now.." `http://blogs.cisco.com/news/cisco-connections-counter`.

[3] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, "Unlocking the potential of the internet of things," *McKinsey Global Institute*, 2015.

[4] D. Metcalf, S. T. Milliard, M. Gomez, and M. Schwartz, "Wearables and the internet of things for health: wearable, interconnected devices promise more efficient and comprehensive health care," *IEEE pulse*, vol. 7, no. 5, pp. 35–39, 2016.

[5] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 461–472, ACM, 2016.

[6] N. Shariatzadeh, T. Lundholm, L. Lindberg, and G. Sivard, "Integration of digital factory with smart factory based on internet of things," *Procedia CIRP*, vol. 50, pp. 512–517, 2016.

[7] D. Azariadi, V. Tsoutsouras, S. Xydis, and D. Soudris, "Ecg signal analysis and arrhythmia detection on iot wearable medical devices," in *Modern Circuits and Systems Technologies (MOCAST), 2016 5th International Conference on*, pp. 1–4, IEEE, 2016.

[8] M. Haghi, K. Thurow, and R. Stoll, "Wearable devices in medical internet of things: Scientific research and commercially available devices," *Healthcare informatics research*, vol. 23, no. 1, pp. 4–15, 2017.

[9] S. Wang, J. Wan, D. Zhang, D. Li, and C. Zhang, "Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination," *Computer Networks*, vol. 101, pp. 158–168, 2016.

[10] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent Computing: Challenges and Opportunities *," *LIPIcs*, no. 8, pp. 1–8, 2017.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M. F. Chang, S. John, and Y. Xie, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Design Automation Conference*, p. 150, 2015.

[17] 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.

[18] 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.

[19] Measurement and I. D. C. (MIDC), "Solar radiation research laboratory baseline measurement system." `http://www.nrel.gov/midc/srrl_bms/`.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.