Rapid Recovery of Program Execution Under Power Failures for Embedded Systems with NVM

Min Jia, East China Normal University School of Computer Science and Technology, China

Edwin Hsing.-M. Sha, East China Normal University School of Computer Science and Technology, China

Qingfeng Zhuge, East China Normal University School of Computer Science and Technology, China

Rui Xu, East China Normal University School of Computer Science and Technology, China

Shouzhen Gu, East China Normal University Software Engineering Institute, China

Abstract - Embedded systems experience execution interruption due to power failures. After power is switched on, recovering the interrupted program from the initial state can cause negative impact. Some programs are even unrecoverable. To rapid recovery of program execution under power failures, the execution states of checkpoints are backed up by NVM under power failures for embedded systems with NVM. However, frequent checkpoints will shorten the lifetime of the NVM and incur significant write overhead. In this paper, the technique of checkpoint setting triggered by function calls is proposed to reduce the write on NVM. The evaluation results show an average of 99.8% and 80.5% reduction on NVM backup size for stack backup, compared to the log-based method and step-based method. In order to better achieve this, we also propose pseudo-function calls to increase backup points to reduce recovery costs, and exponential incremental call-based backup methods to reduce backup costs in the loop. To further avoid the content on NVM is cluttered and out of NVM, a method to clean the contents on the NVM that are useless for restoration is proposed. Based on aforementioned problems and techniques, the recovery technology is proposed, and the case is used to analyze how to recover rapidly under different power failures.

1. Introduction

From smart homes to smart watches and wearable devices with monitoring have become widespread. However, traditional batteries are not suitable for these devices due to their size and lifetime. Therefore, how to power the currently rapidly developing wearable devices has become a challenge. The emergence of the energy harvesting system solves the power supply problem of wearable devices, but it also brings another challenge. The energy harvest system obtains electric energy from the surrounding environment by sensor [1]. Electricity sources include kinetic energy, electromagnetic radiation, wind and solar energy. However, they are all unstable. In energy harvest system, power failure may occur at any time, between any two lines of code, and within an unpredictable length of time, with almost no warning. The programs may lose data and cannot to continue. When the power is full, the program can only restart form beginning. Frequent power outages and charging causes resource waste and power consume. Worse, in some cases, large programs can never be completed because intermediate results are not saved. Further more, the time and energy cost are horrendous. So, recovery is an issue for these energy harvest system.

Emerging non-volatile memory (NVM) with the advantages of low access latency, high density, byte-addressable, and non-volatile, such as phase change memory (PCM) [2], spin-transfer-torque magnetoresistive random access memory (STT-MRAM) [3] and 3D-XPoint [4], promises to compensate for the volatility and the small capacity of volatile memory. Non-volatile memory technologies can help by allowing near-instantaneous recovery of in-memory state [5]. If the applications checkpoint their state of program to nonvolatile memory before a failure, the program can be forward progress [6]. This technology is called a nonvolatile processor (NVP) [7, 8], which enables the program to continue executing in the case of such an instant on/off. In the NVP, there is a fast NVM. Before each time power failure, the volatile state is copied to the NVM. When the power is restored the next time, the state of processor is copied back, and the program execution can be recovered.

Fig. 1 shows a general system architecture for NVP systems [9]. The energy harvest system obtains electricity from the surrounding environment. Once the voltage detector detects that the voltage is low, it will back up the volatile state into non-volatile memories includes registers and volatile memory space. This way of backing up data is called on-demand backup. Embedded systems usually have a small number of registers, and their values are frequently modified during program execution [9]. Therefore, the value of the volatile register can be designed to be automatically backed up. Registers are proposed to back up with non-volatile flop-flops (NVFFs) [7, 10, 11]. On the other hand, the memory space was much larger than the register size. So, memory space more important to be reduced. However, in most cases, failures occur almost without little warning. Existing studies based on voltage detector to alert and backup the state of program execution. They are not suitable for instant power failures.

In another method, the processor periodically checks the processor state to the non-volatile part. Checkpointing, as shown in the Fig. 2, saves some or all of the state of the program to NVM, without waiting for the power alarm. It allows the program to correctly continue from the checkpoint position. This provides the possibility to rapid recovery of program execution. The more frequent the processor checkpoints, the fewer the rollbacks needed after recovery. The extreme scenario is to back up every instruction, where no rollbacks are ever needed [12]. And then after a failure, the execution of the program can be restored as quickly as possible. Obviously, the more checkpoints are set, the faster the recovery. Meanwhile, it will causes a

Figure 1: System architecture with energy harvesting system powered NVPs [9].

lot of writing to NVM. However, the endurance of NVM is limited and the latency and energy cost of write is high [13–15]. A large number of write operations would reduce the lifetime of NVM and incur significant write overhead. Thus, it is not feasible to set a checkpoint for each instruction that has been executed.

Figure 2: Execute the program under normal or failure or checkpoint

To achieve the goal that writing NVM as less as possible while recovering quickly without voltage warning, this paper uses the *callq* instruction to determine the checkpoints. There are some special cases: 1) If one program without *callq* instruction, the pseudo function call will be inserted in the program. 2) If the function call is in loop, backup is only done when the loop is executed to the power of two. Meanwhile, the content on NVM needs to be cleaned up because there are some useless and useful information mixed on NVM. And due to the NVM is used as on-chip memory, the size of NVM may be small, and not cleaning the useless content will cause NVM overflow. If the clean operation is frequent, the required content may have been removed at the time of recovery. Otherwise, there may be too much content on the NVM to hold subsequent backups. To address this problem, a NVM clean method is proposed to clean the contents on NVM that are useless for recovery. The detailed description for the method of clean is introduced in the subsequent paper. In the target system, the backup operation of the state at each checkpoint is atomic. When power failure is occurred, the backup controller will ensure that the latest state in NVM is all or nothing. The atomicity can guarantee correct recovery. The target system architecture is shown in Fig. 3. SRAM and NVM together serve as on-chip memory for embedded devices. Where the NVM is used to store the program

execution state of checkpoint. After power failure and restart, the latest state of checkpoint would be load from NVM, and this program can be executed from this state rather than beginning. Although the target system architecture of this paper is embedded systems with NVM, the design proposed is applicable to any system with NVM.

Figure 3: Target system architecture

Specifically, this paper makes the following contributions:

- propose an universal backup and recovery system architecture, which is different from the traditional system with a power predictor and power supply from nature.

- propose a checkpoints setting method triggered by a function call. At runtime, after receiving an counter signal and *callq* instruction signal, the program conducts backup.

- propose a method to clean the contents in NVM that are useless for recovery.

- analyze the effectiveness of the proposed scheme under different power failures.

- evaluate the efficacy of the proposed schemes when compared with existing checkpoint backup; analyze the NVM write sizes by the proposed strategies.

The remainder of this paper is organized as follows. Section 2 summarizes related work. Section 3 provides a motivational example to illustrate the main idea of the proposed techniques. Section 4 introduces the proposed scheme in detail. Section 5 analyzes the backup size and the recovery speed for different cases under different power failures. Experimental results are provided in Section 6. Finally, Section 7 concludes this paper.

2. Background and Related Work

In this section, we first introduction related works on energy harvesting systems. Then, the existing work of the checkpoint scheme in the energy harvesting systems are presented.

Energy harvesting systems. Energy harvesting systems make the development of battery-free devices possible. Energy sources derive from solar [16], motion [17], radio frequency (RF), and thermal energy [18, 19] are all potential sources. For example, Taneja et al. [20] design a micro-solar power sensor networks, Wang et al. [11] propose a nonvolatile microprocessor powered with solar under storage-less and converter-less, Paradiso and Feldmeier [22] design a compact, wireless and self-powered pushbutton controller that functions with a piezoelectric conversion mechanism. However, there is an essential challenge with harvested energy. They are all unstable [23]. Because the energy vary with its surroundings and times [24–26]. In traditional cMOS-based processors, all the logic will be lost after shutdown and restart, causing the program to re-execute from the beginning [9]. In order to overcome this issue, many researches have been proposed to deploy NVM in the energy harvesting system in order to store the execution state of the program [27, 28].

The initial exploration into backup for energy harvesting systems focuses on flash [29, 30]. However, flash writes, beyond being energy expensive, are slow and complex. So, another popular choice FeRAM is proposed as NVM for backup in energy harvesting systems [31–34]. In comparison with Flash memory, FRAM boasts extremely low voltage writes, as low as a single volt [35]. Writting to FeRAM are also nearly as fast as writes to SRAM and are bit-wise programmable [36]. Wang et al. [11] developed a compare-and-write ferroelectric nonvolatile flip-flop for energy harvesting applications. Since the volatile logic is copied to the NVM, NVP can record the execution status and resume execution from the interrupted location.

However, due to the high density, high write latency and the limited endurance of NVM, reducing the size of the content to be backed up can greatly improve system performance. Wang et al. [37] proposed the compression-based strategies to

reduce the area of NV registers. Compressing the data before the backup can significantly reduce the sizes of the required backup, but it also brings the overhead of compression and decompression.

Checkpointing. We use the definition of checkpointing by Brenstein et al. [38], who define it as "an activity that writes information to stable storage during normal operation in order to reduce the amount of work [the system] has to do after a failure." In the past few decades, many systems explored the checkpoints of distributed systems [39–41]. In order to enable the programs to be executed continuously, researchers deployed non-volatile memory in energy harvesting devices to save volatile working state at the checkpoint [9, 27, 42, 43]. For example, Mementos [27] is a software system for transiently powered RFID-scale devices. It measures the code written by the user during compilation, the trigger point compares the power supply voltage with the threshold, and triggers the checkpoint to back up the volatile state to flash when the power supply voltage is less than the threshold. However, flash memory has a limited write endurance, and access to flash memory is quite slow. Xie et al. [42] proposed a instruction scheduling to reduce the backup size to NVM. Zhao et al. [9] proposed to reduce the NVM size of the stack backup by flexibly selecting the backup location. Li et al. [43] proposed a checkpoint-aware loop tiling technique. There are mainly two categories of checkpointing schemes: dynamic checkpointing [27] and on-demand checkpointing [9, 42, 43].

But their technologies are all based on the system with battery warning and ignore systems that have no power alerts. This paper solutes the aforementioned issues from the perspective of software for the system without any warning before power failure.

3. Motivation

In this section, we employ an example to illustrate the benefit of exploiting function call as checkpoint to reduce NVM size used for stack backup.

3.1. Motivational Example

Fig. 4(a) presents a sample code in the left, where the *main* function invokes function *multi* to get the product of two numbers. The corresponding pseudo disassembly instructions is shown in the Fig. 4(a). We analyze the stack usage as shown in Fig. 4(b). At the beginning of program execution, the main function is allocated a frame to store the context information of this function. For example, local variables and temporary variables. It can also be seen from the disassembly instructions *sub rsp*, *rsp*, 2 that are opened up space for local variables *a* and *b*. When the program enters function *multi*, *multi* is allocated with another frame. In the Fig. 4(b), the line in the graph shows the first trend from rising to steady, its indicate that the program into *main* function. When the line in the graph shows the second trend from rising to steady, its indicate that the program into *multi* function, and it goes inside the callee function *multi* for execution. Afterwards, the line in the graph shows a trend from steady to downward, it indicates that the callee function *multi* exits and returns to the caller function *main* to execute.

In the case of no power detector, checkpoint backup procedures should be done in advance to make the program forward progress. The most naive way is to use each instruction as a checkpoint [12], which we call log-based method. The log-based backup strategy is common used due to high reliability, because never roll backs. The main idea of the log-based backup is that each instruction is executed, all the volatile logic (including stack and registers, etc.) are written consistently to the NVM. Why we back up whole stack space in each time? Because, the stack size or stack content dynamically change with the execution of instructions. As shown at Fig. 4, some instructions whose execution will affect the stack size or stack contents such as *push*, *pop*, *store*, *call*, *sub*, *return*, and so on. In this way, it writes NVM frequently. Excessively frequent checkpoints can have high overhead. This is not acceptable to NVM. It is equivalent to using NVM as the main memory of the system.

Another naive way to set the step size S, every S instructions, as a checkpoint, which we call step-based method. The main idea of the step-based backup is that every time S instructions are executed, all the volatile logic (including stack and registers, etc.) are written consistently to the NVM. Why we back up whole stack space in each time of step size? The answer is the same as mentioned above. Therefore, although this method reduces the number of backups, the sizes of each backup does not change. The entire stack space needs to be backed up, and there exist a lot of redundancy.

Instead of the log-based backup method and step-based backup method, we propose to backup single stack frame at function call. The function call is the natural checkpoint, because if the function into callee function would openup itself stack frame, then the stack frame of caller function was fixed (here, in particular, the content is fixed). So, when the instruction *call Lmulti* is executed, we back up the stack frame of *main* function. Compared with the log-base backup method, it reduces the number of backups by 46 times and the backup size of 8.2KB. Compared with the backup method of step-base, it reduces the number of backups by 4 and the backup size of 760B bytes.

We presented the size of the stack space for the first 1000 instructions, as shown in Fig.5. The stack behavior in the benchmark also confirms that the stack size changes dynamically as the program executes. As above, when the line in the graph shows a trend from rising to steady, its indicate that the function call is incurred, and it goes inside the callee function for execution. When the line in the graph shows a trend from steady to downward, it indicates that the callee function exits

```
void main( )
{
  int a = 5 ;
  int b = 3 ;
  int c = multi( a , b ) ;
  printf ( "%d" ,c) ;
}

int multi( int x , int y )
{
  int z = 0 ;
  while( y > 0 )
  {
    if( y & 1 == 0 )
    {
      y = y >> 1 ;
      x = x << 1 ;
    }
    else
    {
      z = z + x ;
      y = y >> 1 ;
      x = x << 1 ;
    }
  }
  return z ;
}
```

```
Main:
1   mov   r15, 10000
2   mov   sp, r15
3   sub   sp, sp, #2
4   mov   r2, #5
5   mov   r3, #3
6   str   r3, [r15, #-1]
6   str   r2, [r15, #-2]
7   push  r3
8   push  r2
9   call  Lmulti
          ...
Lmulti:
1   push  r15
2   mov   r15, sp
3   sub   sp, sp, #1
4   push  r2
5   push  r3
6   push  r4
7   push  r5
8   push  r6
9   ldr   r2, [r15, #2]
10  ldr   r3, [r15, #3]
          ...
```

(a)

(b)

Figure 4: Motivation example. (a)The C program and corresponding pseudo disassembly. (b)The stack size of program execution.

and returns to the caller function to execute. As a result, the system can be designed by attaching a smaller NVM. And the wear out of NVM will be reduced. The lifetime of NVM will also increase.

4. Design

The traditional checkpoint design of system based on the energy warnings. When the current voltage below the voltage threshold, the voltage detector sends a warning signal to backup controller to trigger backup. But this design is only suitable for some systems with the power alerts, such as the systems with power provided by the natural environment (i.e. solar energy) which can be detected. In this paper, the schematic of non-volatile processor is shown in Fig. 6. Different from the traditional design, the design proposed is suitable for general systems under power failures without any warning. In this paper, the target system architecture is focused on embedded systems with NVM. The target system architecture is shown in Fig. 3. The basic idea of technology proposed is that the backup operations are triggered by counter signals and function call. And it also cleans up the content in NVM that is useless for recovery by comparing the rbp values in different stack frames. The detailed process is as follows.

The schematic of non-volatile processor is shown in the Fig. 6. In the proposed scheme, the count controller sends the

Figure 5: Dynamic stack sizes along program execution. Benchmarks are from *powerstone* [44].

Figure 6: Schematic for non-volatile processor

"0" or "1" signal to the backup controller. Meanwhile, the instruction register sends the instruction to backup controller. When "1" is issued by count controller and *callq* instruction is issued by instruction register, the backup procedure will be triggered. Once the backup controller is triggered, it sends the stack frame address (the address range between *rbp* to *rsp*) and write signal to the NV controller. The NV controller adds the flag for the stack frame. The flag consists of a monotonically-increasing *epoch_id*, validity of information, *rbp* value and the starting and ending address for stack frame. The current stack frame is tagged valid and then stored to NVM, as well as the tag is stored to NVM with the corresponding stack frame. Make the checkpoint complete. The flag will be retained in NV controller. Then, compare the *rbp* value of current stack frame to the *rbp* value of past stack frame. If the current *rbp* is different from all *rbp* of past stack frame, maintain current state. If there are a stack frame s in previous stack frames that is equal to the *rbp* of the current stack frame s_0, mark stack frames which $epoch\_id \geq s'epoch\_id$ and $< s_0'epoch\_id$ as invalid, as well as remove the corresponding tags in NV controller (Previous Stack frames are marked as invalid if and only if the current checkpoint is complete). NV controller sends a clean signal C to Cleanup controller. Where the address that need be cleaned is included in C. Once cleanup controller is triggered, it cleans the invalid information in NVM. Due to the valid information in NVM are useful for recovery, all of the valid information are loaded from the NVM and sorted by *epoch_id* by Recovery controller during recovery. Considering that registers, such

as special registers and general-purpose registers, are small size and need to be backed up for each checkpoint. The backup of registers can be implemented according to the nonvolatile flip-flops [13]. Therefore, the register backup unit is not shown in the schematic of non-volatile processor. System performance is affected by the cost of NVM write and the time of system recovery. It is vital that to reduce the write cost on NVM, while rapid recovery is satisfied.

4.1. Backup

The location of the checkpoint is very important. The more frequent the checkpoint, the more rapid recovery in program. But it causes amount of cost on NVM write and reduces the life of NVM. Thus, some methods of checkpoint setting are proposed. To avoid power loss during a checkpointing operations, we propose a valid tail. The valid tail writes to the end of checkpoint that includes epoch_id, stack frame address, and valid flag. A checkpoint is complete only if it contains valid tali.

4.1.1. Call- based backup

Use the function call as the checkpoint position. When the program encounters a function call, it will push the address of sub-function return to the stack, and then enter the sub-function for execution. One program code is shown in the top of Fig. 8(a), with multiple embedded function calls. When the main function calls g, the backup controller sends the address of stack frame of main function, the value of rbp of $main$ and write signal to NV controller. Then, the NV controller compare the received rbp and the rbp in NV controller (here, the value of rbp and address is empty in NV controller). A tag with $epoch\_id(1)$ for main function is built in NV controller. The tag and stack frame are stored to NVM together, as shown in top left corner of Fig. 7. The green content represents the tag. And when g calls h, the backup controller sends the address of stack frame of g function, the value of rbp of g and write signal to NV controller. The tag with $epoch\_id(2)$ for g is built in NV controller. The tag and corresponding stack frame are stored to NVM. Then, the NV controller compare the rbp of g and the rbp in NV controller (here, the value of rbp and address in NV controller belong to the main function, the address represents the address of stack frame of main in NVM). The rbp of g and the rbp in NV controller is different, so current state is maintained, the state as shown in top right corner in Fig. 7. The blue content is the newest stack frame.

Another program code is shown in the below of Fig. 8(a). In this program, main calls g, main calls h. When the $main$ function calls g, the process is same as the $main$ call g in preceding program. The state of this process is shown in below left corner of Fig. 7. When $main$ calls h, the backup controller sends the address of stack frame of main function, the value of rbp of $main$ and write signal to NV controller. A tag with $epoch\_id(2)$ for main function is built in NV controller. Then, the NV controller compare the rbp of $epoch\_id(2)$ and the rbp in NV controller(here, the value of rbp and address in NV controller belong to the main function, the address represents the address of stack frame of main in NVM). The the rbp of $epoch\_id(2)$ is same as the rbp of $epoch\_id(1)$ in NV controller. The stack frame with $epoch\_id(1)$ in NVM is marked as invalid and removed the tag from NV controller. The current state is shown in below right corner of Fig. 7. The orange content represents the tag and corresponding stack frame are invalid.

Figure 7: Description of backup process

In both the cases in above, all the registers stores as well as the stack frame to NVM. In those cases, the output signal of the counter is always 1. At the same time, the register unit also needs to be backed up. Because in this paper, an X86-64 machine is used, so the registers include sixteen 64bit general-purpose registers, six 16bit segment registers, one 64bit eflag register and one 64bit instruction pointer register(RIP), totaling 156B. For both of the programs as shown in Fig. 8(a), when executes the "*callq g*" instruction, total 188B content are backed up.

```
void main( ) {
   int sum=1;
   for(int i=0;i<10;i++)
   {
       sum+=i;
   }
   g( );
}
void g( ) {
   h( );
}

void main( ) {
   int sum=1;
   for(int i=0;i<10;i++)
   {
       sum+=i;
   }
   g( );
   h( );
}
```

```
void main( )
{
   int sum=0;
   int i;
   for(i=0;i<3000;i++)
   {
       sum=sum+i;
       g( );
   }
}

void g( )
{

}
```

```
void main( )
{
   int a[10]={1};
   int i;
   for(i=0;i<10;i++)
   {
       int j;
       for(j=0;i<10;i++)
       {
           a[i]=a[j]+a[i];
       }
       pseudo_g( );
   }
}

void pseudo_g( )
{

}
```

```
void main( )
{
   int a[10]={1};
   int b[5]={5};
   a[1]=b[2]+1;
   pseudo_g( );
   a[2]=b[4]+a[1];
   a[3]=b[2]+a[2];
   a[4]=a[3]+a[1];
   a[5]=b[3]+1;
   a[6]=a[5]+b[4];
   pseudo_g( );
   a[7]=b[2]+a[1];
   a[8]=b[4]+a[2];
   a[9]=b[4]+1;
}
void pseudo_g( )
{

}
```

(a)   (b)   (c)   (d)

Figure 8: Design Example

4.1.2. Exponential incremental call-based backup

If the function call in a loop, as shown in Fig. 8(b), each function call in the loop triggers a backup mechanism, which will cause multiple writes to the NVM. Thus, an exponential incremental call-based backup approach is proposed. We assume the base is 2. That is, the backup controller can be triggered at num =1, 2, 4, 8, 16... and the last loop (the "num" represents loop time). The counter will output 1 only when the loop is executed to the power of 2, otherwise it will output 0. Therefore, when it is not a power of 2, even if the *callq* instruction is issued, the backup controller is disable to triggered. When it is a power of 2, and the *callq* instruction is issued, the backup controller is able to triggered based on the call-based backup method policy. In this way, the program code as shown in Fig. 8(b) backs up a total of 3.25KB. If apply the call-based backup method in every cycle, will be backed up 750KB. It is obviously that the exponential incremental call-based backup method can reduce the backup content by nearly 230x. The backup operations and content are same as call-backup method.

4.1.3. Pseudo call-based backup

If the program has not a function call, or a large number of instructions are executed still does not encounter a function call, the pseudo-function call will be inserted to as the checkpoint position before the program compilation. The basic idea follows the call-based backup and the exponential incremental call-based backup, the detailed backup mechanisms are described as the part of call-based backup method. The inserts mechanism based on the follow. If there is only one layer of loop and the number of loops is relatively small, then the pseudo function is inserted outside the loop. This case is similar to the main function of the code block shown in Fig. 8(b), and the g function is equivalent to the inserted pseudo function call. If there is only one layer of loop and the number of loops is relatively large, then the pseudo function is inserted inside the loop. This case is similar to the main function of the code block shown in Fig. 8(b), and the *g* function is equivalent to the inserted pseudo function call. If there are multiple layers of loops, it is added inside the outermost loop. This method is shown in

Fig. 8(c). In this case, our method backs up 1.5B. If the program has not a function call, or a large number of continuous instructions does not exist a function call, we set a threshold (T), and insert a pseudo function call when the threshold is reached. For the code shown in Fig. 8(d), we assume that T is 20 in this example. In practice, the value of T can be designed according to actual needs. Therefore, under this setting, our method needs to back up 640B.

The foregoing shows that the proposed approaches to reducing write NVM is effective.

4.2. NVM Cleaning

The NVM cleanup is very important. Because, there are many invalid information in NVM and NVM overflow would incur if never cleanup based on backup technology proposed. So if there are many data chunk on NVM (some are valid, and some are invalid), the invalid blocks not only occupy the space of NVM, but also affect the performance of recovery. So, we should clean the invalid chunk in NVM. We ignore the write operations caused by NVM cleanup.

The schematic of the NVM cleaning process is as shown in Fig. 6. In the proposed scheme, the cleanup procedure is triggered by the clean signal (C). The clean signal includes invalid address of stack frame and write signal. Upon the clean signal is issued by NV controller, the cleanup controller was triggered to clean up the invalid stack frame in NVM. We analysis clean process on the program as shown in Fig. 8(a). When *main* calls *h*, the *rbp* of *epoch_id*(2) is same as the *rbp* in NV controller (because the rbp in NVC belongs to main function). Stack frame of *epoch_id*(1) is marked as invalid, as well as remove the corresponding tag in NV controller. NV controller sends a clean signal to Cleanup controller. The cleanup controller is triggered to clean up the invalid address of main stack frame with *epoch_id*(1), and then, the address is released.

4.3. Recovery

Depending on the backup policy and NVM cleanup, the content on NVM can ensure the recovery is correctly. The Fig. 6 shows the recovery schematic. When the power is restored, it will send a signal to the recovery controller. The recovery controller is triggered and sends the read signal to copy the data from the NVM to SRAM. In this procedure, the recovery controller reads the all stack frame on the NVM. They are valid due to the cleanup mechanism. Then sorting by the marked *epoch_id* of each stack frame to get the correct stack space when recovery. If the failure occurs in the cleanup process, the recovery controller reads the all stack frame on the NVM during recovery, but some are invalid. So, the recovery controller not only sorts the stack frame by the *epoch_id*, but also ignores the invalid stack frame. The tag of invalid stack frame will be send to NV controller by recovery controller to execute cleanup operation. The recovery of the register is implemented by nonvolatile flip-flops [10].

5. Case Studies

In this section, an example is used to analyze the backup and recovery under different power failures. The failures would occur in backup process, cleanup process and program execution process, respectively.

We analysis the C program and its corresponding pseudo assembly code are shown in Fig. 9.

It includes all the methods mentioned in the section 4.1 part(we set the T=300, S=10). We analyze the performance of the call-based backup method and exponential incremental call-backup method.

5.1. Program execution process

By default, the failures occurs in program execution process, the backup process and cleanup process are done normally and correctly. We assume that in the following analysis, failures of program execution process all occur when a certain instruction is about to be executed. Because the failure during the execution of the instruction is equivalent to the instruction not being executed, the failure after the execution of the instruction is equivalent to the failure when the next instruction is about to be executed. In this process, we also compare with log-backup under different failures cases.

5.1.1. Case1

The failure occurred at the location identified by *i*, and both the call-based backup method and exponential incremental call-backup method can recover from the "*push %rbp*" instruction in the *pseudo* function. In this case, 4.2KB contents are backed up and 218 instructions are rolled back. But for the log-based backup method, there are 1231.64KB contents to written NVM. The step-based backup method, there are 123.6KB contents needed to backup and 10 instructions to roll back.

5.1.2. Case2

The failure occurred at the location identified by *b* in the 500th loop. For the call-based backup method, the latest checkpoint is the instruction of "*callq pseudo*" in 499th loop, and then the program restart from "*push %rbp*" in 499th call *pseudo*. There are only 11 instructions to rollback, but 2MB data to store. For the method of exponential incremental call-based backup, it needs to be executed from the 256th call to the *pseudo* function, causing 5114 instruction rollbacks. But compared to the call-based backup method, the exponential incremental call-based backup method only needs to back up 41.7KB. The log-based backup method needs to backup 40.05MB to ensure no rollback. The step-based backup method, there are 2253.9KB contents needed to backup and 18 instructions to roll back.

Figure 9: The example of case study.

5.1.3. Case3

The failure occurred at the location identified by c in the 128th loop in the main function. Both the call-based backup method and the exponential incremental call-based backup method can start execution of the "*push %rbp*" instruction in the *pseudo_g* function and only causes 2 instructions to rollback. But, compared the call-based backup method to exponential incremental call-based backup method, the exponential incremental call-based backup method is better, because only needs to store 37.55KB. But, the call-based backup method needs to write 535.17KB. However, the log-based backup method backs up 11.81MB to ensure no rollback. The step-based backup method, there are 733.48KB contents needed to backup and 12 instructions to roll back.

5.1.4. Case4

The failure occurred at the location identified by d in the 513th loop in the main function. Both the call-based backup method and exponential incremental call-backup, the program needs to be executed from "*push %rbp*" instruction in 512th *pseudo* called. Because the 512th call to the *pseudo* function is the checkpoint for these method and only caused the rollback of 20 instructions. in this case, the call-based backup method needs to back up 2.09MB, while the exponential incremental call-based backup method only needs to back up 49KB. However, the log-based backup method backs up 41.09MB to ensure no rollback. The step-based backup method, there are 2311.08KB contents needed to backup and 9 instructions to roll back.

5.1.5. Case5

The failure occurred at the location identified by e. Both the call-based backup method and the incremental call-based backup method, it is necessary to start execution from the "*push %rbp*" instruction of the sum function, which only causes the rollback of 3 instructions. The incremental call-based backup method only needs to back up 50.06KB, while the call-based backup method needs to back up 4.07MB. However, the log-based backup method backs up 78.11MB to ensure no rollback. The step-based backup method, there are 8.2MB contents needed to backup and 7 instructions to roll back.

5.1.6. Case6

The failure occurred at the location identified by f in the 5th loop in the *sum* function. Both the call-based backup method and the incremental call-based backup method, it is necessary to start execution from the "*push %rbp*" instruction of the sum function. Both methods caused a rollback of 79 instructions, but the exponential incremental call-based backup method only needs to back up 50.06KB, while the call-based backup method needs to back up 4.07MB bytes of content. However, the log-based backup method backs up 78.42MB to ensure no rollback. The step-based backup method, there are 8.22MB contents

needed to backup and 15 instructions to roll back.

5.1.7. Case7

The failure occurred at the location identified by *h*. Both call-based backup method and the exponential incremental call-based backup method, can start execution the *push %rbp*" instruction of the *pseudo* in *sum* function, causing 5 instructions to roll back. The call-based backup method needs to back up 4.08MB, while the exponential incremental call-based backup method only needs to back up 59.29KB. However, the log-based backup method backs up 78.74MB to ensure no rollback. The step-based backup method, there are 8.25MB contents needed to backup and 8 instructions to roll back.

5.1.8. Case8

The failure occurred at the location identified by *g*. Both call-based backup method and the exponential incremental call-based backup method, can start execution the *push %rbp*" instruction of the *pseudo* in *sum* function, causing 11 instructions to roll back. The call-based backup method needs to back up 4.08MB, while the exponential incremental call-based backup method only needs to back up 59.29KB. However, the log-based backup method backs up 78.77MB to ensure no rollback. The step-based backup method, there are 8.25MB contents needed to backup and 6 instructions to roll back.

The Fig 10 illustrates the size of the backups and the number of instruction rollbacks for each method.

(a)

(b)

Figure 10: The case studies' presentation of backup size and number of instruction rollback.

5.2. Backup process

If the backup process is complete, the current checkpoint is valid. If the backup process is incomplete, the valid checkpoint is the lasted checkpoint.

5.2.1. Case1

The failure occurred at the location identified by i. If the backup process is complete, so, upon the power restore, both the call-based backup and the exponential incremental call-based backup method could start at the instruction of "*push %rbp*" in *pseudo*. If the backup process is incomplete, this backup operation is invalid. In the case, both the call-based backup and the exponential incremental call-based backup method could be re-executed the main.

5.2.2. Case2

The failure occurred at the location identified by d in the 9th loop. If the backup process is complete, so, upon the power restore, the call-based backup method could be start at the instruction of "*push %rbp*" in *pseudo* of 9th loop. If the backup process does is incomplete, this backup operation is invalid. So, upon the power restore, the call-based backup method could start at the instruction of "*push %rbp*" in *pseudo* of 8th cycle. And the exponential incremental call-based backup method could always start at the instruction of "*push %rbp*" in *pseudo* of 8 loop. Because the "*callq pseudo*" of 9th loop is not the checkpoint for the exponential incremental call-based backup method.

5.2.3. Case3

The failure occurred at the location identified by j in the main function. If the backup process is complete, so, upon the power restore, both the call-based backup method and the exponential incremental call-backup method could be start at the instruction of "*push %rbp*" in sum function. If the backup process is incomplete, this backup operation is not valid. Upon the power restore, both the call-based backup method and the exponential incremental call-based backup method could start at the instruction of "*push %rbp*" in *pseudo* of 999th loop. Because the "*callq pseudo*" of 999th loop is the checkpoint for the two method.

5.2.4. Case4

The failure occurred at the location identified by k in the sum function. if the backup process is complete, so, upon the power restore, both the call-based backup method and the exponential incremental call-based backup method could be start at the instruction of "*push %rbp*" in *pseudo* function. If the backup process is incomplete, this backup operation is not valid. Upon the power restore, both the call-based backup method the exponential incremental call-based backup method could start at the instruction of "*push %rbp*" in sum function.

In summary, when there is a power failure during the checkpointing, the program can resume working from the last complete checkpoint position stored in NVM. If the current checkpoint has been stored in NVM completely, the last checkpoint position is the current call function. If it is incomplete, thus the last checkpoint is the latest function call as the checkpoint.

5.3. Cleanup process

By default, the cleanup process are based on the correct backup process.

5.3.1. Case1

The failure occurred at the location identified by i. Because there are no invalid information in NVM, the cleanup process would not be triggered.

5.3.2. Case2

The failure occurred at the location identified by d in the 9th loop. For the call-based backup method, regardless of whether the failure occurs before, during, or after cleanup process, when power restore, the call-based backup method could be start at the instruction of "*push %rbp*" in *pseudo* of 9th loop. Because in the 9th backup, the stack frame saved when the pseudo function is called for the 8th backup has been marked as invalid. Thus, even if it is not cleaned, it will be removed according to the flag, without affecting the correct recovery of the program. And the exponential incremental call-based backup method could always start at the instruction of "*push %rbp*" in *pseudo* of 8 loop. Because the "*callq pseudo*" of 9th cycle is not the checkpoint for the exponential incremental call-based backup method, so the cleanup process would not be triggered.

5.3.3. Case3

The failure occurred at the location identified by j in the *main* function. Upon the power restore, both the call-based backup and the exponential incremental call-based backup could be start at the instruction of "*push %rbp*" in sum function.

Table 1
Parameters value and policies of each backup method

backup method	parameters	policy
step-based	S=20	
pseudo exponential incremental call-based	T=20	loop times less than or equal to 10, pseudo call is outside loop
	T=20	loop times more than 10, pseudo call is outside loop

5.3.4. Case4

The failure occurred at the location identified by k in the sum function. Upon the power restore, both the call-based backup method and the exponential incremental call-based backup method could be start at the instruction of "*push %rbp*" in *pseudo* function.

In summary, when there is a power failure in the cleanup process, the program can resume working from the last checkpoint position stored in NVM.

Regarding the method of inserting pseudo-functions in multi-layer loops proposed in the section 4.1.3, there is no specific cases study due to space limited. But it is easy to understand that this method can greatly reduce the write operation to NVM.

6. Experiments

In this section, we present the experimental evaluation to demonstrate the efficiency of the proposed method.

6.1. Experiment Setup

The experiments are conducted with a customized processor simulator.

We conducted the experiments on a set of benchmarks, such as *adpcm*, *blit*, *bcnt*, *qurt*, and *crc*. These benchmarks are from the *powerstone* benchmark suite [44]. The assembly code generated by the GCC compiler for each benchmark is used as input. As described in section 4, each method is influenced by the setting of various parameters and policies. Table 1 summarizes the parameters and policies of each method. In Table 1, S=20 indicates that every 20 instructions are statically taken as a backup point, T=20 indicates that a pseudo-function call is inserted every 20 instructions when there are no loops and function calls. When the times of loop is less than or equal to 10, and there are no function calls inside or outside the loop, we insert a pseudo function call outside the loop as a backup point. When the times of loop is more than 10, and there are no function calls inside the loop, we insert a pseudo function call outside the loop as a backup point. In the following experiments, we have added the pseudo function calls to all the pseudo-function call methods with these policies. After the position of backup points is fixed, 50 failure points are randomly selected in each program to analyze the relationship between the size of backup and the number of rollback instructions at the time of failure.

6.2. Evaluation Results

Fig. 11 shows the analyzed current stack frame size upon each instruction in representative benchmarks. Here, only 7000 instructions are described, in order to obviously show the fluctuation trajectory. The fluctuation verifies the feasibility of using function calls as backup locations. The proposed method only needs to back up the stack frame (that is, the stack frame of the caller function) when the *call* instruction is encountered. The traditional log-based and step-based method not only need to back up the stack frame when the checkpoint instruction is encountered, but also the stack frame of other caller functions until the *main* function is backed up.

Fig. 12 describes step-based, call-based, exponential incremental call-based backup (called incre-call in figure) and pseudo exponential incremental call-based backup (called pseuso-incre-call in figure) the backup content size normalized to the log-based method for tested benchmarks. It shows that the call-based scheme delivers 98.8% backup content size reduction on average, the exponential incremental call-based scheme delivers 99.6% backup content size reduction on average, the pseudo exponential incremental call-based scheme delivers 99.6% backup content size reduction on average, when compared with conventional log-based backup. It also shows that the call-based scheme delivers 80.5% back up content size reduction on average, the exponential incremental call-based scheme delivers 91.9% backup content size reduction on average, the pseudo exponential incremental call-based scheme delivers 91.4% backup content size reduction on average, when compared with conventional step-based backup.

Fig. 13 describes call-based, exponential incremental call-based backup (called incre-call in figure) and pseudo exponential incremental call-based backup (called pseuso-incre-call in figure) the number of rollback instruction compared to the step-based method for tested benchmarks. It shows that the call-based method delivers 5.36X rollback of instruction on average, the exponential incremental call-based method delivers 8.7X rollback of instruction on average, the pseudo exponential incremental call-based method delivers 3.81X rollback of instruction on average. Although, the best approach in terms of the number of instruction rollbacks is log-based, the backup content is huge, it is easy to corrupt the NVM, and our method is better tradeoff.

Figure 11: Stack frame of each instruction for benchmarks *adpcm*, *crc*, and *fir*.

7. Conclusion

This paper aims to reduce write overheads to NVM during backup while recovering rapidly. Different from conventional log-based and step-based method, we propose to use function calls as checkpoints, and through offline analysis, leading to a smaller backup size for NVM. The evaluation shows 99.8% and 80.5% NVM backup size reduction on average. In order to better achieve this, we also proposed inserting pseudo-function calls to increase backup points to reduce recovery costs, and exponential incremental call-based backup methods to reduce backup costs in the loop. However, in this paper, we are disable to solution the recursive function. In future work, we will consider the issue.

References

[1] S. Sudevalayam and P. Kulkarni. Energy harvesting sensor nodes: Survey and implications. IEEE Communications Surveys Tutorials 13 (3) (2011) 443–461.

[2] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy effi-cient main memory using phase change memory technology. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). Association for Computing Machinery, 2009, pp. 14–23.

[3] Y. Chen, W.-F. Wong, H. Li, and C.-K. Koh, "Processor caches built using multi-level spin-transfer torque ram cells," in Proc. Int. Symp. Low Power Electron. Design (ISLPED), 2011, pp. 73–78.

Figure 12: The proposed call-based and optimized call-based backup sizes are normalized to the log-based backup size design of each benchmark.

Figure 13: The proposed call-based and optimized call-based backup compared to the step-based number of rollback instruction design of each benchmark.

[4] Intel and Micron. Intel and micron produce breakthrough memory technology, 2015. http:/newsroom:intel:comcommunityintel_newsroomblog20150728 intel-and-micron-produce-breakthrough-memory-technology.

[5] D. Narayanan and O. Hodson, "Whole-system persistence,"Acm Sigplan Notices 47 (4) (2012) 401–410.

[6] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power M*core architecture," in Proc. IEEE Power Driven Microarchitect. Workshop, 1998, pp. 145–150.

[7] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M.-F. Chiang, Y. Yan, B. Sai, and H. Yang, "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops," in Proc. Eur. Solid-State Circuits Conf., 2012, pp. 149–152.

[8] X. Sheng, Y. Wang, Y. Liu, and H. Yang, "SPaC: A segment-based parallel compression for backup acceleration in nonvolatile processors," in Proc. Design Autom. Test Europe Conf. Exhibit. (DATE), 2013, pp. 865–868.

[9] M. Zhao et al., "Stack-size sensitive on-chip memory backup for selfpowered nonvolatile processors," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. 36 (11) (2017) 1804–1816.

[10] N. Sakimura, T. Sugibayashi, R.Nebashi, and N.Kasai, "Nonvolatile magnetic flip-flop for standby-power-free SoCs," in Proc. IEEE CustomIntegr. Circuits Conf., 2008, pp. 355–358.

[11] J. Wang, Y. Liu, H. Yang, and H. Wang, "A compare-and-write ferroelectric nonvolatile flip-flop for energy-harvesting applications," in Proc. Int. Conf. Green Circuits Syst., 2010, pp. 646–650.

[12] K. Ma et al., "Nonvolatile Processor Architecture Exploration for Energy-Harvesting Applications," in IEEE Micro 35 (5) (2015) 32-40. doi:

10.1109/MM.2015.88.

[13] Arulraj J., Pavlo A., Dulloor S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data , 2015, pp. 707–722.

[14] Kannan S., Arpaci-Dusseau A. C., Arpaci-Dusseau R. H., Wang Y., XU J., and Palani G. Designing a true direct-access file system with devfs. In *FAST* (2018), 2018, pp. 241.

[15] Zhang, Y., Yang, J., Memaripour, A., and Swanson, S. Mojim: A reliable and highly-available non-volatile memory system. Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, 2015, pp. 3–18.

[16] Jay Taneja, Jaein Jeong, and David Culler. Design, modeling, and capacity planning formicro-solar power sensor networks. In Proceedings of the 7th International Conference on Information Processing in Sensor Networks. 2018, pp. 407–418.

[17] Nathan S. Shenck and Joseph A. Paradiso. Energy scavenging with shoe-mounted piezoelectrics. IEEE Micro 21 (3) (2001) 30–42.

[18] Qadeer A. Khan and Sarvesh J. Bang. Energy Harvesting for Self Powered Wearable Health Monitoring System. Technical Report. Oregon State University, 2009.

[19] Vladimir Leonov. Thermoelectric energy harvesting of human body heat for wearable sensors. IEEE Sensors Journal 13 (6) (2013) 2284–2291.

[20] Jay Taneja, Jaein Jeong, and David Culler. Design, modeling, and capacity planning formicro-solar power sensor networks. In Proceedings of the 7th International Conference on Information Processing in Sensor Networks, 2007, pp. 407–418.

[21] Cong Wang, Naehyuck Chang, Younghyun Kim, Sangyoung Park, Yongpan Liu, Hyung Gyu Lee, Rong Luo, and Huazhong Yang. Storage-less and converter-less maximum power point tracking of photovoltaic cells for a nonvolatile microprocessor. In Proceedings of the 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC'14), 2004, pp. 379–384.

[22] J. A. Paradiso and M. Feldmeier. A compact, wireless, self-powered pushbutton controller. In Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp'01), 2001, pp. 299–304.

[23] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. Power management in energy harvesting sensor networks. ACM Transactions on Embedded Computing Systems 6 (4) (2007) 32-es.

[24] C. Park and P. H. Chou, "Ambimax: Autonomous energy harvesting platform for multi-supply wireless sensor nodes," in Proc. IEEE Commun. Soc. Sensor Ad Hoc Commun. Netw. 1 (2006) 168–177.

[25] J. Taneja, J. Jeong, and D. Culler, "Design, modeling, and capacity planning for micro-solar power sensor networks," in Proc. Int. Conf. Inf. Process. Sensor Netw., 2008, pp. 407–418.

[26] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava, "Design considerations for solar energy harvesting wireless embedded systems," in Proc. Int. Symp. Inf. Process. Sensor Netw., 2005, pp. 457–462.

[27] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for longrunning computation on RFID-scale devices," in Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS), 2011, pp. 159–170.

[28] H. Jayakumar, A. Raha and V. Raghunathan, "QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers," 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, 2014, pp. 330-335, doi: 10.1109/VLSID.2014.63.

[29] A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Idetic: A highlevel synthesis approach for enabling long computations on transientlypowered ASICs," in Proc. Pervasive Comput. Commun. Conf. (PerCom), 2013, pp. 216–224.

[30] B. Ransford, S. S. Clark, M. Salajegheh, and K. Fu, "Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling." in Proc. HotPower, 2008, pp. 5.

[31] S. Ducharme, T. J. Reece, C. M. Othon, and R. K. Rannow, "Ferroelectric polymer langmuir-blodgett films for nonvolatile memory applications," IEEE Trans. Device Mater. Rel. 5 (4) (2005) 720–735.

[32] Y. Horii et al., "4 Mbit embedded FRAM for high performance system on chip (SoC) with large switching charge, reliable retention and high imprint resistance," in Proc. Int. Electron Devices Meeting, 2002, pp. 539–542.

[33] H. Nakamoto et al., "A passive UHF RF identification CMOS Tag IC using ferroelectric RAM in 0.35-μm technology," IEEE J. Solid-State Circuits 42 (1) (2007) 101–110.

[34] H. Shiga et al., "A 1.6 gb/s DDR2 128 Mb chain FeRAM with scalable octal bitline and sensing schemes," IEEE J. Solid-State Circuits 45 (1) (2010) 142–152.

[35] M. Qazi, M. Clinton, S. Bartling and A. P. Chandrakasan, "A Low-Voltage 1 Mb FRAM in 0.13 μm CMOS Featuring Time-to-Digital Sensing for Expanded Operating Margin," in IEEE Journal of Solid-State Circuits 47 (1) (2012) 141-150. 2012, doi: 10.1109/JSSC.2011.2164732.

[36] J. V. D. Woude and M. Hicks, "Intermittent computation without hardwaresupport or programmer intervention," in12th USENIX Symposium on Op-erating Systems Design and Implementation (OSDI 16), 2016, pp. 17–32.

[37] Y.Wang et al., "A compression-based area-efficient recovery architecture for nonvolatile processors," in Proc. Design Autom. Test Europe Conf. Exhibit. (DATE), 2012, pp. 1519–1524.

[38] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987. ISBN0-201-10715-5.

[39] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. In Proceedings of 1986 ACM Fall Joint Computer Conference (ACM '86), 1986, pp. 1150–1158.

[40] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—a hunter of idle workstations. In Proc. 8th Int'l Conference on Distributed Computing Systems (ICDCS '88), 1988, pp. 104–111.

[41] J. A. McDermid. Checkpointing and error recovery in distributed systems. In Proc. 2nd Int'l Conference on Distributed Computing Systems (ICDCS '81), 1981, pp. 271–282.

[42] M. Xie, C. Pan, J.Hu, C. J. Xue, and Q. Zhuge, "Non-volatile registers aware instruction selection for embedded systems," in Proc. IEEE 20th Int. Conf. Embedded Real-Time Comput. Syst.Appl., 2014, pp. 1–9.

[43] Li, F, Qiu, K., Zhao, M., Hu, J., Liu, Y., Guan, Y., & Xue, C. J. Checkpointing-aware loop tiling for energy harvesting powered nonvolatile processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38 (1) (2018) 15-28.

[44] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power M*core architecture," in Proc. IEEE Power Driven Microarchitect. Workshop, 1998, pp. 145–150.