Fixing the Broken Time Machine: Consistency-Aware Checkpointing for Energy Harvesting Powered Non-Volatile Processor*

Mimi Xie[1], Mengying Zhao[2], Chen Pan[1], Jingtong Hu[1], Yongpan Liu[3], Chun Jason Xue[2]
[1]School of Electrical and Computer Engineering, Oklahoma State University, OK, USA
[2]Department of Computer Science, City University of Hong Kong, Hong Kong
[3]Department of Electronic Engineering, Tsinghua University, Beijing, China

ABSTRACT

Energy harvesting has become a favorable alternative to batteries for wearable embedded systems since it is more environmental and user friendly. However, harvested energy is intrinsically unstable, which could frequently interrupt a processor's execution. To tackle this problem, non-volatile processors have been proposed to checkpoint the whole volatile processor state into attached non-volatile memories periodically. When power resumes, the processor can copy the checkpointed state back to volatile memories and continue execution. However, without careful consideration, the process of checkpointing and resuming could cause inconsistency among different memory addresses and lead to irreversible errors. In this paper, we present a consistency aware checkpointing scheme that ensures correctness for all checkpoints. The proposed technique efficiently identifies all possible inconsistency positions in programs and inserts auxiliary code to ensure correctness. Evaluation results show that the proposed checkpointing technique can successfully eliminate inconsistency errors and greatly reduce the checkpointing overhead.

1. INTRODUCTION

From smart phones to smart watches and glasses, smart electronic devices have become an integral part of our daily life. The vision for wearable devices is to interweave technology into our everyday life and improve the quality of life. While the vision is promising and exciting, there are several challenges to achieve this goal. One of the imminent challenges is how to power all these small devices. First, it is difficult to shrink the size of a battery while maintaining the required power supply. Second, closely wearing many batteries will pose safety and health concerns for users. What is more, charging all these batteries every one or two days will give users bad experiences. Therefore, researchers are actively pursuing power alternatives and trying to replace batteries completely. Out of all possible solutions, energy harvesting is one of the most promising techniques to meet both the size and power requirements of wearable devices.

Energy harvesting devices generate electric energy from their surroundings, e.g., solar, wind and kinetic, using direct energy conversion techniques. The obtained energy can be used to recharge capacitors or to directly power the electronics [11]. However, there is an intrinsic challenge with harvested energy. They are all *unstable* [4]. With an unstable power supply, the processor execution will be interrupted frequently. Frequent turning-off and booting-up will place extra burden on limited power budget. What is worse, in some cases, large tasks can never get finished since the intermediate results cannot be saved. To address this problem, Non-volatile Processor (NVP) was proposed to enable instant on/off for these devices [13, 18, 14]. In NVP, the processor's state in SRAM can be backed up into nonvolatile cells upon power failures and the state will be copied back from the closest checkpoint next time power comes back. In this way, the program can resume its execution from last checkpointed position.

For any computer system, correctness is always the most important requirement. However, frequent checkpointing and resuming can cause computation errors. Ransford and Lucia [6] found that frequent power failures will cause inconsistency which makes the nonvolatile memory a broken time machine. Consider the case that right after a checkpoint is taken, an instruction loads a number N from non-volatile memory and increments N by one. Then the value $N+1$ is stored back to the non-volatile memory. After that, a power failure occurs. Then next time the processor rolls back to this checkpoint. It will immediately read the number and increment it by 1 again. However, it will read a value from the future, $N+1$, instead of the original N. Therefore, $N+2$ will be stored to the memory and all subsequent computation will be incorrect. The corresponding load and store instructions are called an error pair. This error exists in battery-less energy harvesting systems even if checkpointing is performed only when a power drop is detected. As [7] pointed out, the checkpointing could fail when saving the state after detecting a power drop. Therefore, the execution has to roll back to the nearest successful checkpoint and such

*This work was supported in part by the NSFC under grant 61271269 and High-Tech Research and Development (863) Program under contract 2013AA01320 and the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions under contract YETP0102.
†This work was partially supported by NSF CNS-1464429.

DAC '15 June 07 - 11 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2744769.2744842.

error occurs.

From the above example, we can see a value illegally "time travel" from the future to the checkpoint and causes errors. In order for NVP to work properly, this broken "time machine" has to be fixed in a systematic way. In this work, we propose a systematic consistency-aware checkpointing mechanism for energy harvesting powered NVP with low checkpointing overhead. Specifically, this paper makes the following contributions:

- A consistency-related error locating mechanism is proposed to find all the potential error pairs and all the program paths between each error pair;
- A consistency-aware checkpointing algorithm is designed to eliminate all error pairs and it generates the minimal number of checkpoints;
- Detailed experimental evaluation is implemented to demonstrate the efficiency of the proposed consistency-aware checkpointing mechanism.

The remainder of this paper is organized as follows. Section 3 describes the hardware and software model and defines the problem in this paper. Section 4 provides a motivational example to illustrate the motivation of this paper. Section 5 presents the PEL and CACI algorithms. Detailed experimental evaluation is provided in Section 6. Finally, Section 7 concludes this paper.

2. RELATED WORKS

Energy harvesting sources, including solar, wind, finger motion, and footfalls, are studied and modeled [3, 10]. They have different characteristics on predictability and magnitude. Due to considerations on area, weight, and maintenance, large capacitors are not favored in energy harvesting powered wearable devices, which makes the energy source even more unstable [12].

In order to overcome the instability of harvested energy, researchers develop NVP for energy-harvesting devices [13, 18]. In NVP, non-volatile memory is attached to the processor. The volatile execution state is checkpointed into the non-volatile memory upon power failure. Ransford et al. [7] present a software system for transiently powered RFID-scale devices in which execution state is checkpointed to flash upon power loss.

FRAM is preferred as the attached non-volatile memory (NVM) due to its comparable access efficiency to SRAM and a satisfactory endurance of 10^{14} write cycles [18]. Many works have been done to reduce the volatile state to be backed up. Wang et al. [8] propose data compression based hardware design. Xie et al. [15] transmit writes on registers to the main memory. Xie et al. [16] and Zhao et al. [17] employ instruction scheduling and software analysis methods to find best checkpointing positions.

Even though the above-mentioned works can improve the checkpointing efficiency, they all suffer from the potential risk of errors in energy harvesting systems, which severely degrades the functionality and service quality of the energy harvesting powered devices. In this work, we develop schemes to identify all possible inconsistency errors in programs and eliminate errors by auxiliary code insertion to ensure correctness.

3. HARDWARE AND SOFTWARE MODEL

In this section, we will present the hardware and software models and formally define the problem in this paper.

In this paper, we are targeting embedded systems that

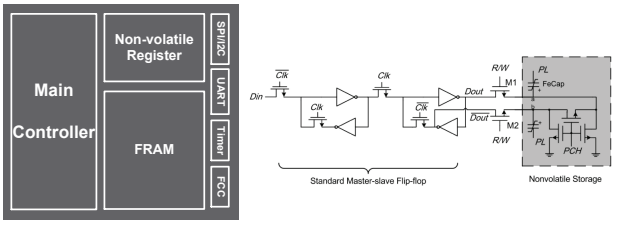


Figure 1: System architecture.

are powered directly by energy harvesting [12]. Since there is no power storage, to defend against frequent power interruptions, the states in volatile registers need to be checkpointed periodically. The processor architecture is shown in Figure 1(a), which consists of the processing unit, non-volatile register file, and non-volatile on-chip memory.

Non-volatile register file consists of FRAM based non-volatile flip-flops (NVFFs) [13]. The NVFF structure is shown in Figure 1(b). An FRAM-based non-volatile storage is attached to a standard volatile flip-flop. The content of a volatile flip-flop can be saved to the NV storage. Therefore, when the content of the volatile register is lost due to power failure, the content in the NV storage can be copied back into the volatile register. Then, the processor can resume working from the saved checkpoint. The on-chip memory is completely deployed with FRAM, which has fast access speed and long lifetime. Since the on-chip memory is non-volatile, the content does not need to be checkpointed. TI's MSP430FRxx series microcontrollers [2] is an existing microprocessors with similar architecture.

In this work, we are aiming to systematically optimize the software such that it works correctly on NVP. The proposed algorithms take Control Flow Graph (CFG) as input. A CFG is a directly connected graph represented as CFG=$<V$, $E>$. The vertex set V contains the basic blocks of this program. The edge set $E=\{e|bb_i \rightarrow bb_j, bb_i \in V, bb_j \in V\}$ denotes the set of directed edges and each edge represents the execution flow direction from one basic block to another. A basic block is a sequence of consecutive instructions in which control flow enters at the beginning and leaves at the end without halting or branching except at the end.

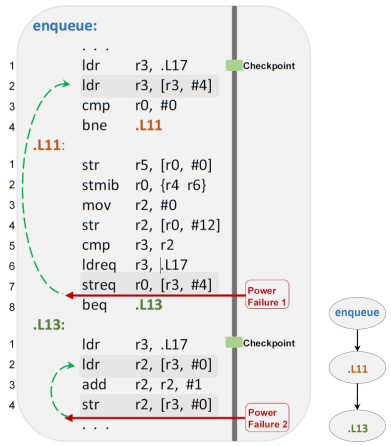
The problem this work targets is formulated as follows. Given the basic block based CFG of a program, determine checkpointing positions so that potential inconsistency errors in the program can be fully eliminated. At the same time, the number of checkpointing positions is minimized.

4. MOTIVATIONAL EXAMPLE

In this section, we present an example to show the motivation of this work. For illustration purpose, part of the code from dijkstra's algorithm is used. The assembly code generated by gcc-arm cross-compiler is shown in Figure 2(a).

In this figure, there are three basic blocks of code whose names are *enqueue*, *.L11* and *.L13*, respectively. The control flow graph (CFG), which represents the relation of these three blocks of code, is shown in Figure 2(b). Because of the instability, two checkpoints are set at the end of the first instruction in *enqueue* and *.L13*. However, wrong checkpoint positions may result in vital errors both insides a basic block and across multiple basic blocks. Figure 2(a) shows how NV inconsistency causes different execution errors.

First, let us use block *.L13* to show that an error may



(a) (b)

Figure 2: Example code: Dijkstra's Algorithm

occur inside a basic block if there is a power failure. Here, a checkpoint is taken after the first instruction of *.L13*. Then instruction 2 of *.L13* loads the value in Mem[r3+0] to register r2. Instruction 3 increases the value in r2 by 1. After that instruction 4 stores the new value in r2 to the same address Mem[r3+0]. Assume that a power failure occurs at the end of instruction 4. Then, when power returns, the computation will roll back to the checkpoint at the end of instruction 1.

Although the execution can continue after power recovers, the computation is no longer correct. Before the power failure, the variable at Mem[r3+0] is already increased by 1. However, after the power recovers, the same operation is executed again, which means this variable is increased by 2 totally. Therefore, all subsequent computations that depend on this variable will no longer be correct either.

This inconsistency may also occur across several basic blocks. In Figure 2(a), this error occurs across *enqueue* and *.L11* as well. In this code, *.L11* is executed sequentially after *enqueue*. Instruction 2 of *enqueue* loads value stored in Mem[r3+4] into register r3, and instruction 7 of *.L11* stores another value in register r0 into the same address Mem[r3+4]. This pair of load and store operations change the value stored at the same address in NVM. When power failure occurs at the end of instruction 7 of *.L11*, the execution has to restart from instruction 2 by restoring the states backed up at the checkpoint before instruction 2. At this time, load instruction still loads a value from Mem[r3+4]. However, this value is already a different value. The resumed execution is different from the one without power failure. Therefore, there will be an error in the execution.

From our analysis, we found that there are 6 potential errors that reside inside a basic block and there is 1 potential error across two blocks in dijkstra's algorithm alone. This type of error happens because of checkpointing only part of the execution states, namely only the states on registers. In this example, carefully rearranging the positions of the checkpoints can eliminate this inconsistency. For example, if the first checkpoint is taken at the end of instruction 2 of *enqueue*, and the second checkpoint is taken at the end of instruction 2 of *.L13*, inconsistency errors will be eliminated.

From this example, we can see that in order to ensure correct execution, we have to find these potential locations first. Besides, we need to smartly decide the checkpoint positions to eliminate these potential inconsistencies. Third, minimal number of checkpoints is desirable since checkpoint consumes power. In this paper, we will devise three algorithms that will achieve these three goals.

5. ALGORITHM

In this section, we propose an efficient scheme to eliminate consistency-related errors by setting up consistency-aware checkpoints instead of traditional periodical checkpointing.

5.1 Challenges and Analysis

The errors result from the fact that after power failures, the program execution goes back before a "load" instruction and loads an updated memory data. Consequently, inserting a checkpoint between each error pair can fully eliminate these errors. This is because after a power failure, the program execution will go back to the new checkpoint which is after the load instruction, and the updated memory data will not be loaded and thus there is no inconsistency error. Therefore, we want to determine all the checkpointing positions, so that: 1) all the consistency-related errors can be eliminated; 2) the number of checkpoints is minimized while guaranteeing the correctness. The first goal guarantees the correctness of the program while the second goal minimizes the overhead.

One challenge in this problem lies in how to locate the errors. Since a program consists of many branches and loops, inconsistency errors both inside the same basic block and across different basic blocks need to be located. In this paper, we will first locate the potential load-store error pairs and search for the acyclic paths between the load instruction and the store instruction. By doing this, all kinds of errors can be located. The second challenge lies in where to insert the checkpoint. In this paper, we set the distance between adjacent checkpoints to be up-bounded by a constant L_{max}. This distance is to ensure that the processor can resume execution as fast as required. The distance is defined as the energy consumption of instruction execution, and we name it as the checkpointing energy distance in this paper. Then it will be better to insert a checkpoint at every L_{max} energy distance. Meanwhile, the inserted checkpoints should eliminate the errors.

This problem is NP-Complete, which can be proved by reducing from the set cover problem [5]. Therefore, in this section, we will propose an efficient two-step heuristic scheme to solve this problem. The first step is error locating, where all the load-store pairs that could potentially lead to errors are identified. The second step is consistency-aware checkpointing, where checkpoints are determined.

5.2 Potential Error Locating

Based on previous analysis, a pair of "load" and "store" instructions to the same address in the memory is a potential error region. A pair can reside both in a single basic block or across multiple basic blocks. If this pair of instructions exist inside a block, it is relatively easy to be found. However, if this pair of instructions exist across multiple blocks, it is a nontrivial task since we need to traverse all possible execution paths of the program. In this subsection, we will present the Potential Error Locating (PEL) algorithm, which can identify all potential error-prone load-store pairs as well as corresponding paths between each pair.

Algorithm 5.1 shows the procedure of the PEL algorithm.

Algorithm 5.1 Potential Error Locating (PEL)

Input: Basic block based *CFG*, the instruction sequence of each basic block;
Output: All pairs of memory modification instructions and all acyclic paths between each pair;
1: Derive basic block list S based on the depth-first search on *CFG*;
2: $n \leftarrow 0$;
3: **for** each basic block BB_i in S **do**
4: **for** each instruction k from 1 to *Len(*BB_i*)* **do**
5: **if** k loads a value from the memory address Mem **then**
6: **for** each basic block BB_j in S **do**
7: **for** each instruction l from 1 to *Len(*BB_j*)* **do**
8: **if** l stores a different value to the same memory position Mem **then**
9: Record all the acyclic paths from BB_i and BB_j to the set $PATH_n$;
10: Record instruction index k and l to the set $INDEX_n$;
11: $n \leftarrow n + 1$;
12: **end if**
13: **end for**
14: **end for**
15: **end if**
16: **end for**
17: **end for**

The input is a CFG and the instructions in each basic block. To identify load-store pairs, all basic blocks in CFG are examined in the depth-first order to locate "load" instructions (Line 5). All pairs of "load" and"store" instructions targeting to store a different value at the same memory address are searched, and all possible acyclic paths between each pair are stored (Line 6-14).

An example is shown in Figure 3(a) to illustrate the algorithms. There are five basic blocks in this example, containing 6, 8, 8, 5, and 2 instructions, respectively. Based on Algorithms 5.1, 4 "load" instructions will be found: *ldr a* at position (BB_0, ins1), *ldr c* at position (BB_1, ins4), *ldr b* at position (BB_1, ins2) and *ldr a* at position (BB_4, ins0). Figure 4 shows paths between each pair.

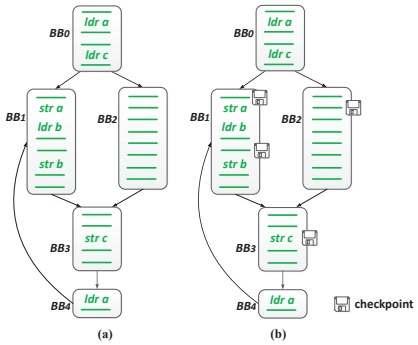


Figure 3: An example.

5.3 Consistency-aware Checkpointing

After deriving all the potential error locations, we are ready to determine an error-free checkpointing scheme with the minimal number of checkpoints and the distance between adjacent checkpoints within L_{max}.

The main challenges lie in the following aspects. First, to reduce all inconsistency errors, there needs to be at least one checkpoint between each load-store pair. Simply inserting a checkpoint between each pair is not feasible due to the high overhead. Therefore, effective approaches are necessary to reduce checkpoints. Second, since the runtime path is not available during the offline analysis, all possible paths need to be taken into consideration to reduce the checkpointing overhead. Third, the checkpoint locations in various basic blocks may affect each other. Therefore, a global optimal solution is needed.

Algorithm 5.2 Consistency-aware checkpoints inserting algorithm (CACI)

Input: Basic block based CFG,$INDEX$, $PATH$;
Output: Consistency-aware checkpoints;
1: $E1_i \leftarrow 0, i = 1, 2, ..., N$;//N is the number of all basic blocks.
2: $E2_i \leftarrow 0, i = 1, 2, ..., N$;
3: **for** each BB_i, $BB_i \in$ breadth-first traversal array S **do**
4: Search $INDEX$ and $PATH$ to find all the load-store pairs (l_i, s_i), single load indices ll_i and single store indices ss_i in BB_i, and store them separately into sets LS, L and S;
5: $Energy \leftarrow 0$; $start \leftarrow 1$; $K_i \leftarrow Len(BB_i)$;
6: Find all the predecessors of BB_i, $BB_{p1}, BB_{p2}, ..., BB_{pn}$;
7: $Energy \leftarrow Energy + max(E2_{p1}, ..., E2_{pn})$;
8: **for** $k = start$ to K_i **do**
9: $Energy \leftarrow Energy + en_k$;
10: **if** $(Energy > L_{max})$ **or** $(k == K_i)$ **then**
11: $ckp \leftarrow min(k, s_j, ss_j) - 1$, $s_j \in LS, ss_j \in S$;
12: Insert a checkpoint after instruction ckp;
13: **ReduceErrors**(ckp, LS, L, S);
14: **end if**
15: $start \leftarrow ckp + 1$;
16: $Energy \leftarrow 0$;
17: **end for**
18: Find all the descendants of BB_i, $BB_{d1}, ..., BB_{dn}$;
19: **if** $sum(en_{ckp_{last}}, ..., en_{K_i}) + max(E1_{d1}, ..., E1_{dn}) > L_{max}$ **then**
20: Insert a checkpoint after instruction K_i;
21: **end if**
22: **if** BB_i has no checkpoint **then**
23: $E1_i \leftarrow sum(en_1, ..., en_{K_i}) + max(E1_{d1}, ..., E1_{dn})$;
24: $E2_i \leftarrow sum(en_1, ..., en_{K_i}) + max(E2_{p1}, ..., E2_{pn})$;
25: **else**
26: $E1_i \leftarrow sum(en_1, ..., en_{ckp_{first}})$;
27: $E2_i \leftarrow sum(en_{ckp_{last}}, ..., en_{K_i})$;
28: **end if**
29: **end for**

Algorithm 5.3 *ReduceErrors*(ckp, LS, L, S)

1: **for** each load-store pair (l_i, s_i), $(l_i, s_i) \in LS$ **do**
2: **if** $ckp \geq l_i$ **and** $ckp < s_i$ **then**
3: Remove (l_i, s_i) from LS and renew LS; Remove (l_i, s_i) from $INDEX$;
4: **end if**
5: **end for**
6: **for** each single load ll_i, $ll_i \in L$ **do**
7: **if** $ckp \geq ll_i$ **then**
8: Remove ll_i from L and renew L; Remove ll_i from $INDEX$ and its corresponding paths from $PATH$;
9: **end if**
10: **end for**
11: **for** each single store ss_i, $ss_i \in S$ **do**
12: **if** $ckp < ss_i$ **then**
13: Remove ss_i from S and renew S; Remove ss_i from $INDEX$ and its corresponding paths from $PATH$;
14: **end if**
15: **end for**

In this subsection, we propose a polynomial time heuristics, the consistency-aware checkpoints inserting (CACI) al-

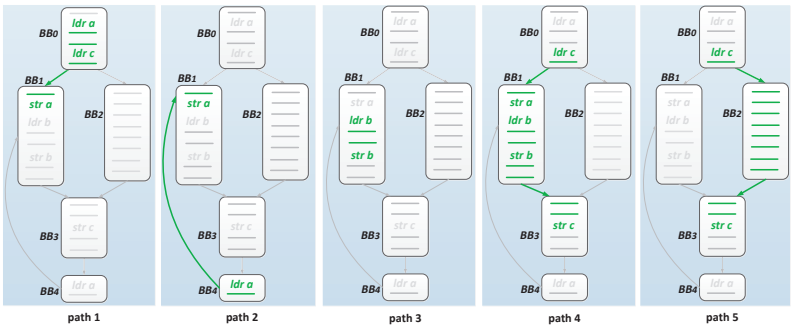


Figure 4: Paths between all load-store pairs.

gorithm, which is shown in Algorithm 5.2. The input is a CFG and all the paths between load-store pairs. Checkpoint positions inside a basic block are highly dependent on those in its predecessors. At the same time, they also affect those in its successors. To record this kind of relation, we develop two arrays: $E1$ to represent the distance between the first instruction and the first checkpoint in the current basic block, if any; $E2$ to represent the distance between the last checkpoint in this basic block and the last instruction of this basic block (Line 1-2).

Each basic block is processed in the breadth-first order in CFG. For each basic block, first the *Energy*, which represents the distance between this basic block and previous checkpoint is initialized as the maximum value of its predecessors (Line 7). This initialization guarantees the L_{max} principle whichever path is taken at runtime. Inside the process of a basic block, *Energy* is updated by going through each instruction.

There are two cases to trigger checkpoint insertion (Line 10). First, once *Energy* reaches L_{max}, one checkpoint will be inserted. Second, one checkpoint will be inserted to eliminate errors. Two sets, $(l_i,\ s_i)$ and ss_i, are constructed to guide the error detection. $(l_i,\ s_i)$ includes all the load-store pairs inside BB_i and ss_i records all the single stores inside BB_i. If both sets are not empty, one checkpoint will be inserted before the first store instruction (Line 11-12).

Take BB_1 in Figure 3(a) as an example. We assume *ldr/str* instructions consume 2-unit amount of energy and others consume 1 unit. Before processing BB_1, BB_0 is processed with no checkpoints inside it and $E2_0$=8. Thus for BB_1, $(l_1,\ s_1)$={*(ldr b, str b)*}, ss_1={*str a*}, and *Energy* is initialized to be max{$E2_0$, $E2_4$}=8. By going through all the instructions in BB_1 before instruction *str b*, *Energy* is increased to L_{max}, which is 15. Thus, a checkpoint is inserted here. Note that the inserted checkpoint may be able to eliminate errors for one or more paths between load-store pair(s). In this example, this checkpoint can eliminate the error for path 3 shown in Figure 4.

Thus, after each checkpoint insertion, Algorithm 5.3 will be called to update the potential error location and all the paths pass the error location (Line 13). Then the process of this basic block starts from the instruction after the checkpoint (Line 15-16).

In the example, the processing will start from *str b* in BB_1, with current $(l_1,\ s_1)$ being empty and ss_1={*str a*}. Next checkpoint will be inserted before ss_1, followed by the update of $E1_1$=1 and $E2_1$=4. As a result, path 1 and 2 in Figure 4 are removed from the error set. Similarly, after processing BB_2, one checkpoint is inserted after the second instruction. Therefore, $E1_2$=2 and $E2_2$=6. After processing BB_3, one checkpoint is inserted after *str c*. Thus, $E1_3$=2 and $E2_3$=2. There will be no checkpoint in BB_4 and $E1_4$=5 and $E2_4$=5. At the end of processing BB_4, the loop to BB_1 is examined to guarantee the distance between two checkpoints is less than L_{max} (Line 18-21). The final checkpoint set is shown in 3(b). There are 4 total checkpoints.

After CACI, all loops in the DFG will be checked whether it contains a checkpoint. If not, a checkpoint is inserted at the end of the loop. This insertion only happens when the energy consumption of a loop iteration is less than L_{max}.

In a nutshell, the proposed schemes achieve an error-free checkpoint set by assigning checkpoints either before *store* or at the distance of L_{max} since the previous checkpoint.

6. EXPERIMENTAL RESULTS

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

6.1 Experiment Setup

Table 1: Target System Specification.

Component	Description
CPU	Core:1
NV registers	SRAM (attached with FRAM [9]) read energy/write energy: 0.1nJ
NV memory	FRAM [18], read energy: 0.2nJ, write energy:1.0nJ

The experiments are conducted with a custom ARM processor based simulator. The system specifications are shown in Table 1. The volatile register is based on SRAM, with read/write energy of 0.1 nJ. The non-volatile register and on-chip non-volatile memory are based on FRAM with read energy of 0.2 nJ and write energy of 1.0 nJ.

Assembly code generated by GCC-ARM cross-compiler is used as input. The proposed techniques serve as another pass of front-end source code analysis. Therefore, it can be easily incorporated into any existing compilers. We conducted the experiments on a set of 9 benchmarks including aes, dijkstra, and fourierf, etc. These benchmarks are from the Mibench [1] benchmark.

6.2 Experimental Results and Analysis

In this section, we compare the proposed consistency aware checkpointing scheme with the general checkpointing method without considering the errors. Table 2 shows the comparison results. It lists the number of potential errors when the checkpointing energy distance is set to be 5nJ, 10 nJ, 20 nJ and 50 nJ, separately. For example, the third row of the

table corresponding to benchmark dijkstra shows that there are totally 55 potential error pairs. Among the 55 error pairs, there are 8 error pairs such that the energy distance of each pair is less than 5 nJ. The proposed CACI algorithm can find all the potential error pairs and eliminate them. From the table, we can see that the number of errors varies in different benchmarks. Benchmark fourierf has as many as 65 potential errors while qsort only has 1 error. However, even a single error can cause the fatal results of the program. Therefore, the proposed error aware algorithm is of significant importance.

Table 2: Number of potential error pairs given different maximal checkpointing energy distances.

Benchmarks	Checkpointing distance (nJ)					
	5	10	20	50	Unlimited	CACI
aes	1	1	1	1	13	0
bmhisrch	1	1	1	1	1	0
dijkstra	8	11	13	27	55	0
fourierf	6	13	19	49	65	0
patricia	7	9	13	23	23	0
pbmsrch	0	0	0	0	1	0
qsort	0	0	0	1	1	0
sha	13	14	14	14	16	0
bf_cbc	5	5	7	39	63	0

Table 3 shows the comparison of checkpoints to NV registers between the proposed error aware algorithm and the error unaware algorithm, when the checkpointing energy distance is 5 nJ and 10 nJ. From the table, we can see that the proposed CACI algorithm can guarantee the correct execution of the program at the expense of only 6.3% more checkpoints on average when the checkpointing energy distance is set to be 5 nJ and 25.2% on average when the checkpointing energy distance is set to be 10 nJ. The general error unaware algorithm generates fewer checkpoints because it does not take care of the potential errors.

Table 3: Comparison of number of checkpoints given different checkpointing energy distances.

Benchmarks	Error unaware		Error aware		Overhead	
	5	10	5	10	5	10
aes	443	219	443	219	0%	0%
bmhisrch	13	3	13	4	0%	33.3%
dijkstra	26	3	32	5	23.1%	66.7 %
fourierf	25	6	27	9	8%	50%
patricia	27	3	30	6	11.1%	50%
pbmsrch	14	4	14	4	0%	0%
qsort	15	2	15	2	0%	0%
sha	41	14	46	17	12.2%	21.4%
bf_cbc	41	19	42	20	2.4%	5.3%
Average					**6.3** %	**25.2** %

6.3 Running Time Analysis

Table 4: Time cost of the inconsistency aware checkpointing algorithm

Benchmarks	Energy distance is 5 nJ		Energy distance is 10 nJ	
	Locating	Checkpointing	Locating	Checkpointing
aes	5716ms	231ms	6130ms	411ms
bmhisrch	80ms	20ms	81ms	20ms
dijkstra	100ms	60ms	110ms	60ms
fourierf	120ms	61ms	120ms	66ms
patricia	80ms	40ms	90ms	40ms
pbmsrch	70ms	20ms	70ms	20ms
qsort	80ms	22ms	80ms	20ms
sha	90ms	40ms	100ms	40ms
bf_cbc	140ms	36ms	158ms	40ms
Average	719ms	59ms	771ms	83ms

Table 4 lists the running time cost of the proposed inconsistency aware checkpointing algorithm for each benchmark. The detailed time cost of both the error locating and checkpoint inserting given in the table are collected on a single core processor running at 3.00GHz. Error locating takes more time than checkpoint inserting does, since error locating searches for all the acyclic paths between each error pair. The table shows that both error locating and checkpoint inserting are efficient.

7. CONCLUSIONS

In this work, we propose an inconsistency-aware checkpointing scheme for energy harvesting powered non-volatile processors. To guarantee the correct execution and reduce checkpoint overhead, we propose a systematic approach to determine the most appropriate checkpointing positions for each basic block by considering all execution paths that have an influence on the correctness and the overhead. The evaluation results show that all the inconsistency errors can be located, and the overhead of the extra checkpoints for eliminating the errors is acceptable given the importance of execution correctness.

8. REFERENCES

[1] M. R. Guthaus, J. S. Ringenberg, et al. Mibench: A free, commercially representative embedded benchmark suite. In *2001 IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[2] T. Instruments. Msp430frxx microcontrollers. `http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/ultra-low_power/msp430frxx_fram/overview.page`.

[3] X. Jiang, J. Polastre, et al. Perpetual environmentally powered sensor networks. In *IPSN*, pages 463–468, 2005.

[4] A. Kansal, J. Hsu, et al. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4):32, 2007.

[5] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. 1972.

[6] B. Ransford and B. Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC, 2014.

[7] B. Ransford, J. Sorber, et al. Mementos: system support for long-running computation on rfid-scale devices. *ACM SIGPLAN Notices*, 47(4):159–170, 2012.

[8] X. Sheng, Y. Wang, et al. Spac: A segment-based parallel compression for backup acceleration in nonvolatile processors. In *DATE*, pages 865–868, 2013.

[9] H. Shiga, D. Takashima, et al. A 1.6 gb/s ddr2 128 mb chain feram with scalable octal bitline and sensing schemes. *IEEE Journal of Solid-State Circuits*, 45(1):142–152, 2010.

[10] T. Starner. Human-powered wearable computing. *IBM systems Journal*, 35(3.4):618–629, 1996.

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

[12] C. Wang, N. Chang, et al. Storage-less and converter-less maximum power point tracking of photovoltaic cells for a nonvolatile microprocessor. In *ASP-DAC*, pages 379–384, 2014.

[13] Y. Wang, Y. Liu, et al. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *ESSCIRC*, pages 149–152, 2012.

[14] Y. Wang, Y. Liu, et al. Pacc: A parallel compare and compress codec for area reduction in nonvolatile processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(7):1491–1505, 2014.

[15] M. Xie, C. Pan, et al. Non-volatile registers aware instruction selection for embedded systems. In *RTCSA*, pages 1–9, 2014.

[16] M. Xie, C. Pan, et al. Checkpoint-aware instruction scheduling for nonvolatile processor with multiple functional units. In *ASP-DAC*, pages 316–321, 2015.

[17] M. Zhao, Q. Li, et al. Software assisted non-volatile register reduction for energy harvesting based cyber-physical system. In *DATE*, pages 567–572, 2015.

[18] M. Zwerg, A. Baumann, et al. An 82ua/mhz microcontroller with embedded FeRAM for energy-harvesting applications. In *ISSCC*, pages 334–336, 2011.