Software Assisted Non-volatile Register Reduction for Energy Harvesting Based Cyber-Physical System

Mengying Zhao*, Qingan Li†, Mimi Xie‡, Yongpan Liu§, Jingtong Hu‡, Chun Jason Xue*
*Department of Computer Science, City University of Hong Kong, Hong Kong
†State Key Laboratory of Software Engineering, Wuhan University, China
‡School of Electrical and Computer Engineering, Oklahoma State University, USA
§Department of Electronic Engineering, Tsinghua University, China

***Abstract*—Wearable devices are important components as information collector in many cyber-physical systems. Energy harvesting instead of battery is a better power source for these wearable devices due to many advantages. However, harvested energy is naturally unstable and program execution will be interrupted frequently. Non-volatile processors demonstrate promising advantages to back up volatile state before the system energy is depleted. However, it also introduces non-negligible energy and area overhead. Since the chip size is a vital factor for wearable devices, in this work, we target non-volatile register reduction for application-specific systems. We propose to analyze the application program and determine efficient backup positions, by which the necessary non-volatile register file size can be significantly reduced. The evaluation results deliver an average of 62.9% reduction on non-volatile register file size for stack backup, with negligible storage overheads.**

I. Introduction

Cyber-physical systems (CPS) are physical and engineered systems whose operations are monitored, coordinated, controlled and integrated by a computing and communication core. As one category of CPS, medical cyber-physical systems (MCPS), which involve sensors, embedded software and networking capabilities, necessitate the development of wearable technology [1]. Wearable technology enables the devices to keep close contact with patients to monitor the body status such as breathing band and blood pressure. In order to make a computer system "wearable", there are a couple of important challenges. First, these systems must be small enough to avoid any discomfort to users. Second, these small computers must be powered properly. For the wearable devices, battery is no longer a favorable power source due to 1) large size and weight; 2) safety and health concerns; 3) frequent recharges. Therefore, researchers are actively pursuing power alternatives. 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 its surroundings using direct energy conversion techniques. Examples of power sources include kinetic, electromagnetic radiation (including light and RF), and thermal energy. The obtained energy can be used to recharge a capacitor or, in some cases, to directly power the electronics [2]. However, there is an intrinsic challenge with harvested energy. They are all *unstable* [3]. With an unstable power supply, the processor execution will be interrupted frequently. Frequent turning-off and rebooting will place extra burden on limited power budget. What is worse, in some cases, large tasks can never finish the execution since the intermediate results cannot be saved. To address this problem, Non-volatile Processor (NVP) was proposed to enable instant on/off execution for these devices [4, 5]. In the NVP shown in [4], a non-volatile FRAM (Ferroelectric Random Access Memory) is attached to the processor's registers. Every time there is a power outage, the processor's state, including registers and stack space, will be saved to the NV FRAM. Then the next time power comes back, the processor's state is copied back and the program execution is resumed.

One challenge faced by the energy harvesting based embedded system with NVP is the significant area overhead. A 40% area overhead is observed in a low-pass digital filter by replacing traditional registers with Magnetic NV flip-flops [6]. Thus the area reduction for NV processors is necessary to satisfy the size requirements for energy harvesting based wearable devices. In order to reduce NVP area, a hardware compressor/decoder was proposed to compress the whole processor state [7, 8]. It reduces the NV register file size at the cost of extra logic circuits to conduct compressing and decompressing. In this work, we target the NV register reduction from the software perspective. Among the program state to back up, stack space is often much larger than the registers. Reducing stack space is the focus of this paper. Two key observations are as follows. First, not every data is necessary to resume program execution; Second, the backup does not have to take place right away. Instead of compressing all the data, in this paper, we analyze the program execution path and identify the reachable positions where a smaller program state is needed to be saved. In this way, a much smaller NV register file is needed to back up the program state when the energy source is depleting.

To the best of our knowledge, this is the first work to reduce NV register file size at the software level. The proposed software technique can work together with hardware compression logics to provide even better efficiency. Specifically, this paper makes the following contributions.

- Propose to reduce the NV register file size by stack utilization analysis and backup position optimizations for application-specific energy harvesting systems;

978-3-9815370-4-8/DATE15/©2015 EDAA

- Propose a set of schemes to determine the optimal backup position for each basic block, and then decide the required minimum NV register file size;
- Evaluate the efficacy of the proposed schemes when compared with existing instant backup.

The remainder of this paper is organized as follows. Section II summaries related work. Section III introduces the state-of-the-art NVP and presents motivation examples. Section IV presents the proposed scheme. Section V presents evaluations and Section VI concludes this paper.

II. Related work

In this section, we will describe related work on energy harvesting systems and non-volatile processors.

Energy harvesting sources can be used to deploy long lifetime battery-less devices. Solar, wind, finger motion, footfalls, breathing, and blood pressure are all promising sources [9, 10]. They have different characteristics on predictability and magnitude, while are all unstable. In order to overcome the instability of harvested energy, FRAM based NVP is proposed to be deployed in energy-harvesting devices to achieve instant on/off execution [4, 11, 12]. FRAM is adopted due to its comparable access efficiency to SRAM and good endurance. With FRAM attached, the original volatile logics can be backed up into non-volatile memory before the energy is depleted. Then the program state can be copied back when the system is recharged, and the program execution can be resumed efficiently.

There is previous work on lifetime enhancement and energy reduction on NV registers [13, 14]. Since many energy-harvesting applications require a small size, in this work, we focus on NV size reduction. Previous related work relies on data compression to reduce the content size for backup [5, 7, 8]. These strategies are based on hardware and the efficacy is highly dependent on the data compression ratio. In this work, we tackle this problem from the software perspective. Based on the observation that not every data is necessary to resume program execution, we analyze the program execution trace and find a reachable backup position where the smallest program state is needed to be saved. In this way, the required NV size can be effectively reduced.

III. Preliminaries and motivation

In this section, the non-volatile processor is first introduced; then a motivation example is presented to illustrate how backup procedures affect the NVP size; finally the target problem is presented and defined.

A. Non-volatile processor

Fig. 1 shows the structure of the state-of-the-art NV processor, including on-chip non-volatile register files to back up volatile logics of the system when energy is depleted. Conventionally, all the volatile status, e.g. general purpose register, user variables and stack, are copied to the NV register file so that the program can resume after system is recharged. Thus the necessary NV register file size is conservatively designed to be the same as, or even larger than the volatile logic. Stack size dominates the internal volatile logics (around 63% [4]) and thus this paper focuses on the stack size reduction.

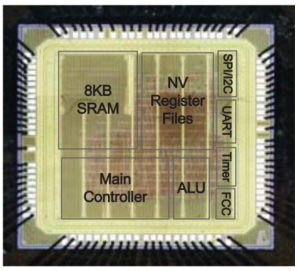

Fig. 1. NV processor structure proposed in [4].

B. Motivation example

Fig. 2(a) presents a sample code, where the *main* function invokes function *g*; *g* invokes *h*; and *h* invokes *i*. We analyze the stack usage as shown in Fig. 2(b). At the beginning of program execution, the *main* function is assigned a frame (also called the active record) to store the context information for this function. Local data, including local variables and temporary variable, are stored in this frame. The stack consists of allocated frames. Here we assume that the frame size for each function is consistent. When the program enters function *g*, *g* is allocated with another frame. It is similar for the following functions. We can see that at time t_1, the stack space has four frames while only has one frame at time t_2 as function *i*, *h* and *g* have finished and returned.

Assume that the system receives an energy warning signal at time t_1. We know that limited energy is left and the system would be forced to shut down soon. The conventional backup strategy adopts *instant backup*, where all the processor states are backed up to NV register files at t_1. In this case, this system needs NV register file with the size of four frames for the stack section. Instead of consuming a large portion of energy to back up, we propose to spend some energy to continue the program execution until t_2. At t_2, only the frame of *main* function is in the stack space and thus there is only one frame to back up at this moment. If the available energy can always support the program to arrive t_2 and also enough for backing up one frame, we can design the NVP with NV register file size of one frame instead of traditional four frames

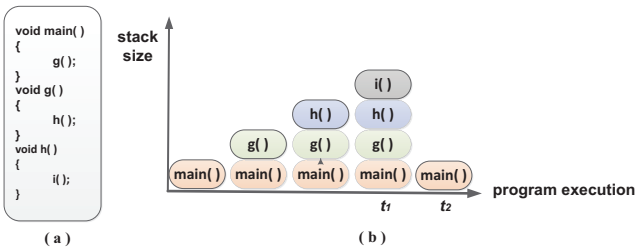

Fig. 2. Motivation example.

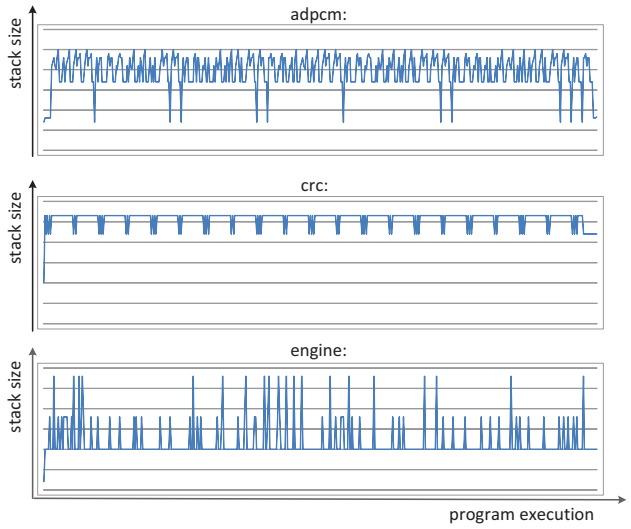

Fig. 3. Dynamic stack sizes along program execution for three benchmarks from *powerstone* [15].

by changing the backup position, with a stack size reduction of 3×.

To understand the stack behavior during program execution, we analyze several benchmarks from powerstone suite [15]. Fig. 3 shows the dynamic stack sizes along the program execution for three benchmarks. The stack size distributions confirm the feasibility of the proposed stack reduction scheme with frequent stack size fluctuations.

The proposed scheme essentially suggests a more flexible energy utilization after receiving the energy warning signal. Instead of instantly backing up all the processor status, it is possible to continue executing for a little while and find a better and reliable backup position, where the stack size to back up is smaller. As a result, the system can be designed with a smaller NV register area.

C. Problem Definition

Before presenting the problem formulation, we first introduce the basic block based control flow graph (CFG) for program representation. A basic block is a sequence of consecutive instructions in which control flow enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

A *CFG* is a directed 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. Fig. 4 shows an example of *CFG* with ten basic blocks. Even though instructions are sequentially executed inside basic blocks, there maybe branches and loops between basic blocks and a *CFG* can represent all possible program structures.

The problem in this work can be formulated as follows. Given the input includes:

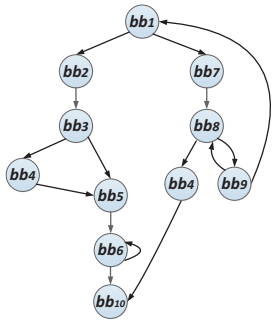

Fig. 4. An example of a basic block based CFG.

- The available energy amount Ene when receiving energy warning, which is a fixed constant;
- The basic block based CFG of a specific application;
- The system prototype as well as the energy consumption model for both instruction execution and backup,

to determine, for each block:

- A constant $B(bb)$, indicating to back up the volatile logics after executing the subsequent $B(bb)$ basic blocks, if the energy warning takes place in basic block bb.

Conventionally, $B(bb)$=0 for each basic block, indicating instant backup after receiving energy warning. In this work, we aim to calculate $B(bb)$ for each basic block so that the size of NV register files can be reduced to save chip area. The main idea is to conduct offline profiling and analysis, then attach $B(bb)$ information to each basic block, so that the backup positions can be efficiently determined at runtime.

The main challenge in the proposed scheme lies in the jumps among basic blocks resulting from branches and loops. Since the execution path cannot be determined during offline analysis, one solution is to enumerate all cases and specify the optimal backup position for each possibility. However, this would involve huge area overhead to store the information, as well as significant runtime performance overhead to match the real situation with the stored records. In this work, we target an offline strategy, whose decision introduces low-storage overhead and efficient runtime utilization. Specifically, for each basic block, we analyze limited feasible paths and conclude a single decision, i.e. $B(bb)$. We will show that the $B(bb)$ fits all possible subsequent execution possibilities for basic block bb, and the NV register size can be reduced by moving the backup position to $B(bb)$ blocks later.

IV. Software Assisted NV Register Reduction

In this section, we introduce the proposed software assisted NV register file reduction scheme. The objective is to reduce the necessary NV register number used for stack backup.

Algorithm 1 shows the offline analysis procedure. For each basic block, we first derive all possible backup positions by analyzing the limited feasible paths (Line 2), then $B(bb_i)$ can be calculated by a strategy named "*MinCF*" (Line 3). The constant $B(bb_i)$ means that if energy warning happens within basic block bb_i, instead of instant backup, the program should

Algorithm 1 Backup stack size reduction.

Input: program *CFG*, available energy *Ene*, all basic blocks *bb*;
Output: NV register number *NV#*

1: **for** each basic block bb_i, $bb_i \in V$ **do**
2: Derive the feasible backup set $F(bb_i)$ for all possible paths; *//Section IV-A*
3: $B(bb_i)$= $MinCF(F(bb_i))$, $bb_i \in V$; *//Section IV-B*
4: **end for**
5: *NV#= Max*($B(bb_i)$); *//Section IV-C*

continue the execution to finish the subsequent $B(bb_i)$ number of basic blocks and then back up the volatile logics at the end of the corresponding basic block. Regardless of the jump choices at runtime, the determined $B(bb_i)$ can always guarantee a successful backup and reduce the stack size to back up, when compared to the instant backup. We will explain the corresponding steps in detail in Section IV-A, IV-B and IV-C, respectively.

A. Deriving feasible backup sets F(bb) for all possible paths

Theoretically, the backup can take place at any instruction during the program execution. In this work, we assume all the backup positions are at the end of basic blocks due to the following reasons. First, write operations to NV memory are much more energy-consuming than instruction executions. Based on the statistic of power stone benchmarks, one basic block only contains 8 instructions on average. So it is reasonable to assume the available energy when receiving the energy warning can always support to the end of the current basic block. Second, function returns, which enable the lifetime termination of local variables and thus stack size reduction, always happen at the end of basic blocks. Last but not least, the stack analysis at the granularity of instructions is a heavy burden for both offline profiling and storage. Consequently, the unit of basic block is a better analysis granularity. In this section, we will show that, for a specific basic block *bb*, how to figure out all the possible backup positions regardless of runtime jumps, to form the feasible backup set *F(bb)* for each possible path.

In the context of energy warning detected within basic block *bb*, we can flexibly utilize the available energy *Ene*. Conventionally, all the energy are used for instant backup. In this work, we suggest to split it into two partitions, instruction execution and backup respectively, with the following equation.

$$Ene \geq Ene(Ins) + Ene(backup) \quad (1)$$

where *Ene(Ins)* and *Ene(backup)* represent the energy consumption for instruction execution and backup respectively. For the example in Fig. 2, *Ene(Ins)* refers to the energy consumed by executing the program from t_1 to t_2, and *Ene(backup)* refers to the energy for backing up the *main* function at t_2. Equation (1) guarantees a successful, i.e. *feasible* backup.

There are only a limited number of *feasible* backup positions for any basic block due to the limited available *Ene*. Fig. 5

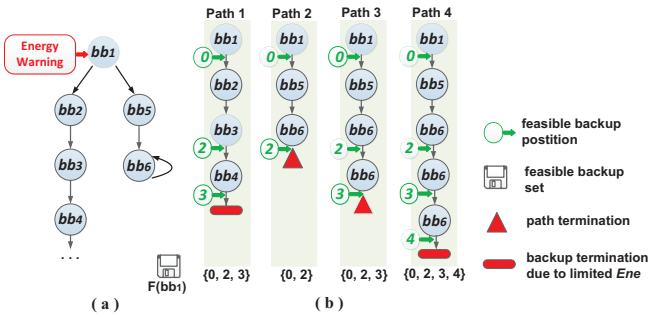

Fig. 5. An example of deriving the feasible backup set.

(a) shows the *CFG* of an example application. Assume that the energy warning is detected within basic block bb_1, we can enumerate all possible paths. Without loss of generality, for the left branch, we assume that *Ene* can support the program execution as far as bb_4; and for the right branch, *Ene* can support the program execution for bb_5 and three times of bb_6. Fig. 5 (b) illustrates all possible paths.

To examine the feasibility of backup at each point in the paths, we can derive *Ene(Ins)* by instruction counting and modeling in terms of energy consumption; and derive *Ene(backup)* by analyzing the active stack content at each point. Only positions satisfying Equation (1) are labeled as *feasible* backup candidates. As Fig. 5 shows, the feasible set for path 1 is $\{0, 2, 3\}$, which means feasible to back up after the current basic block (the end of bb_1), after subsequent 2 basic blocks (the end of bb_3), and after subsequent 3 basic blocks (the end of bb_4). The end of bb_2 is not feasible due to the large stack size to back up at this point, resulting in a large *Ene(backup)* and Equation (1) cannot be satisfied. The large stack size may result from a large number of function calls. Similarly, the feasible backup set for path 2, 3, and 4 are $\{0, 2\}$, $\{0, 2, 3\}$ and $\{0, 2, 3, 4\}$, respectively. Both the detailed instructions and stack utilizations at each point can be attained by offline static analysis.

B. Determining the backup position B(bb) through MinCF

Based on Section IV-A, the feasible backup set *F(bb)* for each possible path can be derived. As discussed, finding an optimal position for each path will impose huge storage and performance overhead. Thus we propose to determine one specific number for a basic block, *B(bb)*, to guide runtime backup. We propose a strategy called *MinCF*.

Table I summarizes the procedures to attain $B(bb_1)$ in Fig. 5 through the *MinCF* strategy. By examining the feasible backup sets from all possible paths, the common factors, *CF*, are obtained by keeping the common feasible backup positions among all paths, as listed in the second column in Table I. Among positions in set *CF*, the one with the minimum backup size is selected as $B(bb_1)$, as shown in the third column. To make the decision between position 0 and 2, we need to evaluate the expected static size at both positions. By considering all possible paths, *stackSize(0)* is evaluated by

the maximum stack size among {*backup position 0 in path 1, path 2, path 3 and path 4*}, and *stackSize(2)* is the maximum stack size among {*backup position 2 in path 1, path 2, path 3 and path 4*}. The stack size for one certain position is selected as the maximum value for all possible paths at runtime.

TABLE I
DETERMINING $B(bb_1)$ IN FIG. 5 THROUGH *MinCF* STRATEGY.

Feasible backup sets	*CF*	*MinCF*: $B(bb_1)$
Path 1: {0, 2, 3} Path 2: {0, 2} Path 3: {0, 2, 3} Path 4: {0, 2, 3, 4}	{0,2}	0, if *stackSize(0)*< *stackSize(2)*; 2, otherwise.

Based on *MinCF*, the *B(bb)* for each basic block can be derived. By attaching the *B(bb)* information to each basic block after static profiling, at runtime, the backup position can be decided by continuously executing subsequent $B(bb_i)$ basic blocks, if the energy warning is detected within the current basic block. Note that the *B(bb)* decision of a basic block may not be the optimal solution for all runtime cases. It is essentially a tradeoff between the NV register file size and storage/performance overhead, enabling efficient runtime decision by only attaching one number to each basic block.

Even though our evaluation confirms the extremely small chance for *CF* to be an empty set, we will show a solution when it happens. As Fig. 6 shows, if the feasible backup sets of the two possible paths are {3} and {2}, respectively, one *nop* basic block can be inserted into the right path to change the original backup position to basic block 3. Then the $B(bb_1)$ will be decided as 3 based on *MinCF*. Note that once a nop basic block is inserted, all related *B(bb)* need to be recalculated due to the modified *CFG*.

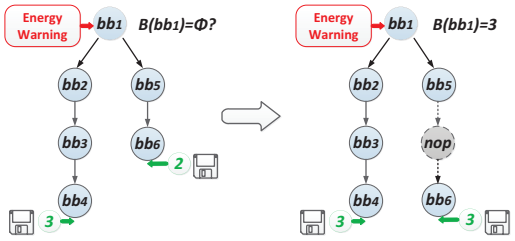

Fig. 6. Inserting *Nop* basic blocks to handle with empty common factors.

At runtime, once an energy warning is detected, let's say, in basic block bb_i, the value of $B(bb_i)$ will be read out to guide the program execution to continue for $B(bb_i)$ basic blocks. We can identify basic blocks by terminator instructions, e.g. *ret*, *jmp* and *jnz*.

C. Addressing the required NV register file size

The *B(bb)* numbers indirectly determines the required NV stack size for corresponding basic blocks. From the view of system design, the NV register file size needs to guarantee the successful backup whenever the energy warning is detected. Thus the NV register file size should be the lower-bound of the maximum *B(bb)* values, as stated in Line 5 of Algorithm 1. The concluded NV register size for stack backup can be referred to for application-specific chip designs to reduce the chip size.

V. EXPERIMENTS

In this section, we will present the experimental evaluation to assess the efficacy of the proposed NV stack reduction scheme.

A. Experiment setup

In the experiments, an NVP architecture similar to NVP [4] is applied. The processor contains 128-byte volatile space, including 100 bytes of stack and 28 bytes of registers. The write energy for FRAM is 10nJ/byte based on data sheet from cypress [16]. Based on the data we collected from NVP in [4], the energy consumption for each instruction execution is set to be 0.32nJ. The size of capacitor is set to be sufficient to back up all 128-byte volatile data when receiving energy warnings. The benchmarks are from *powerstone* suite [15], whose program sizes are suitable for application-specific embedded systems.

We implement the proposed scheme to reduce the stack size out of the 100 bytes, and for the remaining 28 bytes, we always back them up. We evaluate the NV register reduction by trace-based path analysis. The program interpretation is supported by LLVM [17]. The baseline is the conventional methodology to instantly back up all stack contents.

B. Evaluation results and discussions

Table II lists the NV stack size reduction for tested benchmarks. It shows that the proposed scheme delivers 62.9% stack reduction on average, when compared with conventional instant backups where all 100-bytes stack are backed up.

TABLE II
NV STACK SIZE REDUCTION

Benchmark	NV register size for stack	reduction
adpcm	28 byte	72%
bcnt	24 byte	76%
blit	48 byte	52%
crc	44 byte	56%
engine	20 byte	80%
fir	12 byte	88%
g3fax	65 byte	35%
pocsag	56 byte	54%
Average	37.1 byte	62.9%

To take a deeper look at the stack behaviors, we also examine the dynamic stack usage, as summarized in Fig 7. In addition to the total stack size, we also show the maximum dynamic stack size for each test bench, as well as the stack size after integrating the proposed scheme. It can be observed that it is suboptimal to back up all 100-byte stack since for most test benches, not all of the stack size is under occupation. Actually only 66% of the stack is active on average. The proposed scheme can further reduce the stack size by 29% through optimizing the backup positions. To figure out the reason behind various stack reductions for different test benches, we study the stack utilization characteristic for three typical

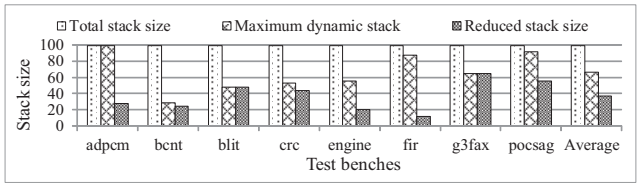

Fig. 7. Comparison among total stack size, the maximum dynamic stack size, and the reduced stack size as proposed.

benchmarks *fir*, *blit* and *g3fax*, as illustrated in Fig. 8. For *fir*, the dynamic stack size shows frequent and dense fluctuations, providing possibilities to find a backup position with smaller stack size and thus delivers significant stack size reduction. For *blit*, the stack size remains consistent due to the fixed function call, so the proposed scheme can only back up all the active stack content. Even though there are stack size changes in *g3fax*, the available energy cannot support the searching to the stack reduction positions since they are too far away. Consequently, the proposed scheme cannot achieve a smaller backup size than the dynamic stack for *g3fax*. It is observed that most benchmarks have similar characteristics with *fir*, and thus generally the proposed scheme can achieve promising stack size reduction.

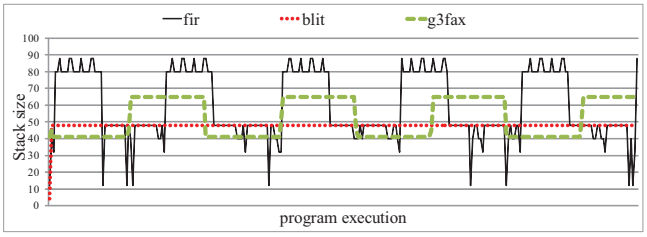

Fig. 8. The dynamic stack size along program executions for three typical benchmarks, *fir*, *blit* and *g3fax*.

We also test the cases with more available energy when receiving energy warning. The stack size reductions achieve 67.0% and 69.7% when 10% and 20% more energy are available. The reduction increases with more energy because the proposed scheme can search more backup positions with more energy available and thus can figure out better positions with less content to copy to the NV register file.

C. Overhead analysis

1) Storage overhead: The proposed scheme attaches a number to label the backup positions for each basic block. Based on our statistic, for one benchmark, there are a total of 420 different numbers to store in maximum, indicating at most 9 bits of storage overhead for one basic block. We calculate the storage overhead by bit number introduced for all basic blocks divided by the total code size, giving it 3.8% on average.

2) Performance overhead: At runtime, the logic we add is that once the energy warning is detected, the value of *B(bb)* for the current basic block is read out and the program continues for the subsequent *B(bb)* basic blocks. Thus the overhead is one extra *read* operation for each backup and a counter to trace number of basic blocks, which is trivial compared with the program execution.

VI. Conclusion

In this work, we propose a software assisted non-volatile register file reduction scheme for energy harvesting based wearable devices. Different from conventional instant backup upon energy warnings, we propose to identify more efficient backup positions in terms of stack size. To achieve this, a set of approaches of offline analysis and profiling are developed to determine the satisfactory backup positions for each basic block by considering all possible execution paths. The backup positions are attached to basic blocks for runtime reference. The proposed strategy can significantly reduce the non-volatile register size for stack backup. The evaluation results show a 62.9% stack size reduction on average.

Acknowledgment

The work described in this paper was partially supported by a grant from City University of Hong Kong (Proj. 9231168), 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.

References

[1] I. Lee and O. Sokolsky, "Medical cyber physical systems," in *DAC*, 2010, pp. 743–748.
[2] S. Sudevalayam and P. Kulkarni, "Energy harvesting sensor nodes: Survey and implications," *IEEE Communications Surveys Tutorials*, vol. 13, no. 3, pp. 443–461, March 2011.
[3] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power management in energy harvesting sensor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 4, Sep. 2007.
[4] 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 *ESSCIRC*, 2012, pp. 149–152.
[5] X. Sheng, Y. Wang, Y. Liu, and H. Yang, "Spac: A segment-based parallel compression for backup acceleration in nonvolatile processors," in *DATE*, 2013, pp. 865–868.
[6] N. Sakimura, T. Sugibayashi, R. Nebashi, and N. Kasai, "Nonvolatile magnetic flip-flop for standby-power-free socs," in *IEEE Custom Integrated Circuits Conference*, Sept 2008, pp. 355–358.
[7] Y. Wang, Y. Liu, Y. Liu, D. Zhang, S. Li, B. Sai, M.-F. Chiang, and H. Yang, "A compression-based area-efficient recovery architecture for nonvolatile processors," in *DATE*, 2012, pp. 1519–1524.
[8] Y. Wang, Y. Liu, S. Li, X. Sheng, D. Zhang, M.-F. Chiang, B. Sai, X. Hu, and H. Yang, "Pacc: A parallel compare and compress codec for area reduction in nonvolatile processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 7, pp. 1491–1505, July 2014.
[9] X. Jiang, J. Polastre, and D. Culler, "Perpetual environmentally powered sensor networks," in *IPSN*, 2005, pp. 463–468.
[10] T. Starner, "Human-powered wearable computing," *IBM systems Journal*, vol. 35, no. 3.4, pp. 618–629, 1996.
[11] M. Zwerg, A. Baumann, R. Kuhn, M. Arnold, R. Nerlich, M. Herzog, R. Ledwa, C. Sichert, V. Rzehak, P. Thanigai, and B. Eversmann, "An 82ua/mhz microcontroller with embedded FeRAM for energy-harvesting applications," in *ISSCC*, Feb 2011, pp. 334–336.
[12] J. Wang, Y. Liu, H. Yang, and H. Wang, "A compare-and-write ferroelectric nonvolatile flip-flop for energy-harvesting applications," in *ICGCS*, June 2010, pp. 646–650.
[13] M. Xie, C. Pan, J. Hu, C. J. Xue, and Q. Zhuge, "Non-volatile registers aware instruction selection for embedded systems," in *RTCSA*, 2014, pp. 1–9.
[14] M. Xie, C. Pan, J. Hu, C. Yang, and Y. Chen, "Checkpoint aware instruction scheduling for prolonging the lifetime of nonvolatile registers in multiple functional unit processors," in *ASPDAC*, 2015.
[15] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power m*core architecture," 1998.
[16] Cypress: http://www.cypress.com/.
[17] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *CGO*, March 2004, pp. 75–86.