Surviving Transient Power Failures with SRAM Data Retention

Songran Liu[1,2], Wei Zhang[2], Mingsong Lv[2*], Qiulin Chen[3], and Nan Guan[2]

[1]Northeastern University, China
[2]The Hong Kong Polytechnic University, Hong Kong
[3]Huawei Technologies Co., Ltd., China

***Abstract*—Many computing systems, such as those powered by energy harvesting or deployed in harsh working environment, may experience unpredictable and frequent transient power failures in their life time. The systems may fail to deliver correct computation results or never progress, as computation is frequently interrupted by the power failures. A possible solution could be frequently saving program states to non-volatile memory (NVM), such as using checkpoints, so that the system can incrementally progress. However, this approach is too costly, since frequent NVM writes is time and energy consuming, and may wear out the NVM device. In this work, we propose an approach to enable a system to use volatile SRAM to correctly progress in the presence of transient power failures, since SRAM is capable of retaining its data for seconds or minutes with the charge remained in the battery/capacitor after the CPU core stops at its brown-out voltage. The main problem is to validate whether the data in SRAM are actually retained during power failures. In our approach, we validate only a subset of the program states with Cyclic Redundancy Check for efficiency. The validation technique requires maintaining a backup version of the program states, which additionally provides the system with the ability to progress incrementally. We implement a run-time system with the proposed approach. Experimental results on an MSP430 platform show that the system can correctly progress on SRAM in the presence of transient power failures with low overhead.**

***Index Terms*—transient power failure, SRAM data retention**

I. Introduction

Transient power failure is a major issue that endangers dependability in many computing systems. For example, in energy harvesting IoT systems (powered by solar, wind, etc.), the system may experience frequent power failures due to the fluctuation of ambient energy input [1]. For systems deployed in harsh working environments, such as sensing systems in mine tunnels or on huge machines, shock and vibrations may cause the computing system to be frequently disconnected from the power source with a poor contact [2]. Recently, it was reported that the power management unit can be attacked, resulting frequent power downs [3], [4]. Transient power failures are unpredictable and occur frequently due to various unpredictable triggers (a transient power failure typically lasts for milliseconds or seconds [1]–[4]). They may cause a system to produce incorrect results, impede a system from progressing, and finally hinder the system from providing sustainable services. In mission critical systems, service interruption may even leads to catastrophic consequences.

When a power failure occurs, a common approach is to simply let the system restart. As transient power failures can be frequent, the system may never progress if frequent power failures cause frequent system restarts. To avoid this problem, one can save intermediate program states to non-volatile memory (NVM) [5], [6], so that when the system recovers, it can resume from a recent program point by reloading the states saved in NVM. Since transient power failures are highly unpredictable, state saving has to be frequently conducted to ensure incremental progress. However, frequently saving states to NVM is not practical for several reasons. First, reading and writing NVM are typically time and energy consuming, which adversely impede forward progress. Second, frequent writes to NVM may very soon wear out the NVM device.

The main objective of this work is to enable a system to correctly progress in the presence of frequent transient power failures without costly state saving to NVM. Specifically, we use volatile SRAM, equipped in most embedded processors, to save the program states across power failures with its *data retention* feature. Data retention refers to SRAM's ability to retain its data for seconds or even minutes with the charge remained in the battery/capacitor after the CPU core stops at its brown-out voltage [7].

Unfortunately, SRAM data are not guaranteed to be retained even with known power off duration, because the time of data retention varies from seconds to minutes depending on the hardware and the working conditions [8]–[10]. To leverage SRAM data retention to survive transient power failures, correctness validation to SRAM data must be provided.

In this paper, we propose a holistic approach to ensure both forward progress and logical correctness of a system under transient power failures using SRAM data retention. To ensure progress, we adopt a double buffering technique which efficiently backs up program states on SRAM after a program segment finishes execution. An even bigger problem is to validate that the states are correctly retained across power failures with low overhead. So, we propose a technique to dynamically validate only subsets of the program states with Cyclic Redundancy Check (CRC) before and after a power

This work was supported by Huawei Innovation Research Program "Collaborative Research Project on Computational Sensing", the Research Grants Council of Hong Kong (GRF 15204917 and 15213818), and Natural Science Foundation of China (grant No. 61772123 and 61672140).

*Corresponding author: Mingsong Lv. Email: mingsong.lyu@polyu.edu.hk

failure. We implemented a run-time system with the proposed approach. Experimental results conducted on an MSP430 platform show that the system can correctly progress on SRAM in the presence of transient power failures efficiently.

II. Overview

The main objective of this work is to ensure a system will make forward progress and provide correct computation in the presence of transient power failures. To this end, several design issues must be resolved.

A. Design Targets and Challenges

To ensure forward progress. Transient power failures are by nature unpredictable due to their triggers. If a power failure occurs, the CPU context will be immediately lost. Thus, the system can not resume from exactly where it was interrupted, and will generally experience a complete restart. However, restarting a system means losing finished work and re-executing the program, which is too costly. In our approach, we let the system incrementally progress across power failures with much less re-execution overhead.

To do this, a program is written as multiple program segments connected by control flows. At the beginning of a program segment, the program states will be backed up. If a power failure occurs in a program segment, the system will resume from the beginning of the failed segment by reloading the backup states. Furthermore, we choose not to save the backup states on NVM, because frequent NVM write is time and energy consuming and may severely impact the lifetime of the NVM. Instead, we save the backup states on volatile SRAM, as the SRAM data retention feature allows the memory data to survive seconds or even minutes of power down, which is generally enough for transient power failures.

To ensure logical correctness. Although SRAM can retain data across power failures, the retention time is very hard to predict since it can be affected by many factors including hardware features and working environments [8]–[10]. If after the system recovers, the backup data in SRAM is lost, the system will have to conduct a complete restart; otherwise, reloading incorrect states will lead to erroneous computation or system failure. To avoid this problem, we need to validate the memory data before and after a power failure.

Error detecting code (EDC), such as Cyclic Redundancy Check (CRC), is widely used for data validation in many applications. To validate memory, we can first compute an EDC checksum for the data before the power failure. After the system recovers from the power failure, a new EDC checksum is recomputed. If the new checksum is identical to the old one, and the data are not modified by the program after the old checksum is obtained, one can conclude that with very high probability the data are not changed during the power failure [11]. In this work, we choose CRC for memory validation, since almost all System-on-Chip processors are equipped with a hardware CRC component [12]–[14].

To ensure validation efficiency. Although CRC can be conducted with hardware, the time overhead to verify large data (e.g., the whole SRAM [10]), will be too high. If a checksum computation can not be finished between two consecutive power failures, memory validation is not possible, and thus the system can not continue. So we seek to minimize the amount of data to validate. As a program segment only modifies part of the program states, we partition states and only update the checksum for the partition that is modified. In Section III, we will provide technical details for such a design.

B. Overview of our Approach

We give an overview of our approach by Fig. 1 and explain it with an example in Fig. 2. For simplicity of the presentation, here we only demonstrate the main behavior of the system under our approach, and leave details to the next section.

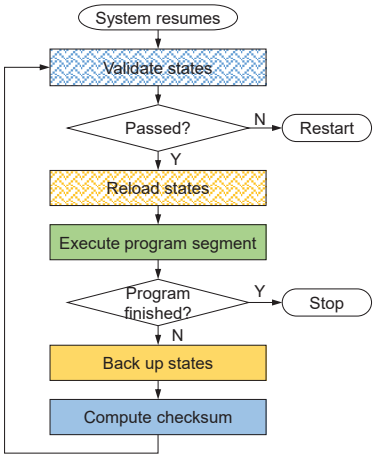

Fig. 1: The main work flow of the proposed approach

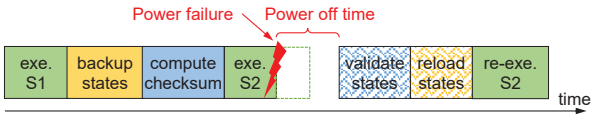

Fig. 2: An example to show the system behavior

In Fig. 2, the program starts with executing program segment $S1$, and $S1$ modifies a subset of the program states. After $S1$ is finished, the current states are backed up to SRAM as a restoration point. After that, a CRC checksum for the new backup is computed (note that we only compute the checksum for the subset of data that are modified). Assume during the execution of the next program segment $S2$ a power failure occurs. After the system recovers, it will resume from the most recent restoration point, i.e., the beginning of $S2$. During resumption, the system first validate the backup states on SRAM to check if data are lost during power off (only the data that will be used by $S2$ are validated). If the power off time is not long, data will be retained on SRAM and the validation will pass. The backup data are then reloaded, and the system continues execution from the beginning of the failed program segment $S2$. If the power off time is too long and the backup data are lost, a complete restart has to be conducted which restores the system to its initial state.

III. Design

In this section, we detail the design of proposed approaches: 1) *double buffering*, the technique to enable incremental progress and also to support efficient memory validation; 2) *partial CRC*, the technique for efficient memory validation.

Beforehand, we introduce the hardware and software models in this paper. The hardware has SRAM and a very small NVM to store CRC checksum data. The system is equipped with a power source, such as a battery or capacitor. After the CPU core browns out, it can continue supply the SRAM with energy. A program is constructed by a collection of program segments implemented as functions. Each program segment has its own stack initialized when the program segment is started, and all segments used shared data (implemented as global variables) to exchange information.

A. Double Buffering

Double buffering is a standard technique for state backup. The main principle is to maintain two versions of data. One version is stored in a *working buffer* which is operated by program execution; the other version is stored in a *backup buffer* to save the states at a certain program point for future restoration. Typically, the working buffer is allocated on SRAM, and the backup buffer is allocated on NVM for persistent storage. In this work, we allocate both buffers on SRAM to improve efficiency, and leverage SRAM data retention to provide persistence to the backup buffer.

1) Double buffering shared data: When a program segment is executed, a stack is allocated in main memory for temporary local variables of the segment. After the program segment finishes, the stack is destroyed. Only shared data, which are used by multiple segments and whose life time span across the whole program, need to be persistently maintained. In our design, shared data are managed by double buffering.

2) Efficient Buffer Management: At the end of a program segment, the up-to-date data values are stored in the working buffer. For state backup, one can actually copy the shared data from the working buffer to the backup buffer. The data copying takes time, and if a power failure occurs during the copying process, both buffers can be in an inconsistent state. So, at the time to update the backup buffer, we conduct a role swap (implemented by pointer swap) of the two buffers: the working buffer immediately becomes the new backup buffer and vise versa. Since the new working buffer contains old version data after the swap, before a new program segment starts execution, the system will always copy the data modified by the finished program segment from the new backup buffer to the new working buffer. Even though power failure may occur during this copying, as the backup buffer is not written by the program, the copying can be redone after the system resumes, as long as backup data are retained.

B. Partial CRC

As discussed in Sec. II, a system can not afford validating very large data, in that the time of CRC computation may not fit into the duration between two consecutive power failures. The system will not correctly execute without memory validation. So we try to minimize the data volume to validate.

After a program segment is finished, the shared data in the working buffer is updated with new values. The working buffer is soon swapped into the backup buffer. To enable validation to the backup buffer, a CRC check is conducted on the data in the new backup buffer. In our design, instead of conducting a CRC check to the whole buffer, we aim at only re-computing the checksum for the data that have been modified by the finished program segment. Since different program segments may access different subsets of the shared data, we dynamically maintain the memory partitions of the shared data resulted by program segment execution, and re-compute the CRC checksum for the changed partitions. We call the proposed technique *partial CRC*. Next, we first present the mathematics that enables us to do partial CRC, and then use a motivating example to show how partial CRC works.

1) The Mathematics for Partial CRC:

The computation of CRC takes the bit sequence of the data to be checked as the dividend, a pre-defined n-bit vector called generator polynomial as the divisor, and conducts a division over the $GF(2)$ domain [15]. The remainder is the CRC checksum. According to the mathematics over $GF(2)$, CRC check has the following properties [16].

Assume a block of data A is the concatenation of n sub-blocks, denoted by $A = A_1 \circ A_2 \circ \cdots \circ A_n$. Each sub-block A_i has x_i binary bits. We use G to denote the generator polynomial, and use $\otimes$, $\%$, $\oplus$ and $\ominus$ to denote multiple, remainder, addition and subtraction operations under $GF(2)$. The checksum for a data block A is denoted by $CS(A)$.

Property 1: The checksum for A can be computed from its sub-blocks with equation (1), where $z_i = \sum_{i+1}^{n} x_i$.

$$CS(A) = CS(A_1 \circ \underbrace{0 \cdots 0}_{z_1}) \oplus CS(A_2 \circ \underbrace{0 \cdots 0}_{z_2}) \oplus ... \oplus CS(A_n) \quad (1)$$

Property 2: Computing $CS(A \circ \underbrace{0 \cdots 0}_{z})$ from $CS(A)$.

$$CS(A \circ \underbrace{0 \cdots 0}_{z}) = (CS(A) \otimes ((1 \circ \underbrace{0 \cdots 0}_{z})\%G))\%G \quad (2)$$

As $((1 \circ \underbrace{0 \cdots 0}_{z})\%G)$ with different values of z can be computed and saved in advance, the computation of equation (2) can be conducted very efficiently. With equations (1) and (2), we are able to compute the checksum for a large data block with the checksums of its composing sub-blocks, and vise versa. For example, let $A = A_1 \circ A_2$, if $CS(A_1)$ and $CS(A_2)$ are known, $CS(A)$ can be easily computed from them; otherwise if $CS(A)$ and $CS(A_1)$ are known, $CS(A_2)$ can be easily computed from them.

2) Partial CRC:

We now use an example in Fig. 3 to show how partial CRC works, by which the CRC computation for the shared data in the backup buffer can be minimized.

Initially, the memory range of the shared data has only one partition $P1$. The checksum for $P1$, $CS(P1)$, is computed.

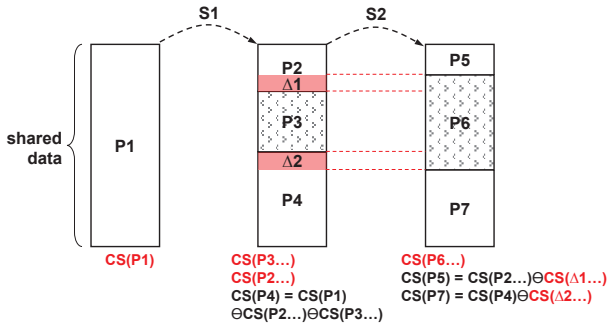

Fig. 3: An example to show the partial CRC technique

The execution of program segment $S1$ modified the data in partition $P3$, which creates another two partitions, $P2$ and $P4$ with unmodified data. The ideal case for high efficiency is to only redo CRC for $P3$, but not for $P2$ and $P4$. From now on, we maintain a checksum for each partition.

We discuss how to minimize the efforts to get the checksums for the three partitions. Currently, only the checksum for $P1$ is known. Assume $P4$ is the largest partition. We can conduct CRC check for $P2$ and $P3$ to obtain $CS(P2)$ and $CS(P3)$, and then leveraging *Property 1* and *2* to compute the checksum for $P4$ by $CS(P4) = CS(P1) \ominus CS(P2...) \ominus CS(P3...)$. This allows us to save a CRC check for $P4$. Here we use "..." to represent the zeros appended with the partition. The number of zeros is equal to the number of bits from the end of a partition to the end of the buffer.

In computing $CS(P4) = CS(P1) \ominus CS(P2...) \ominus CS(P3...)$, we need the old data in partition $P3$. However, data in $P3$ have been modified by the execution of segment $S1$. Note that we have two buffers for the data, and at the beginning of a program segment, the old data is stored in the backup buffer. We can use the old data in the backup buffer to compute $CS(P3...)$. The checksum for the new data in $P3$ is obtained by conducting a CRC check.

The program moves on and program segment $S2$ is executed. The data in $P3$, and partly in $P2$ and $P4$ are modified, creating partitions $P5$, $P6$ and $P7$. CRC check is performed on the new data in $P6$. Partition $P5$ differs with $P2$ by $\Delta 1$, so the checksum for $P5$ can be obtained by $CS(P5) = CS(P2...) \ominus CS(\Delta 1...)$. Similarly, the checksum for $P7$ can be obtained by $CS(P7) = CS(P4) \ominus CS(\Delta 2...)$. The checksums for $\Delta 1$ and $\Delta 2$ can be computed by CRC check on the old data in the backup buffer.

As can be seen, the double buffering mechanism not only ensures program progress, but also provides an indispensable capability to support partial CRC.

The execution of a program segment may access variables that are disjoint in memory address. We associate a memory range with each program segment that covers all its accessed data. Although a little redundancy may be introduced by the memory range, this handling significantly reduces memory partition management overhead.

To summarize, partial CRC has two main components: memory partition management and checksum computation. The first component maintains a dynamic list of memory partitions resulted from the execution of program segments; the second module computes a checksum for each memory partition with *Property 1* and 2 to reduce validation overhead.

Algorithm 1 The partial CRC Algorithm

Input: $\{P_1, P_2, ..., P_n\}$: previous memory partitions
Input: R: the memory range updated by the program segment
Output: New memory partitions with computed checksums

```
for each P_i do
    Switch (P_i's overlapping with R)
    Case 1: P_i partially overlaps with R
        Let P_i1 and P_i2 be the two new partitions
        Compute cs = CS(Min{P_i1, P_i2}) by CRC
        CS(Max{P_i1, P_i2}) = CS(P_i) ⊖ cs
    Case 2: P_i covers R
        3 partitions are resulted: P_i1, P_i2, P_i3
        assume P_i1 is the largest partition
        Compute CS(P_i2) and CS(P_i3) by CRC
        CS(P_i1) = CS(P_i) ⊖ CS(P_i2) ⊖ CS(P_i3)
    Replace CS(P_i) with the CS of the sub-partitions
Remove the checksum for the partitions covered by R
Remove the partitions covered by R
Compute CS(R)
Add a new partition R
Update the checksum for the whole shared data on NVM
```

The formalization of partial CRC is given by Algorithm 1. Each time a program segment is executed, only the memory partitions that either partially overlap with or cover the memory range R (accessed by the program segment) need recomputation. Case 1 and Case 2 correspond to the execution of $S1$ and $S2$ in Fig. 3, respectively. The main principle is to always apply *Property 1* and 2 to obtain the checksums for large partitions. After the checksums for the new partitions are obtained, the memory partitions are updated, and a new checksum for the whole shared data in the backup buffer is computed from the checksums of the partitions.

For storage of checksums, we save the checksums for memory partitions on SRAM and the checksum for the whole backup buffer on NVM. As data on SRAM may be lost during power failures, each time the system resumes, the checksum saved on NVM is firstly used to validate the checksums for memory partitions on SRAM; if correct, these checksums can then be used to validate the data in the backup buffer.

Since power failures may occur during CRC check and checksum update, checksums on SRAM and NVM are all double buffered. If the above situation occurs, one of the buffers persists the correct checksums, and the system is able to restore to a previous consistent program point.

For 16-bit CRC, only 4 bytes of NVM space is required to store the checksum for the whole backup buffer (Note that the checksum is double buffered). If the backup buffer is allocated on NVM, after a program segment finishes, the shared data will be written to NVM. As we allocate the backup buffer on

SRAM, only the checksum, orders of magnitude smaller than the shared data, needs to be written to NVM.

IV. Experiments and Evaluation

A. Experimental Setup

We designed a run-time system to implement the proposed approach with double buffering and partial CRC. The system is deployed on a TI MSP430 board with 12KB SRAM and FRAM-based non-volatile memory. The processor has a hardware CRC component and we apply 16-bit CRC. DMA is used for data copying between the working buffer and the backup buffer. A programmable power supplier is used to generate different power traces for evaluation.

We adopt benchmark programs[1] from the embedded systems domain [17] [18]: Activity Recognition (AR), Adaptive Differential Pulse-Code Modulation (AD), Bitcount (BC), software Cyclic Redundancy Check (CRC), Cuckoo Filter (CK), Fast Fourier transforms (FFT), RSA encryption (RSA) and Selection sort algorithm (SRT). Each program is decomposed into program segments. Each program segment S is declared by an API AEB(S), and the control flow from one segment to another is specified by a macro NEXT(S_i) in the source segment to specify a destination segment S_i. All the shared variables are grouped into a data structure. The memory range for each program segment is pre-compute offline.

B. Empirical Results and Evaluation

Experiments are conducted to evaluate the overhead of the implemented run-time system. First, we explore the overhead without power failures. Each program is run 50 times, and the total execution time is recorded. Fig. 4 shows the total execution time and its breakdown for each benchmark program. On average, the total overhead occupies around 20% of the total execution time. The first source of overhead comes from the time to do CRC check (accounting for $4 \sim 11\%$ of the total execution time). This overhead depends on the average size of shared data that a program accesses. For instance, program CRC has the lowest memory check overhead as it only has 36 bytes shared data. The second source of overhead is the time consumed on copying data between the working buffer and the backup buffer (accounting for $1 \sim 4\%$ of the total execution time). Note that copying data is much faster than conducting CRC check on the MSP430 platform. The third source of overhead comes from the run-time system itself, including the time for memory partition and checksum management.

Then, we explore system performance with re-execution in the presence of power failures. All benchmark programs execute with 5 different power traces with power cycles $10ms$, $50ms$, $100ms$, $200ms$, $400ms$, where power cycle is the average power on time. This setting covers different transient power failure scenarios [2]–[4], [18]. Fig. 5 shows the results, where execution times under these settings are normalized to that without power failures. The execution times of the benchmarks increase when power cycle becomes shorter. The reason

[1]Benchmarks available at https://github.com/ESLab20/surviving-TPF

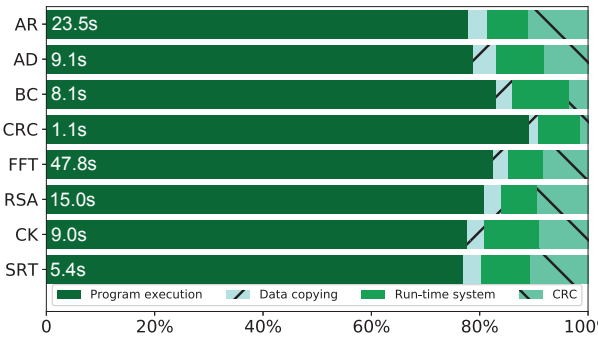

Fig. 4: Execution time breakdown for the benchmark programs

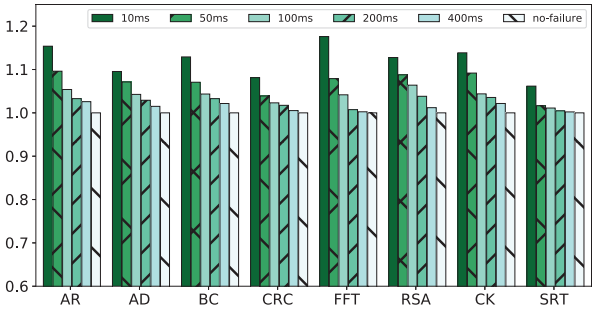

Fig. 5: Execution time under different power failure frequencies (normalized to no-failure case)

is, as power failures occur more frequently, the probability that a program segment encounters power failures increases, and thus the re-execution overhead becomes larger. Different programs have different sensitivity to the power cycles. For example, FFT is most sensitive, because it has very large program segments, and thus the re-execution penalty is larger when a program segment is failed by a power failure.

We compared our proposed approach with a related work called TotalRecall [10] which also uses SRAM data retention to ensure program progress. The main principle of TotalRecall is to predict the occurrence of a power failure, conduct CRC check for the whole SRAM, and then stop execution and wait for the power failure to occur.

As discussed before, transient power failures can not be precisely predicted due to various unpredictable triggers. The problem is, TotalRecall may produce imprecise predictions for transient power failures. If a power failure occurs before the predicted time, the SRAM may have not been checked. In this case, TotalRecall will experience a costly complete restart, hindering the system from progressing. Such behavior has been demonstrated by our experiments. Otherwise, if a power failure occurs after the predicted time, TotalRecall will cease to progress according to its working principle. To further explore the latter scenario, we define a metric, *Relative Error Distance*, denoted by ε, to model the prediction precision.

$$\varepsilon = \frac{T_{real} - T_{est}}{T_{est}} \tag{3}$$

where T_{est} (T_{real}) is the duration from the latest power up time point to the estimated (actual) power off time point, with $T_{est} \leq T_{real}$. A bigger ε indicates a more imprecise prediction. $\varepsilon = 0$ corresponds to the case of perfect prediction, which generally does not occur in reality.

We explore the performance under different prediction precision. As TOTALRECALL predicts power failures according to supply voltage decrease, stable power on durations are generally predicted for a given battery/capacitor. For each occurrence of power failure, we set $T_{est} = 50ms$, and let the actual power on duration fluctuate in range $[T_{est},\ T_{est} \cdot (1+\varepsilon)]$. Execution times for both approaches under different ε are shown in Fig. 6, with all execution times normalized to the case by OUR with $\varepsilon = 0$.

Our proposed approach outperforms TOTALRECALL in all ε settings. First, the CRC computation cost is too high by TOTALRECALL since the whole SRAM is checked by CRC. Even if power failure prediction is perfect (the $\varepsilon = 0$ case), the high cost impedes forward progress. Second, since the predicted power failure is earlier than the actual power failure, TOTALRECALL will stop after the CRC computation. By comparison, our system always makes progress.

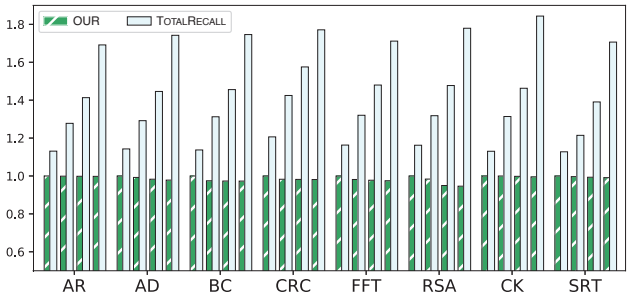

Fig. 6: Execution times by OUR and TOTALRECALL (normalized to OUR with $\varepsilon = 0\%$). For each program, results for four Relative Error Distance settings, $\varepsilon = 0\%$, $\varepsilon = 25\%$, $\varepsilon = 50\%$, $\varepsilon = 100\%$, are given from left to right.

V. RELATED WORK

Transient power failure is a critical problem to the dependability of many computing systems [1]–[4]. For the systems to survive transient power failures, system states have to be saved frequently to NVM. Techniques in related domain [5], [6] can be used for this purpose, but they all incur considerable writes to NVM, which is not only time consuming but may also cause the NVM device to be easily worn out. SRAM data retention [7]–[9] is a nice property that can be used to ensure system progress without relying on NVM. However, validating memory data across power failures is very time consuming. In a related work TOTALRECALL [10], the system conduct CRC check for the whole SRAM each time a check is scheduled. As shown in the experiments, the high overhead of checking the whole SRAM renders the system to be very inefficient in forward progress. This work solves the above problems in a different way. The proposed approach conducts efficient memory validation by checking only a small subset of the memory data with the support of a double buffering mechanism designed also for forward progress.

VI. CONCLUSION

This paper presents an approach to enable a computing system to survive transient power failures with very low overhead. The approach adopts double buffering to ensure a program will progress in the presence of frequent power failures. Notably the backup data are stored on SRAM and are persistence across power failures with the SRAM data retention feature. To validate whether data are correctly retained across power failures, a technique called partial CRC is proposed to conduct memory validation with very low overhead. Experimental results demonstrate that the proposed approach has low runtime cost and considerably outperforms related work.

REFERENCES

[1] D. Ma, G. Lan, M. Hassan, W. Hu, and S. K. Das, "Sensing, computing, and communications for energy harvesting iots: A survey," *IEEE Communications Surveys Tutorials*, 2020.

[2] M. Malewski, D. M. Cowell, and S. Freear, "Review of battery powered embedded systems design for mission-critical low-power applications," *International Journal of Electronics*, 2018.

[3] M. N. Islam and S. Kundu, "Pmu-trojan: on exploiting power management side channel for information leakage," in *ASP-DAC*, 2018.

[4] M. Ansari, J. Saber-Latibari, M. Pasandideh, and A. Ejlali, "Simultaneous management of peak-power and reliability in heterogeneous multicore embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[5] I. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, 2013.

[6] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *2nd Summit on Advances in Programming Languages*, 2017.

[7] H. Jayakumar, A. Raha, and V. Raghunathan, "Hypnos: An ultra-low power sleep mode with SRAM data retention for embedded microcontrollers," in *Proceedings of CODES+ISSS*, 2014.

[8] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu, "TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks," in *Proceedings of 21st USENIX Security*, 2012.

[9] N. A. Anagnostopoulos, T. Arul, M. Rosenstihl, A. Schaller, S. Gabmeyer, and S. Katzenbeisser, "Low-temperature data remanence attacks against intrinsic sram pufs," in *Proceedings of 21st DSD*, 2018.

[10] H. Williams, X. Jian, and M. Hicks, "Forget failure: Exploiting sram data remanence for low-overhead intermittent computation," in *Proceedings of 25th ASPLOS*, 2020.

[11] P. Koopman and T. Chakravarty, "Cyclic redundancy code (crc) polynomial selection for embedded networks," in *DSN*, 2004.

[12] "Texas instruments: Cyclic redundancy check module, referenced in sep. 2020," https://www.ti.com/lit/ug/slau398f/slau398f.pdf?ts=1600658052095.

[13] "Microchip: Cyclic redundancy check in microchip family, referenced in september 2020," https://www.microchip.com/design-centers/8-bit/peripherals/core-independent/cyclic-redundancy-check.

[14] "Stm32 family: Using the crc peripheral in the stm32 family, referenced in sep. 2020," https://en.wikipedia.org/wiki/STM32.

[15] R. Lidl and H. Niederreiter, *Finite fields*. Cambridge university press, 1997.

[16] M. Walma, "Pipelined cyclic redundancy check (crc) calculation," in *Proceedings of 16th ICCCN*, 2007.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of 4th WWC*, 2001.

[18] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent Execution without Checkpoints," in *Proceedings of OOPSLA*, 2017.