Energy-efficient Fault Tolerance Approach for Internet of Things Applications

Teng Xu, and Miodrag Potkonjak
Computer Science Department
University of California, Los Angeles
{xuteng, miodrag}@cs.ucla.edu

ABSTRACT

Fault tolerance (FT) is essential in many Internet of Things (IoT) applications, in particular in the domains such as medical devices and automotive systems where a single fault in the system can lead to serious consequences. Non-volatile memory (NVM), on the other hand, is commonly used to improve system reliability due to its unique properties to retain data even if the power supply is lost. However, one of the most important drawbacks of NVM is that it imposes significant overhead regarding timing and energy. In this paper, we have proposed a unique technique with the use of NVM to create FT application specific architecture with almost no timing overhead and low energy overhead.

We address the implementation of applications that are specified using synchronous data flow model of computation. We combine the use of NVM and classical CMOS transistors so that NVM judiciously stores selected complete states of the pertinent program. It allows the program to resume from the saved state in NVM when faults occur. The frequency of the state selection can be flexibly adjusted for an arbitrarily specified FT timing/energy overhead. Moreover, to find an optimal state selection (with low overhead), we have applied an improved min-cut max-flow algorithm. On a variety of typical benchmarks, the simulation results indicate that our approach incurs only a small overhead over lower bounds. It is also generic in a sense that it can be applied to a wide spectrum of underlying IoT architectures and computational models.

1. INTRODUCTION

The prevailing of Internet of Things (IoT) enables a variety of applications including smart homes, wearable devices, intelligent automotive, and so on. Such IoT applications have imposed new requirements on fault tolerance (FT) and energy management. On one hand, in some IoT applications such as implanted devices and airplane control systems, a single fault can be ultra dangerous and even life threatening. Therefore, to detect and to correct faults in such IoT applications have become especially important, in other words, FT is required. On the other hand, energy/power has become a primary design criterion for many IoT applications. It is because such IoT devices are normally highly constrained by their limited battery life. To summarize, the FT in IoT applications has to be addressed in a way with high energy efficiency.

NVM is a type of memory that can retain the stored data even after external power has been turned off. Common types of NVM include phase-change memory (PCM), magnetoresistive random-access memory (MRAM), resistive RAM (RRAM), and memristors. Such resistive memory technologies share the properties of high scalability, non-volatility, and high density. Due to the unique properties of NVM, it is commonly used to improve system reliability. The principal idea is to store program execution data in NVM as a checkpoint. Thus, the system can be recovered from a wide range of transient and permanent faults using the stored information.

NVM based FT approach provides a possible solution to enhance the reliability of IoT devices. It is especially suitable for the IoT applications that calculate real-time dataflow. For example, in applications such as self-driving cars, airplane navigation systems, or even fall detection systems for elderly or disabled people, they all require their data to be processed in realtime. And when a fault indeed occurs in the process, the system has to be recovered with an ultra-short period. NVM based FT is ideal for these IoT applications because it is much faster to restart a program execution with the most recent checkpoint and the set of inputs instead of falling behind trying to recover an older state from the beginning of the program.

However, one major problem of the FT mechanism using NVM is that it employs large timing and energy overhead. As shown in Table 1, NVM employs a significant cost in writing time and writing energy when compared to other memory technologies. Consequently, when used in FT, it is crucial to reduce the number of write access to the NVM. Meanwhile, it is also an important issue to schedule the writes at proper clock cycles to avoid congestion.

The goal of our paper is to build an energy efficient FT approach using NVM. Our key idea is to store some states of a program in the NVM cache in such a way that the program can be resumed from the saved state after faults occur. We have addressed and optimized two major issues in our approach, respectively:

- Where should each state be?
- How to schedule the write of states to the NVM cache?

ICCAD '16, November 07-10, 2016, Austin, TX, USA
 ISBN 978-1-4503-4466-1/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967034

Features	SRAM	eDRAM	MRAM	PRAM
Density	Low	High	High	Very high
Speed	Very fast	Fast	Fast read Slow write	Slow read Very slow write
Dyn.Power	Low	Medium	Low read High write	Medium read High write
Leak.Power	High	Medium	Low	Low
Non-volatile	No	No	Yes	Yes

Table 1: Comparison of different memory technologies.

Regarding the first question, the key requirement of state selection is to minimize the overhead. To address this issue, we have applied the max-flow min-cut algorithm on the data-flow model of the program. It is straightforward in the sense that each write to the NVM takes large timing and energy overhead. Therefore, the fewer data to write to NVM, the less overhead the system will have. Furthermore, to enable multiple states in a single program, we extend the standard min-cut algorithm to the "multiple min-cuts algorithm", where the total overhead of all the states is minimized.

The second question targets to solve the issue of slow write to NVM cache. According to Table 1, the write to NVM takes heavy timing overhead. Thus, it is important to minimize the influence of states writing to the overall execution speed of the program. In other words, the program execution should not be halted to wait for the write to NVM to finish. To addressed this issue, we combine the use of NVM and classical CMOS register files so that when multiple variables are generated at the same time, they are first temporarily stored in some dedicated register files which are fast and low-energy consuming. Then the program can continue without being influenced while the write from register files to NVM cache is executed simultaneously.

We summarize our contributions in the following.

- We have proposed an NVM based architecture for FT by storing recoverable program states.
- We leverage the issue of large energy and timing overhead to write to NVM by proposing min-cuts for state selection, so that the amount of data needs to be written to NVM is minimized. By introducing register files for temporary storage, the extra timing overhead is almost eliminated.
- Our approach is generic in a sense that it can be used to recover systems from a wide range of permanent and transient faults on many underlying IoT architectures and computational models.

The paper is organized in the following way. We first review the related literature in Section 2. Then we introduce our system model and assumptions in Section 3. The detailed algorithms for state selection and variable scheduling are discussed in Section 4. Lastly, we discuss our test results on a variety of benchmarks in Section 5.

2. RELATED WORK

We survey the related work along several research and development directions: non-volatile memories, fault tolerance, program slicing techniques, and max-flow algorithms. These efforts are intrinsically multi-disciplinary and due to space limitations we consider only most directly related technologies and mechanisms.

2.1 Non-volatile Memory

Several types of non-volatile memory have been developed and studied as potential alternatives for CMOS flip-flops, registers, cache, main memory as well as for long term storage including flash, ferroelectric RAM, phase change, and MRAM [1]. In our simulation, we have used spintronic memories, specifically spin-transfer torque RAM (STT-RAM) due to its attractive properties regarding integration flexibility, size, and relatively low latency and energy overhead [2]. Hybrid memory systems have seen also widely studied, and it has been demonstrated that they are an attractive alternative in several types of systems and workloads [3].

2.2 Fault Tolerance

The design of reliable and fault-tolerant circuits and systems has a long history since a landmark pioneering paper by Moore and Shannon was published in 1956 [4]. There is a number of excellent related books including recent ones [5][6]. Three dominant aspects of fault-tolerant systems are fault abstractions into models (e.g. soft upsets [7], delay faults, and power supply reduction), fault detection, and fault recovery. Our focus is on fault recovery. We consider all types of faults such as failure of the power supply and soft upsets that can be fully resolved through the use of NVM. More recently, a memory architecture to enable NVM-based checkpoints for FT is proposed by Kannan et al. [8]. Compared to their work, our paper has emphasized on using an algorithmic way to find optimal positions of NVM-based checkpoints.

2.3 Program Slicing

A slice of a program can be defined as an executable subset of the program that computes the same functions as the initial program for a selected subset of variables [9]. Techniques for program slicing have emerged in the seventies as popular mechanisms for program debugging [10][11]. They are still essential mechanisms for the analysis of software products [12].

The availability of several systems such as shared memory multiprocessors with global checkpoints is optimized for on-line processing systems [13]. A checkpoint consists a set of variables required for a program to restart after a fault detection. In CAD literature, a special type of checkpoint named cut was used for the export of difficult to observe variables and for the injection of difficult to control variables, such that to enable high speed debugging of complex integrated circuits. Under the assumption of static synchronous dataflow computational model, Kirovski et al. defined a program cut as a system of variables that are all stored in a minimal number of registers and require minimal additional interconnect network for their complete controllability and observability [14].

2.4 Max-flow Algorithm

Our main optimization step is related to the min-cut max-flow theorem. The theorem has been independently discovered by Elias, Feinstein, and Shannon and by Ford Fulkerson in 1956. A comprehensive coverage of the algorithms for a variety of network flow problems is given in a book by Ahuja, Magnati, and Orlin [15] as well as many other

books. We used linear programming formulation from Papadimitriou and Steiglitz [16] that is both simple and very flexible with respect of modifying what type of min-cuts are required. Finally, it is interesting to notice that retiming for minimization of the number of flip-flops by Leiserson and Saxe can also be used for this tasks and sometimes leads to even more flexible formulations [17].

3. SYSTEM MODELING

3.1 Overview

Our approach works on the platform of application-specific integrated circuits (ASIC). The program execution is performed on the regular datapath while the outputs of functional units can be stored in the NVM based cache. Figure 1 shows the overview of our system. States of the executed program are captured with some particular frequency and are stored in the NVM based cache. When any fault occurs, the program can be recovered with the stored states. Note that we use two NVM spaces in the cache to store program states interchangeably. Each state is only written to a single NVM space, and it overwrites the previous state stored in the space. Such mechanism guarantees that when faults occur in the middle of states write, at least one NVM space in the cache stores a complete state. We also assume that there exists a single port to write data from the datapath to the cache. As a result, the schedule of data write becomes crucial (to avoid congestion).

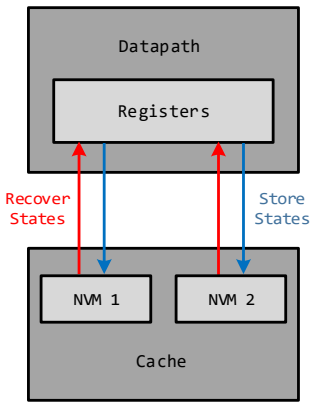

Figure 1: System overview.

3.2 A Motivational Example

The graph G in Figure 2 shows an example data flow graph where I_1 to I_4 are inputs and O_1 to O_2 are outputs. A state in the graph is defined as a set of variables that cuts all possible paths from primary inputs to outputs in the computation. Graph G' depicts the adjustment of the original data graph G in order to fit for the min-cut max-flow algorithm to find a state. Two major changes are made. The first is to add a source node and a sink node to the graph. The second is to create nodes for each arithmetic unit as well as primary inputs and final outputs. Correspondingly, we also add edges to connect the source and the primary inputs as well as edges to connect the outputs and the sink.

Therefore, the goal of the state search algorithm is to, given a computation control data flow graph, find a register

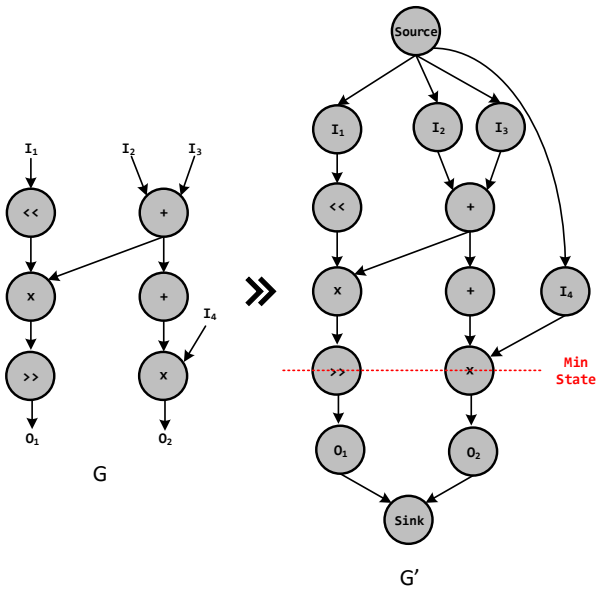

Figure 2: A motivational example of dataflow graph and the corresponding min cut.

subset of minimal cardinality that stores all the variables of at least one complete cut. To find such state, we have applied our modified min-cut algorithm and the minimal state is shown in G' of Figure 2 (assume that each functional unit generates the same amount of data). More details on our min-cut algorithm are illustrated in Section 4.

4. APPROACH

We address our approach of solving two optimization problems in this section. The first is state selection with the motivation to minimize the amount of data that needs to be written to NVM. The second issue is variable scheduling. As we demonstrated in the system overview, only a single port is available to write to cache. Considering that each write to NVM employs a large timing overhead, thus when to write and what to write to the NVM have become an important issue.

4.1 State(s) Selection

Our key algorithm for state selection is through the min-cut max-flow algorithm. The insight is that we want to minimize the energy spent to write a state to the NVM. The required energy is proportional to the amount of data to write. It is a natural idea to apply the min-cut algorithm as it directly returns the state with least data. However, there still exists some modifications we need to make to the standard min-cut algorithm to suit our problem. The first problem is that standard min-cut algorithm applies cut on the edges while in our problem, we should cut nodes since we only need to store a single copy of the arithmetic unit result. To address this issue, we need to convert the original data flow graph to a new graph and then apply the standard min-cut algorithm. The other issue is regarding the number of cuts in the graph. As min-cut only returns the optimal cut across the circuit, it is not addressed in the standard algorithm about how to minimize the total amount of data when multiple cuts are required.

In the following parts, we first discuss our algorithm to find a single optimal min-cut state on the graph, then we extend our algorithm to select multiple states.

4.1.1 Single State Selection

We formally explain our formation of data flow graph. Following the same notation in Section 3.2, we create a directed graph $G' = (V', E')$ where each node $v \in V'$ represents a primary input or an arithmetic unit and each edge $e \in E'$ represents a flow of data between two nodes. Each node in the graph has an attribute *capacity* which describes the data size of the node. For a node of a primary input, the capacity represents the size of the input; for a node of an arithmetic unit, the capacity represents the data size of the unit output. Each edge (from node i to node j) also has the *capacity* attribute which represents the size of data transmitted from node i to node j. We virtually add a source node which has edges connecting to all the primary inputs and a sink node that is linked with all the program outputs. The capacity of all the edges that are directly connected to source or sink is initialized to ∞.

Our goal is to find nodes based min-cut on graph G', which means that the cuts should be applied on nodes instead of edges. To be consistent with the standard min-cut algorithm model which only has capacity constraints for edges, we transform our data flow graph $G' = (V', E')$ to $G'' = (V'', E'')$, where each node v in G' breaks into two nodes v_1, v_2, as well as an edge from v_1 to v_2 to carry the original capacity of node v. After the above step, the node capacities are converted to edge capacities.

Figure 3 shows the transformation from graph G' in the motivational example to graph G''. In G'', each solid edge (black) from node i to node j represents the flow of data from unit/input i to unit/output j. The capacity of such edges is set to ∞. Each dashed edge (red) in G'' is an added edge inside each node of graph G'. The capacity of a dashed edge equals to the capacity of the node it corresponds to, which equals to the size of data the node outputs.

When applying the standard min-cut algorithm on graph G'', it naturally finds a minimal complete cut over only the dashed edges since all the solid edges have the capacity of ∞. Therefore, the corresponding nodes in G' that map to the min-cut edges in G'' are selected to form a complete state. Due to the nature of min-cut, the amount of data that is generated by such state is minimized.

One problem of the above-proposed max-flow based state selection is that the cut result can be highly biased to inputs or outputs. For example, in our graph shown in Figure 3, although the cut we denote in G' indeed finds an optimal state in terms of energy saving, the problem is that the state is too close to sink. To be more specific, the selected state directly saves the program outputs. Consider a fault occurs during the program execution (before the program outputs are generated), our state can not be used as a checkpoint to resume the program. On the other hand, a state too close to primary inputs is also questionable. It is because in such cases when the program resumes from the state, almost all the operations in the program need to be recalculated in which case a state does not help anything.

As explained above, it is important to have a state somewhere in the middle of the program flow, although this can sacrifice the size of cut to no be longer optimal. Here we formally define the position requirement of our desired state s.

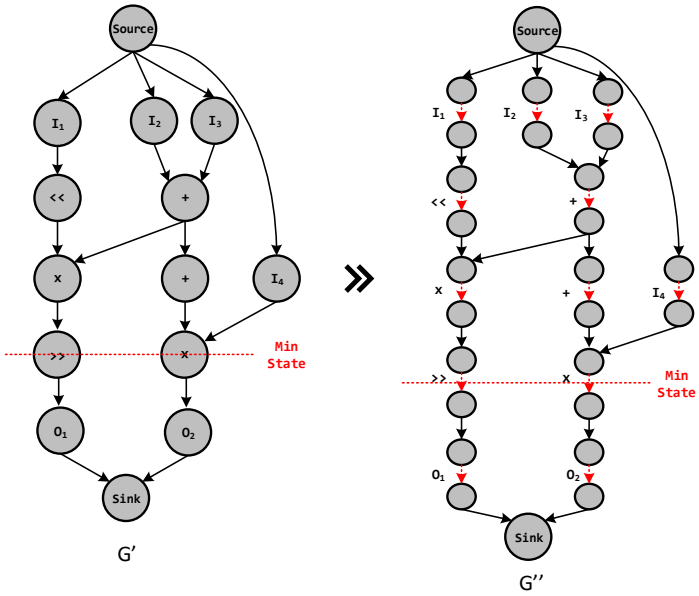

Figure 3: Graph transformation to find nodes based min-cut on the motivational example shown in Figure 2.

The maximum distance from the source to any node in the state (d_{src-s}) has to be smaller than α, and the maximum distance from any node in the state to sink (d_{s-snk}) has to be smaller than β. α and β are constants decided by the size of the graph. Using the above notation, our objective is shown in Equation 1.

$$\begin{aligned} &\text{Minimize: } Capacity(s) \\ &\text{Subject to} \\ &\qquad d_{src-s} > \alpha \\ &\qquad d_{s-snk} > \beta \\ &\qquad \alpha,\ \beta \textit{ are constants} \end{aligned} \tag{1}$$

We propose Algorithm 1 to find a state that meets the above criterion. The basic idea is to iteratively find the top n min-cuts on the graph until a cut meets the position constraint. To find the top n optimal min-cuts ($n > 1$), we iterate through the edges from the 1st optimal min-cut until the $n-1$th optimal min-cut, assign the capacity of selected edges to ∞, and then finds the corresponding min-cut. The nth optimal min-cut is the minimal cut among all the generated min-cuts in the nth iteration. By assigning edges to ∞, we avoid the min-cut algorithm finding the min-cut solutions the same as the top $n-1$ optimal min-cuts. Thus, the cut we achieve comes as the nth optimal min-cut. The search for the top n cuts continues until a cut is found to meet the position constraint.

4.1.2 Multiple States Selection

In the previous discussion of state selection, we assume that there only exists a single cut in the program. In a real scenario, multiple states selection in the program is commonly required. When the frequency of state selection is high, more energy needs to be spent to write states to NVM. Meanwhile, as the distance between different states

Algorithm 1 Single State Selection

Input: Graph $G'' = (V'', E'')$, constants α, β.
Output: Optimal state selection s.

```
Find min-cut s on Graph G''.
Set_s = ∅.
Append s to Set_s.
While d_{src-s} < α or d_{s-snk} < β:
    Set_e = ∅.
    For all cuts s_i in Set_s:
        Select e_ij from s_i.
    Endfor
    G''_modi = G''.
    For each combination of e_ij:
        Modify G''_modi with capacity(e_ij)=∞.
        Find min-cut c on G''_modi.
        Append c to Set_e.
    Endfor
    c_min = c_0 in Set_e.
    For each cut c_i in Set_e:
        If capacity(c_i)<=capacity(c_min):
            c_min = c_i.
        Endif
    Endfor
    s = c_min.
    Append s to Set_s.
Endwhile
Return s.
```

decreases, the expected energy that is required to resume program after faults occur is reduced. There exists a trade-off between the write energy and the resume energy.

The first important issue in multiple states selection is to decide the number of states k. To quantitatively decide an optimal k, we define the following two concepts.

- E_{write}: Energy spent to write the data of a state to NVM.
- E_k: Given k total states ($k > 1$), when faults occur, the expected energy required to recover the program.

We should increase the number of states (k) only when $E_k - E_{k+1} > E_{write}$. As the number of states increase, the average distance between the neighbor states is reduced, thus, on average it takes less computational efforts to recover the program when faults happen. The left side of the equation $E_k - E_{k+1}$ represents the expected energy saving of program recovery by increasing the number of states from k to $k+1$. The right side of the equation E_{write} is the energy required to add a state. Only when the energy saved by increasing a state is larger than the energy overhead, we increase the number of states by 1. A program with long clock cycles but relatively "thin" states tends to have a larger k. It is because in such cases, the difference from E_k to E_{k+1} is large (long execution flow) while the E_{write} is small ("thin" states). As k increases, the difference between E_k and E_{k+1} becomes smaller and smaller, and is eventually converged to be below E_{write}. In most benchmarks, because the write to NVM is energy expensive compared to the calculations in the datapath, k is only set to be a small value.

After k is configured, similar to the single state selection, multiple states selection imposes requirements for the data size of the cut as well as the positions of the cut. Ideally, the total amount of data in multiple states should be as small as possible under the constraint that the distance between each state is larger than some threshold γ. Note that the distance between state A and state B is defined as the average mutual distance between the nodes in A and the nodes B. Depending on the total number of states (k), γ needs to be adjusted to make sure that a k-states solution exists. The above problem can be formulated using the following notations.

$$\begin{aligned} &\text{Minimize: } \sum_{i=1}^{i=k} Capacity(s_i) \\ &\text{Subject to} \\ &\quad d_{src-s_1} > \alpha \\ &\quad d_{s_k-snk} > \beta \\ &\quad d_{s_i-s_{i+1}} > \gamma \ (0 < i < k) \\ &\quad k,\ \alpha,\ \beta,\ \gamma \textit{ are constants} \end{aligned} \tag{2}$$

We employ a similar algorithm as the single state selection to solve the above problem. The detailed algorithmic flow is shown in Algorithm 2. The key idea is to incrementally find top min-cuts in the data flow graph, and then search for all the k cuts combinations such that if the cuts in the combination can satisfy position constraints, the k cuts form a candidate multiple states selection and is appended to Set_m. Eventually, we traverse through all the candidates in Set_m to find the candidate with the smallest sum of cut capacity and use it as our final solution.

4.1.3 *Influence of Unfolding*

Many programs are designed to run multiple iterations, such as various filters in DSP applications. State selection in these programs commonly faces a major problem that each iteration of the program may only have very limited computations. Accordingly, it is not worth it to assign a new state for every iteration. For example, it is possible that $E_0 - E_1 < E_{write}$, making no state is needed per iteration. Therefore, there is a need to update our state selection strategy for such programs.

Unfolding addresses the above concerns. First proposed by Parhi and Messerschmitt in 1989 [18], unfolding is a transformation technique which duplicates the functional units to increase the program throughput while preserving its functional behavior at its outputs. For an unfolding factor J, the core idea of unfolding is to duplicate all the functional units in the original program by J times and reconnect everything without altering program functionality. It produces the same outputs compared to running the original program by J iterations, but generally offers higher throughput and smaller iteration period.

By applying unfolding on iterative programs, we can achieve a data flow graph over multiple program iterations (J). Note that J can be adjusted to suit the need of state selection. On one hand, the graph is greatly expanded in such a way that a state can be selected in every J iterations, avoiding the situation where no state is needed for a single iteration. On the other hand, it provides a more global picture for state selection. We are able to flexibly pick the optimal number of states as well the optimal positions of states using our single state/multiple states selection algorithm.

Algorithm 2 Multiple State Selection

Input: Graph $G'' = (V'', E'')$, constants k, α, β, γ.
Output: Optimal multiple states selection s_i, $i \in \{1, 2...k\}$.

```
Find min-cut s on Graph G''.
Set_s = ∅
Append s to Set_s.
check = 1.
While check:
    Set_e = ∅
    For all cuts s_i in Set_s:
        Select e_ij from s_i.
    Endfor
    G''_modi = G''
    For each combination of e_ij:
        Modify G''_modi with capacity(e_ij)=∞.
        Find min-cut c on G''_modi.
        Append c to Set_e.
    Endfor
    Find c_min in Set_e with the smallest capacity.
    s = c_min.
    Append s to Set_s.
    Set_m = ∅
    For all k cuts combinations (c_1 to c_k) in Set_s:
        If c_1 to c_k satisfy d_src-c1 > α, d_ck-snk > β,
        and d_ci-sc(i+1) > γ (0 < i < k):
            Append {c_1 to c_k} to Set_m.
        Endif
    Endfor
    If size(Set_m)>0:
        check = 0.
        Select c_1 to c_k in Set_m with the smallest sum.
        s_i = c_i, i ∈ {1, 2, ...k}.
    Endif
Endwhile
Return s_i, i ∈ {1, 2, ...k}.
```

4.2 Variable Scheduling

Our design requires writing all the variables in the states to the NVM cache through a single port within a single or multiple computation iterations. However, as shown in Figure 1, the write speed to NVM is slow, making the state writing to NVM becomes a bottleneck of the program execution. Note that when the system writes the data of a state from the registers to the NVM cache, the rest of the program has to halt to avoid overwriting the registers.

We leverage the above issue using temporary register files. Instead of directly storing data from registers in the datapath to NVM, we first transfer the data temporary to a register file, and then store it to NVM. Although the mechanism introduces extra hardware, the advantage comes in two aspects. The first is that the program no longer needs to wait for the write to NVM to finish, which employs large timing overhead. The write from the temporary register file to NVM can be executed in parallel with the program running. The other advantage is that it completely avoids the situation of overwriting. As the temporary register file holds all the data of a state, the original registers that hold the data are free to be used by other functional units. Our mechanism will introduce small timing overhead which accounts for the write from regular registers to the temporary register file, however, compared to the huge time saving caused by writing to NVM, the overhead is negligible.

The variable scheduling of the above scheme can be split into two parts, respectively to schedule the write from regular registers to register files, and to schedule the write from the register files to NVM. The basic policy is generalized in the following.

- Regular register → register files: timing order of variables.
- Register files → NVM: timing order of states.

In terms of the scheduling from the regular register to register files, it should strictly follow the timing order of the generation of variables, so that the regular registers can be immediately freed after the variable calculation. In many cases, the data variables in a single state are generated in different clock cycles. It is likely that the calculation of next functional unit requires the use of an occupied register from the current state. Thus, as soon as a variable is generated, it should be transferred to the temporary register files.

The scheduling from register files to NVM follows the timing order of states generation. It is simply because each NVM space is expected to store a single complete state. Assume that variable v_1 belongs to state s_1, variable v_2 belongs to state s_2, and s_1 is prior to s_2. Both v_1 and v_2 are in the register file, waiting to be written to NVM. Note that we assume there exists only a single port between the register file and the NVM cache, hence, we can only choose to write either v_1 or v_2. Although it is possible that v_2 is generated before v_1, it is meaningless to first write v_2 to NVM. Because if so, when a fault occurs shortly after s_2, neither s_1 nor s_2 is available in the NVM as a complete state for recovery. Nonetheless, if v_1 is stored to NVM as a priority, there is a higher chance that by the time when a fault occurs, s_1 has been ready for recovery as a checkpoint.

5. EXPERIMENTS

5.1 Energy Model

We build energy models for the program execution on datapath ($E_{original}$) as well as the energy required to write the state (E_{state}).

$E_{original}$ consists two parts of energy consumption, energy for arithmetic units, and energy for register read and write. To evaluate $E_{original}$, we have applied the energy model from circuit simulator Hspice. The main advantages of the circuit-level simulation are its accuracy and generality. It estimates the energy consumption of our program execution on datapaths regardless of technology, design, style, functionality, and architecture.

E_{state} also involves two components, the energy to write to temporary register files (E_{reg}) and the energy to write to NVM (E_{nvm}). E_{reg} can be easily modeled using circuit simulator. In terms of E_{nvm}, the type of NVM we have applied in our model is STT-RAM. Its energy and latency data used in our simulation is obtained from CACTI [19] and Chang's paper [20]. Table 2 shows the latency and energy parameters for STT-RAM. The high-capacity cache is a 32nm, 32MB, 16-way cache that is partitioned into 16 banks and uses 64-byte blocks. Given the same size of data, E_{nvm} is orders of magnitude larger than E_{reg}, making E_{nvm} the dominant energy in E_{state}.

Cache	Area (mm^2)	Latency (ns)	Energy ($nJ/access$)
STT-RAM(32MB)	16.39	read:3.06 write:25.45	read:0.94 write:20.25

Table 2: Parameters of STT-RAM (32nm).

5.2 Simulation Results

We apply our FT mechanism on a set of controller benchmarks as shown in Table 3. The first three examples are controllers for Aircraft Advanced Flight Control (VAAC) aircraft. The plane is a British VSTOL air force aircraft with vertical take-off and landing capabilities. Each of the examples corresponds to three different phases and three different speeds of 6, 86, and 122 knots respectively. We see that the effectiveness of the approach increases in the more demanding situation where faster reaction on the measurement of control signals is required. Image and video are small processing systems that work on their corresponding streams by applying a variety of nonlinear filters and transforms for noise removal and contrast enhancement. Chemical.s and chemical.l are small and larger controllers used by a chemical plant. Honda and honda.lp are the high and low speed of an automotive controller for adjusting gasoline and oxygen into the engine. Steam is a small linear controller of a steam plant engine. Finally, gps.nav is a location and tracking system used for GPS-driven navigation. In our simulation, we assume that faults can happen in any arithmetic unit of the design.

In Table 3, column 2, 4, and 5 display the information of the program including the total number of generated variables, the number of multiplications, and the number of additions. Column 3 shows the number of variables in the selected checkpoint state. The size of the variables in each program equals to 16-bit. Column 6 and 7 respectively show $E_{original}$ for the original program, and E_{state} for the selected program state. The last column calculates the energy overhead of E_{state} compared to $E_{original}$. From the results, we have the following observations.

First of all, the percentage of energy overhead has no direct correlation with the size of the program or the size of the state. For example, chemical.s and video are two programs with the largest percentage of overhead. However, chemical.s has only 302 total variables, and video benchmark has as large as 20928 variables. A further observation suggests that the ratio between the number of state variables and the number of total variables has a dominant effect on the percentage of energy overhead. A larger ratio usually suggests a larger percentage. It is because that as the number of overall variables in the program increases, $E_{original}$ grows since more variables suggest more arithmetic units and more write operations to the regular registers. On the other hand, as the number of state variables decreases, the energy required to write data to NVM also decreases dramatically. To conclude, the most favored programs to our FT strategy are the ones with a large number of variables while the size of min-cut on the program is small.

To verify our observation, we choose four benchmark programs from MediaBench to apply our FT scheme. The four programs are intentionally chosen in such a way that they require long execution cycles while the selected state is relatively "small". Therefore, all four programs employ high ratio between the total number of variables and the state variables. As shown in Table 4, the average energy overhead is reduced to only 14.9%.

Secondly, the number of multiplications in the program is also important. Multiplication is among the most expensive operations in a program; an N-bit multiplier consumes around one magnitude more energy compared to an N-bit adder. Thus, if multiplications take a large percentage of the total arithmetic operations, $E_{original}$ will be increased a lot assume the total number of operations is fixed. However, this has no influence on the result of E_{state}, hence the ratio between E_{state} and $E_{original}$ will decrease.

Finally, as shown in the last row of Table 3, without any intentional design selection, the average energy overhead of our FT mechanism is around 41.5%. Although our algorithm targets to reduce the energy overhead, it is still not a negligible ratio for IoT applications. For those IoT devices that are extremely sensitive to energy overhead, our mechanism is not the best option. Alternatively, our FT strategy works best for the IoT systems that require real-time updates, high accuracy calculation, as well as moderate energy restriction.

6. CONCLUSION

We have proposed an NVM based fault tolerance mechanism for IoT applications. By storing intermediate states of a program to NVM, the program can be recovered from the states when faults occur, e.g., the lost of external power. By applying our modified max-flow algorithm on the data flow graph, we optimize the size of states in such a way that the energy overhead to store the state data to NVM is minimized. Moreover, we apply the temporary register files to hold state data before storing them to NVM, reducing the timing overhead to be almost negligible. Our test results on a variety of controller benchmarks indicate that the average energy overhead introduced by our FT scheme is 41.5%. Moreover, when intentionally choose programs that favor the apply of our FT mechanism, the average overhead can be further reduced to 14.9%.

7. ACKNOWLEDGMENT

This work was supported in part by the NSF under Award CNS-0958369, and Award CNS-1059435.

8. REFERENCES

[1] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 6, pp. 1524–1537, 2015.

[2] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for STT-RAM using early write termination," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, pp. 264–268, ACM, 2009.

[3] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *ACM SIGARCH computer architecture news*, vol. 37, pp. 34–45, ACM, 2009.

[4] E. F. Moore and C. E. Shannon, "Reliable circuits using less reliable relays," *Journal of the Franklin Institute*, vol. 262, no. 3, pp. 191–208, 1956.

Design	Number of overall variables	Number of state variables	Number of multiplications	Number of additions	$E_{original}$ (nJ)	E_{state} (nJ)	Percentage of Energy overhead
VAAC.6	223	15	118	90	0.883	0.480	54.4%
VAAC.86	431	17	214	200	1.66	0.544	32.7%
VAAC.122	482	19	241	221	1.87	0.608	32.6%
chemical.s	302	24	152	126	1.17	0.768	65.8%
chemical.l	870	36	438	396	3.38	1.15	34.1%
image	2323	64	966	1303	8.36	2.05	24.5%
steam	112	7	56	49	0.433	0.224	51.8%
honda	361	9	192	160	1.44	0.288	19.9%
honda.lp	376	8	0	368	0.822	0.256	31.1%
sound	269	19	126	124	1.01	0.608	60.3%
video	20928	1536	9728	9664	78.2	49.2	62.9%
gps.nav	489	16	226	247	1.83	0.512	27.9%
						Avg.	41.5%

Table 3: Application of the fault tolerance to a set of controller benchmarks for the estimation of energy overhead.

Design	Number of overall variables	Number of state variables	Number of multiplications	Number of additions	$E_{original}$ (nJ)	E_{state} (nJ)	Percentage of Energy overhead
D/A converter	426	4	222	200	1.69	0.128	7.6%
PGP	1940	34	468	1438	5.84	1.09	18.6%
JPEG.enc	1026	18	504	504	3.96	0.576	14.5%
MPEG2.dec	864	25	527	312	3.68	0.8	21.8%
GSM.enc.dec	1400	22	852	526	5.96	0.704	11.8%
						Avg.	14.9%

Table 4: Application of the fault tolerance to a set of benchmarks from MediaBench for the estimation of energy overhead.

[5] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluatuion*. Digital Press, 2014.

[6] W. H. Pierce, *Failure-tolerant computer design*. Academic Press, 2014.

[7] T. Karnik and P. Hazucha, "Characterization of soft errors caused by single event upsets in CMOS processes," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 2, pp. 128–143, 2004.

[8] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 29–40, IEEE, 2013.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[10] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, pp. 439–449, IEEE Press, 1981.

[11] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[12] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, p. 6, 2014.

[13] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pp. 123–134, IEEE, 2002.

[14] D. Kirovski, M. Potkonjak, and L. M. Guerra, "Cut-based functional debugging for programmable systems-on-chip," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 1, pp. 40–51, 2000.

[15] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network flows," tech. rep., DTIC Document, 1988.

[16] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[17] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1-6, pp. 5–35, 1991.

[18] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *Computers, IEEE Transactions on*, vol. 40, no. 2, pp. 178–195, 1991.

[19] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 3–14, IEEE Computer Society, 2007.

[20] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology comparison for large last-level caches (L 3 Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 143–154, IEEE, 2013.