Intermittent Computing

Bilkish Ara Naikodi, Phalguni Bhangod, Siddharth Gupta
University of Southern California, Ming Hsieh Department of Electrical Engineering
naikodi@usc.com, bhangod@usc.com, gupt232@usc.com
GitHub Repo Link: https://github.com/naikodi/Project599

Abstract*— *Intermittent Computing is applicable to devices that harvest energy from their surroundings when it is available, and store these bursts of energy for computations, thereby eliminating the need for a power source. The energy is not always available continuously and is therefore termed as intermittent. Nowadays, many applications such as medical sensors, small satellites etc., are expected to perform computations with small amounts of power and time. Therefore, energy distribution becomes critical and we allot this limited energy and time to more frequently occurring and meaningful computations. These devices comprise of hardware elements such as a CPU, sensors, transceivers, volatile memory, in which data is lost on power exhaustion and non-volatile memory, in which data is retained even on power exhaustion. To cater to this time constrained operation, we are replacing the conventional main memory with a memristor based memory device, which has access times that are approximately 100 times faster. These energy-starved devices are also more acceptable of approximate results for the mathematical computations. In this project we aim to generate approximate, yet acceptable results in a shorter duration of time and for smaller amounts of energy, for which we propose an architecture that is like What's Next Intermittent Computing Architecture paper but improving upon it by choosing a Memristor device as opposed to a conventional main memory, by designing another approximate multiplier design.

Keywords*-energy harvesting, intermittent computing, approximate computing, memristor****;***

I. INTRODUCTION

In recent years, small and low-power computing devices are seeing increased shifts in technology trends, towards numerous application domains such as medical sensors, implantable devices, satellites and computer vision. These devices operate using energy exclusively from the environment. These battery-less devices are powered entirely by energy gathered from environmental sources such as radio waves, solar light or vibration.

These devices present a unique challenge in terms of energy consumption as energy is not continuously available, but it is available intermittently in bursts, and all the meaningful computations must be done in the time during which the energy is available for. Such devices are also more accepting of approximate results and are error tolerant.

After referring to several papers, significant ones among those being the following,

1. Intermittent Computing - Challenges and Opportunities
2. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices
3. The What's Next Intermittent Computing Architecture,

We plan to implement "What's Next Intermittent Computing Architecture" paper and we propose two improvements over this architecture which are:

1. Replacement of a conventional multiplier with an approximate multiplier design and measure the accuracy rates for multiplication results and,
2. Replacement of conventional memory with a non-volatile memristor based ReRAM.

II. MOTIVATION

The motivation behind the above two changes is:

1. Multiplication was chosen as the operation to be approximated because of its high frequency of occurrence and high time latency.
2. Memristor was chosen over conventional main memory because of its lower area and power consumption, and higher speed of read/write access.

III. BACKGROUND

Energy harvesting devices are an emerging category of embedded systems that operate without the requirement for a battery, by harvesting energy directly off of the environment from sources such as vibrations, solar energy, WiFi etc. These past few years have seen a shift in technology trends towards increasingly small and low-energy computing intermittent devices, across a variety of application categories such as IoT, small satellites, implantable and wearable devices etc. Useful work can be performed in such applications only during the time when energy is available, and this availability pattern is intermittent. Hence the name intermittent devices. The hardware of such devices may contain a CPU, a Microcontroller, sensors and communication elements, volatile and non-volatile memory.

IV. CHALLENGES

The main challenge that these devices face is the shortage of energy availability and intermittent availability, which causes incomplete execution of the operation being performed. This successive combination of periods of activity and inactivity makes a system's control flow unpredictable and hampers a system's forward progress. It may also cause memory to become inconsistent and the device could also become with its surroundings and thereby impact device-to-device communication. All these issues need to be catered to

and, in this paper, we propose some of the smart methods that can be employed to make the computations performed during energy availability more meaningful and increase the number of computations performed.

A. Mathematical Challenges:
The challenges are to determine the advantage in terms of time and power, over existing Intermittent Computing technology by replacing the standard main memory with a memristor and exploring new approximation techniques mainly for multiplication. Multiplication is a basic operation executed maximum number of times in a piece of code in several applications and is time consuming. Hence, we have chosen to approximate multiplication operation.

B. Algorithmic Challenges:
1. Design of flow of Intermittent computing processor stages including the instruction fetch, decode, execute, memory and write-back stages and checkpointing in Python.
2. Conversion of high-level source code in Python to Assembly level Language with replacement of the conventional multiplication operation with operations that perform approximate multiplication (replacement of MULT operand with SHIFT and ADD operands).

C. Software Challenges:
Implementation of flow of Intermittent Computing architectural model in Python, using access times/delays calculated and obtained from NVSim and Cacti, for Memristor based memories.

V. PRIOR WORK

The What's Next Intermittent Computing Architecture paper is one of the few prior implementations of Intermittent Computing. The goal of this paper was to provide a partial answer when energy is scarce but offer the flexibility to refine that answer or computation result if more energy is available. The fundamental concept employed is while working on a particular input A (with power availability), if there is a power outage, an acceptable approximate result would be available for A and the processing of the next input could begin. This helps achieve greater forward progress across all input samples. The output metric in this architecture is best-effort.
This trade-off between energy and approximate computation is an amenable one because many energy-harvesting devices are prone and accepting of approximate computations.

VI. KEY NOVELTIES

1. A novel approximate multiplier was designed that saves computation time and energy by preprocessing one of the operands before multiplication; performed as a series of SHIFT and ADD operations.

2. ReRAM based non-volatile Main Memory (RAM) as opposed to conventional SRAM based volatile Main Memory and the effect of this replacement on number of computations performed (instructions executed), energy and area consumed.

VI. OUR ARCHITECTURE BASED ON WHAT'S NEXT INTERMITTENT COMPUTING ARCHITECTURE & OUR RESULTS

We have worked upon and implemented some of the features differently from the What's Next Intermittent Computing Architecture paper, which are:

1. We have implemented the project flow as described in the paper as a model in Python, with the exception of using a ReRAM based non-volatile Main Memory and obtaining the access times and delays for this replacement, calculated from NVSim and Cacti, and using these values in the Python Model, alongside a Normal Multiplier and simulated them as shown in Fig. 1.1.

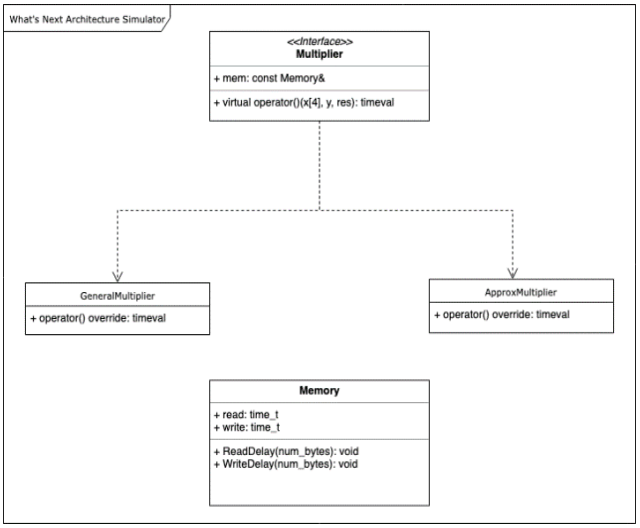

Fig. 1.1 Simulator UML Flowchart

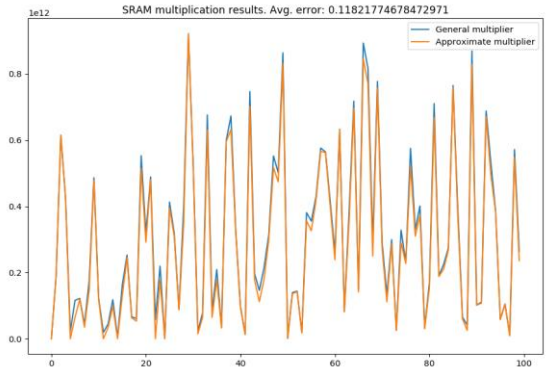

Fig. 1.2
Approximate Multiplier V/s General multiplier as defined in the above diagram.

2. The nvsim.cfg file was used to change the cache size and the .cell type being used with reference to SRAM.cell and ReRAM.cell and running the configuration file generated the area and power consumed by the respective memory models for the Main Memory(RAM).

Standalone Model	Read Latency (ns)	Write Latency (ns)
Main Mem(C)	288.64	152.318
Memristor	4.654	30.048

Table 1.1

Standalone Model	Read Dynamic Energy (nJ)	Write Dynamic Energy (nJ)	Static/Leakage Power (W)
Main Mem(C)	4.541	4.153	83.415
Memristor	4.81	0.967	8.944

Table 1.2

Standalone Model	Area (mm2)
Main Mem(C)	1334.447
Memristor	87.977

Table 1.3

From Tables 1.1,1.2 and 1.3 we can see that our Memristor model has a smaller read and write latency, smaller Write Dynamic Energy, Static Power Leakage and Area respectively when compared to the conventional Main memory(C) (SRAM) with the exception of read dynamic energy.

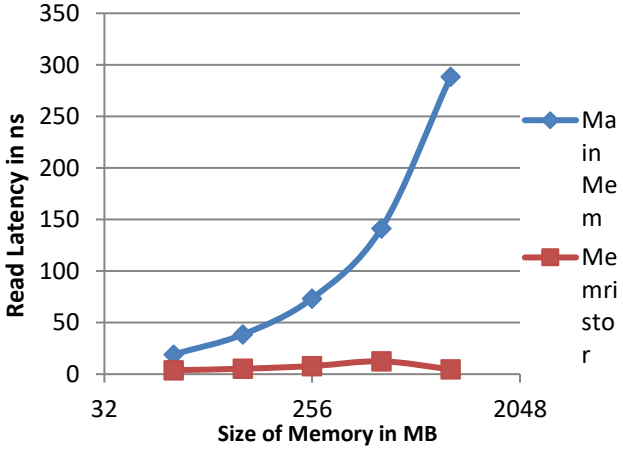

Fig. 1.3

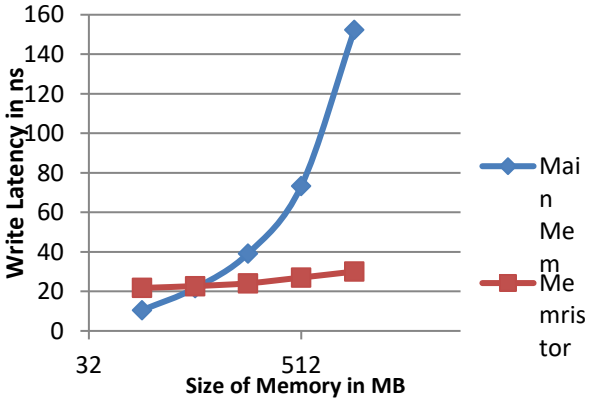

Fig 1.4

From Fig. 1.3 and Fig. 1.4 depict increase in Read and Write Latency respectively with size of memory for SRAM v/s ReRAM. It can be seen that latencies for the SRAM model increase marginally as compared to ReRAM model.

3. While the paper has used the MSB bits in a given word to simplify the computation complexity by assuming the MSB bits to be more significant in the multiplication operation, we have an alternate approximate multiplier that does a more meaningful approximation rather than always fixing the MSB bits to be chosen for approximate multiplication by assigning a priority. In our architecture, we perform preprocessing of one of the operand bits to minimize computation time and before the multiplication actually takes place as a series of SHIFT and ADD operations. The preprocessing involves calculating and storing shift amounts beforehand to avoid iterating through all the bits in the operand chosen and thus reduces the time needed for computation. While this technique presents a small area overhead because of the presence of a shift register, the approximation can further be improved by reducing the size of the shift.
(The Verilog code files are attached or can be viewed in GitHub).

4. To test the accuracy of the approximate multiplier, we have verified the multiplier results for various input combinations specified by the testbench code in Verilog and checked the waveforms for the output on ModelSim
The pseudocode of the multiplier is as follows:

```
A[1] = A;
  for(i=1; i <= n ; i++)
  {
        m = find_leading_one_bit_index(A[i]);
        //check the (n-i+1) bits starting from bit m
        If(A[i][m:m-(n-i)] == {(n-i+1){1'b1}})
        {
              S[i] = m+1; A[i+1] = 0;
        }
        else
        {     S[i] = m;
        //clear current leading one bit
        A[i+1] = A[i] & (~(1 << S[i] ));
        }
  }
```

5. We have designed a 3-stage pipeline architecture in Python, which is inherent in the ARM M0 processors, by coding the Instruction Fetch, Instruction Decode and Execute and Write-Back Stages. We have used the benchmark code for MatMul (Matrix Multiplication) written in C code and converted the benchmark code to assembly level and we have supported all the instructions present in the converted ISA in our Python code to test the performance.
(The Python Code is attached or can be viewed in GitHub).

6. We are however, not testing for the improvement in read/write latency for the Memristor replacement over the SRAM, because we are implementing our pipeline design in Python on an ARM M0 processor whose clock cycle time is around 40 ns, which is way larger when compared with the access times obtained for either the ReRAM or the SRAM. We are instead checking for the number of computations performed.
Combined results for both the novel approximate multiplier and ReRAM replacement over SRAM based main memory:

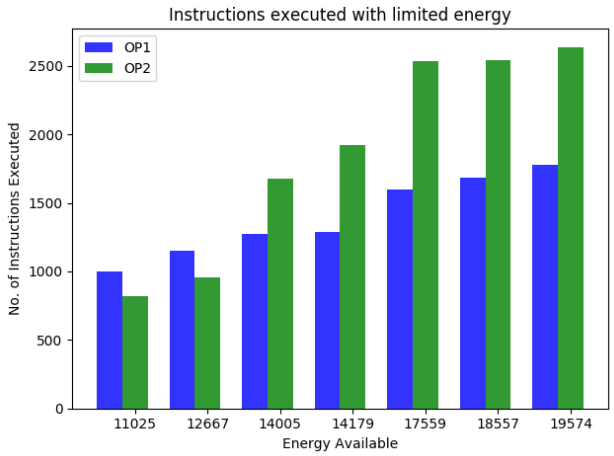

Fig 1.5

From Fig. 1.5 depicts the number of instructions executed by the What's Next Intermittent Computing Architecture (the Blue portion of the Graph or OP1) V/s our novel Intermittent computing model (the Green portion of the Graph or OP2), for the same amount of energy supplied.

Our model fares better for larger amounts of energy values when compared to the What's Next architecture, and this is determined by setting the same energy values for both the models by setting the total available energy value as a counter and decrementing it until it becomes zero (energy is exhausted) and by keeping a count of number of instructions executed before the energy is nullified.

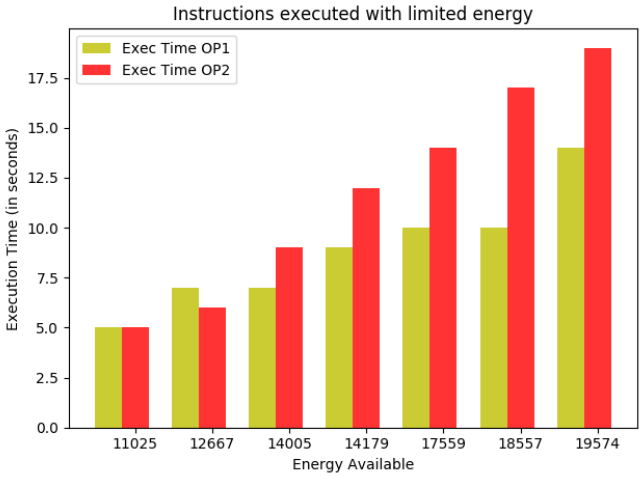

Fig. 1.6

Fig. 1.6 depicts the execution time to execute instructions of count and energy values calculated from Fig. 1.5 for the What's Next Intermittent Computing Architecture (the Yellow portion of the Graph or Exec Time OP1) V/s our novel Intermittent computing model (the Red portion of the Graph or Exec Time OP2).

The What's Next Model fares better than our architecture for higher values of energy. However, this is an acceptable trade-off for being able to execute a greater number of instructions with the same amount of energy for longer execution time periods.

VII. FUTURE SCOPE

Here are some of the improvements that can be done to our current architecture:

1. Implementation of Checkpointing: Checkpointing ensures forward progress on intermittently powered systems in the face of frequent power outages. By saving processor state periodically before power outage, after power availability the system can resume processing immediately without having to restore the state from main memory each time.
2. We can deploy machine learning algorithms to preprocess the data instead of having a fixed hardware scheme for preprocessing, which can have training and testing data sets and employ various algorithms on the data to determine the bit locations that are '1' in a word, or any other form of priority assignment required as per the application as appropriate.

VIII. PROJECT TIMELINE

Phase 1	1. Read papers related to Intermittent computing and got a basic understanding. 2. Learned using Cacti and NVSim for memory modelling .
Phase II	1. Simulated several memory models on NVSim and Cacti for RAM to determine the benefit of memristor replacement in terms of area, latency and power. 2. Developed an approximate multiplier model in Verilog and verified its accuracy in ModelSim, for different shift value parameters.
Phase III	1. Designed a 3-stage pipeline in Python to support the ISA of the benchmark MatMul by replacing MULT operand with a series of SHIFTS and ADDS, with ReRAM as Main memory and test for the performance of our architecture over the What's Next model and tabulate the results.