POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Computer Science and Engineering

Understanding and Testing Intermittence Bugs in Transiently-powered Computers

Supervisor:
Prof. Luca Mottola

Master Graduation Thesis by:
Andrea Maioli
Student ID n. 852429

Academic Year 2017-2018

Abstract

Using the energy harvested from the environment for powering small-scale devices reduces maintenance cost, and enables applications and services considered to be impractical due to battery limitations. Devices powered only with such energy source compute *intermittently*, as energy is available. Their execution is characterized by intervals of active computation, interleaved by periods in which the device is powered off and recharges its energy buffer.

Despite different techniques enable forward progress of programs, an intermittent execution inevitably causes the re-execution of some portion of code. Such re-executions may cause unwanted behaviors, such as the computation of incorrect results or unexpected environment interactions.

The current literature recognizes such unwanted behaviors as generic *intermittence bugs*, and does not provide an in-depth analysis of them. Moreover, no available technique enables testing the effects of all the possible combinations of intermittent executions in practical time. This operation has a complexity which grows exponentially with respect to all the possible interruption points, and can result in years of processing time for analyzing even a simple program.

In this thesis we provide an exhaustive analysis of *intermittence bugs*, including an analysis of their causes, guidelines on how to avoid such unwanted behaviors, and a set of techniques that enable their analysis in practical time. We also analyze the possibility of exploiting *intermittence bugs* for making programs aware of intermittence, and we provide both a technique and a set of guidelines that permit testing the correctness of this new kind of input.

Our contribution includes `ScEpTIC`, an offline tool that implements all the testing techniques that we provide in this thesis. `ScEpTIC` helps the programmer analyze and test the effects of intermittent executions in practical time. Not only it provides a speedup of six orders of magnitude with respect to brute-force approaches, but it also returns a more complete result that correctly identifies the exact position of *intermittence bugs*.

An excerpt of this thesis work is currently submitted for conference publication.

Sommario

Alimentare i dispositivi di dimensioni ridotte usando solo l'energia fornita dall'ambiente riduce i costi di manutenzione e permette la creazione di applicazioni e servizi considerati poco pratici a causa delle limitazioni delle batterie. I dispositivi alimentati con tale sorgente di energia computano solo quando l'energia è disponibile, e sono caratterizzati da un'esecuzione intermittente, che separa intervalli di computazione attiva ad intervalli in cui il dispositivo è spento.

Nonostante differenti tecniche permottono di riprendere l'esecuzione da dove è stata interrotta, la ri-esecuzione di alcuni porzioni di codice è inevitabile. Queste ri-esecuzioni possono causare comportamenti inaspettati, come ad esempio il calcolo di risultati non corretti o interazioni non previste con l'ambiente circostante.

La letteratura riconosce questi comportamenti inaspettati come dei generici *bug da intermittenza*, e non fornisce una loro analisi approfondita. Inoltre, nessuna tecnica al momento disponibile permette di analizzare in tempo ragionevole gli effetti di tutte le possibili combinazioni di esecuzioni intermittenti. Questa operazione ha una complessità che cresce esponenzialmente rispetto a tutti i possibili punti in cui la mancanza di energia può causare uno spegnimento del dispositivo, e si potrebbe impiegare anni di computazione per analizzare anche solo dei piccoli programmi.

In questa tesi forniamo un'analisi esaustiva dei *bug da intermittenza*, che include l'analisi delle loro cause, linee guida per evitare tali bug, ed un insieme di tecniche che permottono la loro analisi in tempi pratici. Analizziamo inoltre l'utilizzo di alcuni *bug da intermittenza* per rendere il programma consapevole dell'intermittenza stessa, e forniamo sia delle tecniche che permottono di verificare la correttezza di questo nuovo tipo di input, che delle line guida per il suo utilizzo.

Il nostro contributo comprende anche `ScEpTIC`, un tool di nostra creazione che implementa le tecniche di analisi fornite in questa tesi. `ScEpTIC` aiuta il programmatore ad analizzare e verificare gli effetti che tutte le possibili esecuzioni intermittenti hanno sul programma. Nei nostri test di laboratorio, `ScEpTIC` ha dimostrato una performance di sei ordini di grandezza superiore rispetto a comuni tecniche di *brute-force* per analizzare tutti i possibili punti di interruzione, e produce un risultato più completo ed accurato rispetto a tali tecniche.

To my grandparents.

Acknowledgements

This journey made me understand what scientific research really means.
I learned a lot about myself and about facing problems from different perspectives. I am truly grateful to all the people that made this journey possible and supported me along this path full of ups and downs.

First of all I want to thank my advisor, Prof. Luca Mottola, for challenging me with an interesting problem and for all the support he gave me. Working with him was inspiring and without his help this thesis would not have been possible.

Then, I want to thank my parents and my better half Federica, who always supported me and encouraged me to pursue my goals. Their help was essential, and I am very grateful for their presence in my life.

Finally, I want to thank all my friends and lab colleagues that in some way or another shared this path with me: Alessandra, Cameron, Davide, Emanuele, Enrica, Francesco B., Francesco Ce., Francesco Ch., Lorenzo, Luca, Marco, Matteo, Michela, Mirko, Pietro, and Schön.

Contents

1 Introduction **1**
- 1.1 Problem . . . 2
- 1.2 State of the Art . . . 8
- 1.3 Contribution . . . 10
  - 1.3.1 Classification and Analysis of Inconsistencies . . . 11
  - 1.3.2 Inconsistency-oriented Debugging Tool . . . 16
- 1.4 Thesis Structure . . . 19

2 Transiently Powered Computing **23**
- 2.1 Saving Work Done . . . 24
  - 2.1.1 Overview . . . 24
  - 2.1.2 Checkpoint-based Solutions . . . 24
  - 2.1.3 Task-based Solutions . . . 25
- 2.2 Checkpoints and Inconsistencies . . . 26
- 2.3 Checkpoint Mechanisms . . . 29
  - 2.3.1 MementOS . . . 29
  - 2.3.2 DINO . . . 30
  - 2.3.3 Ratchet . . . 32
  - 2.3.4 Hibernus / Hibernus++ . . . 34
  - 2.3.5 QuickRecall . . . 35

3 The Problem of Testing **37**
- 3.1 Overview . . . 37
- 3.2 Available Tools . . . 43
  - 3.2.1 EKHO . . . 44
  - 3.2.2 EDB . . . 44
  - 3.2.3 Siren . . . 45
  - 3.2.4 CleanCut . . . 46
- 3.3 Testing Environment Requirements . . . 48

4 Memory-section Specific Inconsistencies **51**
- 4.1 Overview . . . 51
- 4.2 Data Accesses . . . 52

4.2.1 NVM and Memory Sections . . . 52
4.2.2 Data Access . . . 53
4.2.3 Data Access Inconsistencies . . . 54
4.3 Stack . . . 55
4.3.1 Stack, Activation Record and Function Calls . . . 55
4.3.2 Function Calls and NVM . . . 57
4.3.3 Activation Record Inconsistency . . . 60
4.4 Heap . . . 62
4.4.1 Heap Structure and Management . . . 62
4.4.2 Data Accesses . . . 64
4.4.3 Memory Maps . . . 66
4.4.4 Inconsistencies for Heap in NVM . . . 69
4.5 Analyzing Memory Inconsistencies . . . 70
4.5.1 Sequential-equivalence Algorithm . . . 70
4.5.2 Data Access and Activation Record . . . 75
4.5.3 Memory Map Inconsistencies . . . 80
4.6 Finding Inconsistencies . . . 84

5 Environment Interactions and Inconsistencies 91
5.1 Overview . . . 91
5.2 Environment Inputs . . . 92
5.2.1 Input Accesses . . . 92
5.2.2 Testing Input Access Models . . . 93
5.3 Intermittence as Program Input . . . 97
5.3.1 Inconsistencies and Inputs . . . 97
5.3.2 Testing Intermittence-based Inputs . . . 100
5.4 Environment Outputs . . . 104
5.4.1 Ouput Inconsistencies . . . 104
5.4.2 Analyzing Output Inconsistencies . . . 105

6 ScEpTIC: Testing Intermittent Computation 111
6.1 Overview . . . 111
6.2 LLVM IR . . . 112
6.2.1 LLVM and Intermediate Representation . . . 112
6.2.2 IR File Structure . . . 113
6.2.3 LLVM IR Labels and Basic Blocks . . . 116
6.2.4 LLVM IR Instructions . . . 116
6.3 Code Representation . . . 119
6.3.1 ScEpTIC Intermediate Representation . . . 119
6.3.2 Libraries and `ScEpTIC` Built-ins . . . 122
6.3.3 Register Allocation and Tests . . . 126
6.4 Architecture . . . 132
6.4.1 Overview . . . 132
6.4.2 Register File . . . 133

6.4.3 Memory . . . 134
6.4.4 Input and Output . . . 137
6.4.5 Checkpoint Manager . . . 140
6.4.6 Interruption Manager . . . 140
6.5 ScEpTIC AST Execution . . . 142
6.6 Configuration . . . 143
6.6.1 Overview . . . 143
6.6.2 Test Configuration . . . 143
6.6.3 Register File Configuration . . . 145
6.6.4 Memory Configuration . . . 145
6.6.5 Checkpoint Configuration . . . 147
6.6.6 Pre-defined Environment Configurations . . . 148

7 Testing Mechanisms Implementation 151
7.1 Data Interruption Manager . . . 151
7.1.1 Lookup and Memory Map Tables . . . 151
7.1.2 Activation Record Checks . . . 156
7.1.3 Checkpoints Coverage . . . 159
7.1.4 Test Execution Flow . . . 160
7.1.5 Analysis Output . . . 164
7.2 Input Interruption Manager . . . 168
7.2.1 Overview . . . 168
7.2.2 Input Dependency Table . . . 169
7.2.3 Test Implementation . . . 173
7.2.4 Analysis Output . . . 173
7.3 Interaction Interruption Manager . . . 176
7.3.1 Overview . . . 176
7.3.2 Profiling Start, Reset, and Log . . . 177
7.3.3 Test Implementation . . . 179
7.3.4 Test Output . . . 182
7.3.5 Testing Output Inconsistencies . . . 184

8 Evaluation 191
8.1 Overview . . . 191
8.2 Memory Inconsistencies . . . 194
8.2.1 Baselines . . . 194
8.2.2 Evaluation Inputs . . . 199
8.2.3 Execution Depth Estimation . . . 200
8.2.4 Input Source Files . . . 204
8.2.5 Quantitative Evaluation Results . . . 214
8.2.6 Qualitative Evaluation . . . 252
8.3 Input Inconsistencies . . . 254
8.3.1 Evaluation Baseline . . . 254
8.3.2 Input Source File . . . 258

8.3.3 Quantitative Evaluation Results 260
8.3.4 Qualitative Evaluation 267
8.4 Intermittent Execution Analysis 269
8.4.1 Evaluation Setup 269
8.4.2 Qualitative Evaluation 272

9 Conclusion and Future Works **277**

List of Figures

1.1 Inconsistency example described by Lucia et al. [1]. 3

1.2 Example of an intermittent execution which leads to an unexpected state of the environment. 7

1.3 Example of an intermittent execution leading to a state which is not synchronized with respect to the environment. 8

1.4 Example of an *Activation Record Inconsistency*. The stack is allocated into NVM, and thus every write is persistent across power resets. Once the function f_1 returns, it pops the stack content. When the function f_2 is called, it pushes onto the stack the return address, which goes to the same position of where the return address of f_1 was stored. As consequence, when the checkpoint is restored due to a power reset, the function f_1 will pop the wrong return address from the stack, since it was overwritten by $f_2()$ call in the previous power cycle. For this reason, the program counter is set to the address of line 8 and not to the one of line 7, entirely skipping the call of f_2. 11

1.5 Example of an intermittent execution leading to an unexpected behavior due to the allocation of the heap segment in NVM. The variable p points to a cell which was dynamically allocated into the heap. Once the cell is freed, it is removed from the heap. In such a scenario, if a shutdown happens, the code after the checkpoint is re-executed. The following instructions expect the cell pointed by p to be available, but it does not exist anymore. 12

1.6 Example of an intermittence-based input, which helps us in keeping track the number of times we re-execute a certain code region due to power resets. Moreover, from another point of view, it tells us the number of happened power resets. 15

2.1 Example of a device based on the *MSP430* architecture [2]. . 23

4.1 Stack content after running the function prologue in Example 4.3 . . . 57
4.2 Stack content after execution of *CHECKPOINT* in Example 4.4 . . . 58
4.3 Stack content after execution of line 37 in Example 4.4 . . . 59
4.4 Different kind of inconsistencies given the checkpoint placement. . . . 61
4.5 Partitioning of the main memory. . . . 63

6.1 Compiler pipeline. . . . 111
6.2 AST produced from Example 6.1. . . . 119
6.3 Source code parsing pipeline. . . . 120
6.4 `ScEpTIC` Parser Module. . . . 121
6.5 Sequence diagram representing the parsing of the source file. 121
6.6 `ScEpTIC` Abstract Syntax Tree. . . . 123
6.7 `ScEpTIC` Builtins Module. . . . 123
6.8 `ScEpTIC` Linear Scan Register Allocation Module. . . . 128
6.9 Sequence diagram representing the initialization of `ScEpTIC`. 132
6.10 Register File module of `ScEpTIC`. . . . 134
6.11 Memory module of `ScEpTIC`. . . . 134
6.12 I/O module of `ScEpTIC`. . . . 137
6.13 Sequence diagram representing the execution of the current instruction in the `ScEpTIC` AST. . . . 142

7.1 Flow chart representing how the lookup table helps in finding memory access inconsistencies over a memory cell. . . . 153
7.2 Flow chart representing how the lookup table helps in finding memory map inconsistencies over a memory cell. . . . 153
7.3 Sequence diagram representing the operations executed for performing a memory write operation. . . . 155
7.4 Sequence diagram representing the operations executed for performing a memory read operation. . . . 155
7.5 Example of a stack activation record. . . . 156
7.6 Content of the function call lookup in Figure 7.5. . . . 157
7.7 Flow chart representing the work flow for recognizing an activation record inconsistency of the *_stack_check_at_address()* method. . . . 158
7.8 Sequence diagram representing how `ScEpTIC` updates the *Input Lookup Data* during the execution of a *BinaryOperation*. 170
7.9 Flow chart representing the work flow of the *_check_input_lookup()* method. . . . 172
7.10 Sequence diagram representing the operations that `ScEpTIC` executes for updating the *input_table* and the *output_table*. . . . 177

7.11 Flow chart representing how the *run_with_intermittent_execution()* method executes a single instruction. 180

8.1 Graphs of the remaining time after a checkpoint is taken . . . 202
8.2 Graphs of the execution depth after a checkpoint is taken . . 202
8.3 Data evaluation results: Dynamic Analysis of CRC benchmark with global variables allocated in NVM. 215
8.4 Data evaluation results: Static Analysis of CRC benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM. 217
8.5 Data evaluation results: Dynamic Analysis of CRC benchmark with the stack allocated in NVM. 222
8.6 Data evaluation results: Dynamic Analysis of FFT benchmark with variables *realin*, *imagin*, *realout*, and *imagout* allocated in NVM. 224
8.7 Data evaluation results: Dynamic Analysis of FFT benchmark with global variables allocated in NVM. 226
8.8 Data evaluation results: Static Analysis of FFT benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and global variables allocated in NVM. . . . 227
8.9 Data evaluation results: Static Analysis of FFT benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM. 230
8.10 Data evaluation results: Dynamic Analysis of AES benchmark with global variables allocated in NVM. 232
8.11 Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and global variables allocated in NVM. . . . 234
8.12 Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *function-return* strategy of MementOS [3] and global variables allocated in NVM. 236
8.13 Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM. 237
8.14 Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *function-return* strategy of MementOS [3] and the stack allocated in NVM. . 241
8.15 Data evaluation results: Dynamic Analysis of Sense benchmark with global variables allocated in NVM. 244
8.16 Data evaluation results: Static Analysis of Sense benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and global variables allocated in NVM. . . . 246

8.17 Data evaluation results: Static Analysis of Sense benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM. 249
8.18 Input evaluation results: metrics of the input access analysis with one, two, three, and four different inputs. 261
8.19 Input evaluation results: inconsistencies found in the four configurations of our benchmark. Each configuration was run once for every possible combination of input access methods of its inputs. 265

List of Tables

4.1 Elements of the assembly language used in the examples. . . 51
4.2 Instructions of the assembly language we use in the examples of this chapter. 52
4.3 List of functions permitting heap management during runtime. 63
4.4 Status of the heap in Example 4.5 64
4.5 State of the heap after the execution of the *realloc* instruction of line 7 in Example 4.7 66
4.6 Heap status after the execution of *free* at line 5 of Example 4.9 67
4.7 Heap status after the execution of *realloc* at line 7 of Example 4.9 67
4.8 Heap status after the execution of the *malloc* instruction at line 7 of Example 4.10. 68
4.9 Lookup Table of Example 4.12, after the execution of the instruction at line 3. 75
4.10 Lookup Table of Example 4.12, after the execution of the instruction at line 5. 75
4.11 Lookup Table of Example 4.13, after the execution of line 13. 78
4.12 Lookup Table of Example 4.13, after the execution of line 18. 78
4.13 Status of the heap in Example 4.14. 81
4.14 Memory Map Table of Example 4.14, before the execution of the *realloc* instruction at line 5. 81
4.15 Memory Map Table of Example 4.14, after the execution of the *realloc* instruction at line 5. 81
4.16 Execution trace of Example 4.15, that permits us to identify if and where inconsistencies may happen. 86
4.17 Subset of checkpoint and reset locations. 87

5.1 List of functions permitting environment interactions. 91
5.2 Content of the input-dependency table 95
5.3 Content of Interaction Table of Example 5.15 once the execution completed. 101

6.1 Most used LLVM IR first-class types. 114

6.2 Relevant LLVM IR meatadata and their attributes. 115
6.3 Initialization of Algorithm 10 applied to Example 6.8. 130

8.1 Capacitance and Voltage before checkpoint pairs found by the calibration of Hibernus++ [4]. 201
8.2 Access ratio to the support memory of `ScEpTIC` and *Baseline*. 263

List of Algorithms

1 Static sequential-equivalence algorithm to be applied after we take a checkpoint. 72
2 Dynamic sequential-equivalence algorithm to be applied after we take a checkpoint. 73
3 Optimization of the dynamic sequential-equivalence algorithm to be applied after we take a checkpoint. 76
4 Final optimization of the dynamic sequential-equivalence algorithm to be applied after a checkpoint is taken. 83
5 Searching inconsistencies in a dynamic configuration 88
6 Searching inconsistencies in a static configuration 89
7 Sequential execution for testing input access models. 96
8 Intermittent execution algorithm for testing intermittence-based inputs. 103
9 Intermittent execution algorithm for testing effects of non-idempotent output routines. 107
10 Iterative calculation of the overall number of registers used by each function. 129
11 Alternative input access model analysis algorithm, used as baseline 257

Chapter 1

Introduction

Transiently Powered Computing (TPC) consists in using the energy harvested from the environment for powering small-scale devices. This technique is becoming more and more popular in the Internet-of-Things (IoT) area. Having a battery-less system leads to a dramatic reduction of maintenance costs and has the potential to enable applications and services considered to be impractical due to battery limitations [5, 6, 7, 8, 9].

The target devices used in this domain have small form factors, low power consumption, low computational power, and contained costs. They consist in a Micro Controller Unit (MCU) with a very limited amount of memory and storage, which are not sufficient to host an operative system. For this reason, the MCU executes a single program, which is uploaded by the user in form of a compiled binary. An example of a target device is the MSP430 [2], which is a low-power MCU with 512 KB of flash memory and 66 KB of RAM. This class of devices is not provided with a battery, and they store the energy inside a small capacitor.

The energy harvested from the environment has an unpredictable behavior, and devices that use it as main power supply are characterized by frequent shutdowns. When we use such energy source, we can observe an *intermittent execution*, characterized by intervals of active computation separated by intervals in which the device is completely powered off. The length and frequency of such intervals depend on the presence of energy that can be harvested from the environment.

Having frequent shutdowns creates the needs of saving all the information computed by the device into a non-volatile memory (NVM), so to not restart the entire computation over again. This allows the device to continue from the point in which it powered off. There are different techniques to preserve the information computed by the device between shutdowns, and most of them [1,3,4,10,11,12] are based on the concept of *checkpoints*. A checkpoint consists in saving the volatile state of the micro-controller unit into NVM and, depending on the device configuration, it can include the register file,

the stack, and/or the heap. Whenever the device wakes up, it verifies the presence of a valid checkpoint and, if available, the device restores the saved state contained in it.

1.1 Problem

Performing a checkpoint introduces an overhead, from both the energy and computation perspectives. To reduce such overhead, different checkpoint mechanisms [1, 10, 11, 12] allocate portions of the main memory into NVM, such as single variables or the entire stack. In this way, the amount of data saved by a checkpoint is reduced, since only the volatile state must be saved. Unfortunately, the allocation of a memory element into NVM does not come for free. In fact, such allocation introduces a new class of *bugs*, which is referred by different works as *intermittence bug* [13] or *state inconsistency* [1, 10, 14, 15, 16].

When a checkpoint is executed, it saves the volatile information of the state, comprehending the program counter, the content of the register file, and the content of the volatile memory. The NVM is a persistent memory space that does not lose data across power resets, and thus a checkpoint does not save its state. If the execution of a subsequent instruction modifies the NVM, the state that a previous checkpoint saved is no longer consistent. In fact, if a shutdown happens due to a low energy buffer, the content of the NVM is preserved, but it differs from when the checkpoint was taken. This discrepancy between the states at the checkpoint and during the power off causes the *state inconsistency* problem. When there is enough energy to restart the computation, the volatile state is restored from the information that a previous checkpoint saved, and the computation resumes from where the checkpoint was taken. In such a scenario, the content of the NVM contains results produced by *future* instructions with respect to where the execution is resumed. As consequence, the execution of subsequent instructions will use as input those results, producing incorrect values and leading to wrong results. From a *science-fiction* point of view, this bug can be described as a *broken time-machine* [14], which travels back in time while maintaining the effects of changes done in the future.

Data Processing. To better understand this problem, let us focus on Figure 1.1, which is the example used in DINO [1] to describe the *inconsistency* problem. It contains the portion of a program, in which the variables *len* and *buf* are allocated into NVM. The sequential execution of this code sets the variable *len* to *0*, and the array *buf* to '*a*' in the cell number *0*.

Let us now explore what happens during the intermittent execution of the same code. We start the execution, and then we reach the *checkpoint* routine present at line 1, which saves the volatile information such as the register file and the stack content. As next instruction, we run the operation

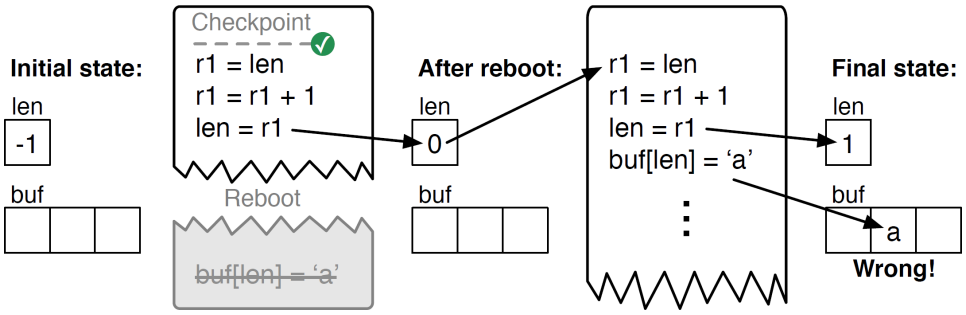

Figure 1.1: Inconsistency example described by Lucia et al. [1].

at line 2. It loads the value of the NVM variable *len*, which is *-1*, into the variable *r1*. Then, we execute the operation at line 3, which increment to *0* the value of variable *r1*. The execution continues with the instruction at line 4. It stores the value of variable *r1* into the NVM variable *len*, which now assumes a value of *0*. Let us now suppose that the energy source is not providing enough energy to our device, and that the energy buffer was completely emptied during the execution of the previous instructions. As consequence, a shutdown happens due to a low energy level.

While the device is powered off, the energy harvested from the environment refills the device energy buffer. When there is enough energy to restart the computation, the previously saved checkpoint is restored. The execution resumes from line 2, and it loads into the variable *r1* the content of the NVM variable *len*, which is now *0*. This loaded value was produced by the instruction at line 4 during the previous execution of the code. Since we restored the checkpoint at line 1, we should not be able to see such result, because it was produced by a future instruction. For this reason, from now on, all the computation will produce incorrect results. In fact, if we continue the execution of the code, *len* will be set to *1* instead of *0*, and the array *buf* will be set to '*a*' in the cell number *1* instead of *0*.

The described inconsistency problem creates the need of verifying the correctness of the *intermittent* executions of the program. To verify the correctness of a *sequential* execution of a program, the literature provides different types of code testing methodologies (e.g. unit testing), but unfortunately very little work exists for *intermittent* execution [13, 17, 18]. It is not possible to use existing *sequential* methodologies, since they are not conceived to account for power outages scenarios.

There are four elements that must be considered to run a test in an *intermittent* execution scenario:

- *Where to generate a power reset*: in an intermittent execution scenario it is not possible to predict where power resets may happen, due to the unstable and unpredictable nature of harvested energy sources. For this reason, a power reset can happen after the execution of any

instruction. For exhaustively testing the correctness of a program, we should verify the consequences of each possible power reset.

- *How to analyze power resets*: for verifying the cause of a power reset, when we generate it we must start from a consistent state. In fact, each power reset we generate may change the non-volatile state, potentially leading to an inconsistency. If we generate a new power reset starting from an inconsistent state, all the results that subsequent instructions produce may be incorrect. As consequence, we are not able to tell if the incorrect results are a consequence of the previous power reset, or if the new power reset causes new inconsistencies. To avoid this problem, we must provide a way to generate one independent test for each possible power reset. In this way, no previous power reset can interfere with the analysis of a subsequent one, and thus we are able to correctly identify the causes of an inconsistent state.

- *How to recognize inconsistencies*: for verifying if the state is consistent, we must define a technique for recognizing if the effects produced by each operation on the state are correct or not. This requires us to know the correct state produced by the execution of each instruction. In fact, if we do not have such information, we do not have any element to which compare the state obtained after a power reset, and thus we are not able to establish if it is consistent or not.

- *Which checkpoints to test*: depending on our requirements, we might want to verify the presence of inconsistencies considering a specific checkpoint placement, or considering all the possible checkpoint placements.
  - *Specific checkpoint placement*: in this case, we already chose where the checkpoints are placed in the code, by either placing them by hand or using a tool such as MementOS [3]. In such a scenario, we must run a group of tests which exhaustively verifies the presence of inconsistencies, by reproducing all the possible power resets. The inconsistencies found by this analysis are only valid with the given checkpoint placement, and in the case we want to change where checkpoints are, we must run another group of tests.
  - *All possible checkpoint placements*: in this case, we did not choose where the checkpoints are placed, and we want to evaluate the presence of inconsistencies without having to specify any placement. For exhaustively testing the presence of inconsistencies in such a scenario, we have to generate all the possible checkpoint placements, otherwise we can lose some relevant information about where inconsistencies may happen. Then, for any of the possible checkpoint placement, we have to run the same group

> of tests we described in the case above. Moreover, since we do not know a priori where the next checkpoint will be, we must account for the effects of all the instruction after the considered checkpoint. For this reason, we should consider checkpoints to be independent, and thus we must run one group of test for each checkpoint, as if no further checkpoints happen after the considered one. In this way, the inconsistencies found by this analysis covers all the possible checkpoint placements.

As consequence, running an exhaustive test in an intermittent execution scenario is not a trivial process, and the required computational effort is considerable. Let us consider the most simple approach we can think of, for finding the presence of inconsistencies, that is to analyze the effects that any possible power reset has over the device state. We describe such approach in Chapter 4, and we use it as baseline for our evaluation in Chapter 8. The resulting algorithm generates a power reset after every line of machine code, since a shutdown can happen at any instant during the execution of the program. For every checkpoint it encounters, the algorithm saves a snapshot of the state, and starts testing for intermittence bugs. For every instruction after the checkpoint, the algorithm executes it, generates a power reset, and restores the state. Then, it compares the resulting state with the saved snapshot. If a difference is found, it restores the saved snapshot and signals the found inconsistency, otherwise it simply continues the execution. As next step, it executes the instructions until it reaches the one after the previous power reset, and repeats this process over again.

From another point of view, we can say that the algorithm runs the program to be tested one time for every possible power reset, and does so for any checkpoint. The total number of instructions executed to run the entire test has a complexity of $O(n_{chk} \cdot n_{instr}^2)$, with n_{chk} equal to the number of checkpoints to be executed and n_{instr} equal to the total number of instructions executed in the equivalent sequential execution of the same code. If we set the algorithm to test any possible checkpoint placement, it will assume checkpoints to happen at any line of code, and thus it will execute the described workflow for any of them, leading to a complexity of $O(n_{instr}^3)$.

For identifying the presence of inconsistencies, we require comparing the states that each power resets produce with the one produced by the equivalent sequential execution of the same code. The effort that this operation requires is significant, since every time we generate a power reset, we must pause the execution for performing s_{dim} comparisons, with s_{dim} equal to the number of elements that the state contains. The overall number of comparisons has a complexity of $O(s_{dim} \cdot n_{reset})$, with n_{reset} equal to $n_{chk} \cdot n_{instr}$. If we set the algorithm to test any possible checkpoint placement, n_{reset} is equal to n_{instr}^2. Furthermore, when we find an inconsistency, we also need

to restore a snapshot to make the state consistant, otherwise we are not able to continue our analysis. The overhead that such operations introduce limits the overall speed that we can reach during the analysis. For these reasons, in our testing environment, our prototype was able to analyze up to $1.8 \cdot 10^4$ instructions per second despite running on modern hardware and using optimized language backends.

Let us now consider the most simple benchmark we use in Chapter 8 to evaluate our work, which is *CRC*. It is a C program with 70 lines of code, and has 300 machine-code instructions in the *.text* section of the compiled binary. Its sequential execution runs $5 \cdot 10^4$ machine-code instructions. Running a test which covers all possible power resets will result in running a number of instructions in the order of 10^{14}, and the test will take years to complete, considering that our setup is able to analyze up to $1.8 \cdot 10^4$ instructions per second. Moreover, using the formula we describe in Chapter 8, we can precisely calculate the number of instructions to be executed, which is $2.34 \cdot 10^{13}$. Considering that, the test would take 41 years to complete, which is an unreasonable amount of time to get a result.

Predicting and exploring all the possible combinations of intermittent execution is a non-trivial task and, according to Hester et al. [19], no available tool is able to do that.

Depending on how a program accesses the memory, state inconsistencies may happen due to the allocation of a portion of main memory into the NVM. The lack of a well-defined testing workflow for intermittent execution leads the programmer to selecting a checkpoint mechanism without taking into account which kind of problems are introduced with the associated memory configuration. As consequence, the program may present poor performance and poor energy optimization, due to checkpoint placements and consequently introduced overhead. Furthermore, depending on the checkpoint mechanism, might not be guaranteed the absence of inconsistencies, and thus the program will not only be inefficient, but it will also produce incorrect results.

Environment Interactions. A common application of TPC devices consist in using them as sensors of wireless networks. In such a scenario, devices will sense data from the environment, process the sensed values, and then store the results or send them to the main node of the system. The unpredictable behavior of the harvested energy does not only affect the data computed by the device, but also affects environment interactions. In fact, having frequent power resets results in the inevitable re-execution of some portion of code, as we shown in previous examples. This re-executions affects not only the data stored in our device, but also the way in which it interacts with the environment:

- *Output.* Let us consider Figure 1.2, which represents a portion of a program that moves a servo by 45°, and let us verify what happens

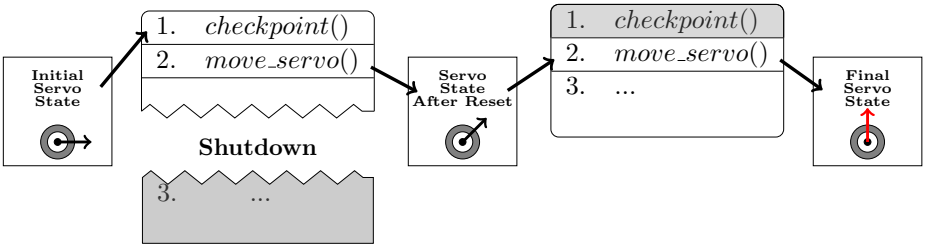

Figure 1.2: Example of an intermittent execution which leads to an unexpected state of the environment.

during the intermittent execution of the code. Let us suppose that the servo initial position is 0°. We execute the instruction at line 2, which moves the servo by 45°, and thus its new position is 45°. Let us now suppose that there is not enough energy to continue the execution, and thus a shutdown happens due to a low energy buffer. When there is enough energy to restart the computation, the latest saved checkpoint is restored and the execution continues from the instruction at line 2. It is the same operation we run before, and it moves the servo by an additional 45°, resulting to a new position of 90°. This state is different from the one we would obtain from a sequential execution, which would set the servo to 45°. Depending on how we wrote the code, we can consider the state to be inconsistent or not. If the following instructions assume that the servo is at 45°, all the subsequent results can be incorrect and the device can have an unexpected behavior.

- *Input.* Let us consider Figure 1.3, which represents a portion of a program that reads the light intensity of the environment, and let us verify what happens during the intermittent execution of the code. We execute the instruction at line 1, which stores the value of the light intensity into variable *a*. Then, we execute the checkpoint at line 2, and thus we save the state. Let us now suppose that there is not enough energy to continue the execution, and thus a shutdown happens due to a low energy buffer. Moreover, let us suppose that the device takes 2 hours to harvests enough energy to restart the computation, and that while the device is not running, the light intensity of the environment changes. When there is enough energy to restart the computation, the checkpoint is restored and the execution continues from the instruction at line 3. From now on, all the performed computation will as value of the light intensity the one stored in the variable *a*, which is 2 hours old, and thus it does not reflect the current environment state. The following instructions will produce a result which is computed using such old value, and this could lead to an unexpected program behavior.

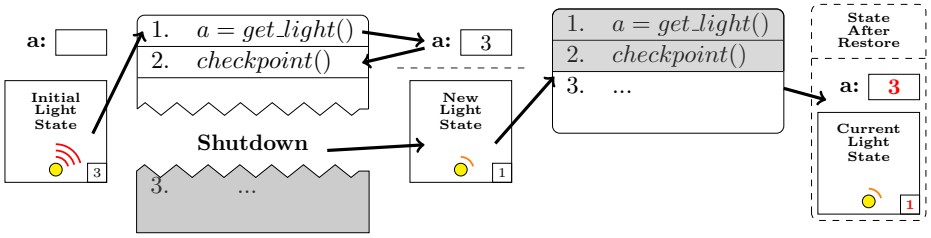

Figure 1.3: Example of an intermittent execution leading to a state which is not synchronized with respect to the environment.

All the previous works addressing inconsistencies [1,10,13,14,15,16] treat them only from a data standpoint, and they do not analyze environment interactions. As we showed in the previous examples, frequent power resets can also result in an unexpected behavior from an environment interaction standpoint. The lack of an analysis for this kind of inconsistency makes us unable to identify where in the code they can happen, under which circumstances they can occur, and how we can avoid such unwanted behaviors. Moreover, not recognizing this type of inconsistency leads us to not expect any unwanted behavior from an environment standpoint. As result, we do not test our program under the circumstances that produce such unwanted behavior, potentially leading us to treat an unreliable program as if it is reliable.

The absence of a general workflow for testing intermittent execution and the lack of a classification of inconsistencies leads to the impossibility of recognizing where hazards to consistency may happen inside the code. The main problem is thus providing both a tool and a general workflow for verifying, analyzing, and testing the results of an intermittent execution, independently of the checkpoint mechanism adopted, the micro-controller unit used, and the nature and patterns of harvested energy.

1.2 State of the Art

Currently four solutions have been proposed to deal with different facets of the testing problem in an intermittent execution environment: EDB [13], Ekho [20], Siren [17], and CleanCut [18].

- **EDB** [13] consists in a debugging environment that addresses the problem of not introducing any energy interference during the debugging of a device. For doing so, when a debugging operation is executed, EDB provides to the tested device a constant energy, so to compensate for the energy consumed by such debugging operation.

EDB is composed by both a hardware and a software solution. The first one consists in a debugging device that must be connected to the MCU of the device we want to test. This component permits us to physically modify the energy level available in the device, access the memory content, and read system's internal information. The second one consists in a software library which enables us to use the debugging capabilities of the hardware device. It permits us setting breakpoints based on the program execution or the energy level, and to verify conditions on the data.

EDB acknowledges the inconsistency problem we described in the previous section as *intermittence bug*. We may use its exposed functionalities for verifying if the state produced after a power reset is equivalent to the one that the sequential execution of the same code produces, but it lacks of dedicated and automated techniques for finding where inconsistencies may happen.

- **Ekho** [20] focuses on the energy perspective and solves the problem of physically recreating the characteristics of an energy harvesting environment, with the aim of being able to perform multiple and repeatable in-lab testing. It provides both a general workflow and tools for recording and reproducing the energy harvested from a specific source. Ekho can be used to test the behavior of a program with respect to a specific energy source, but it does not address the inconsistency problem. In fact, it does not provide any technique for verifying the presence of inconsistencies, or for identifying if the state produced after a power reset is equivalent to the one that the sequential execution of the same code produces.

- **Siren** [17] is a software solution that emulates the execution of a program over the MSP430 [2] architecture, a commonly used MCU in TPC domain, and takes into account the energy perspective. For doing so, it extends MSPSim [21], an MSP430 instruction level emulator, with the concepts of energy buffer and non-volatile memory.

  Siren emulates the energy buffer, and takes into account the energy consumption produced by the execution of any instruction. Then, it uses EKHO's energy representation to reproduce the effects of harvested energy over the energy buffer.

  Siren provides a series of debugging capabilities, such as breakpoints, register and memory monitoring, and profiling of the executed code, which are designed to not interfere with the simulated energy levels.

  As for EDB, we can use Siren exposed functionalities for verifying if the state produced after a power reset is equivalent to the one that the sequential execution of the same code produces, but it lacks of

dedicated and automated techniques for finding where inconsistencies may happen.

- **CleanCut** [18] is a debugging environment which does not focus on inconsistencies, and instead focuses on the energy perspective and solves the problem of *non-terminating path bugs*. A *non-terminating path* consist in a sequence of instructions which demands more energy than the device can store. In such a scenario, the device is not able to complete such sequence and it will restart over again from the first instruction of it. As consequence, the device is stuck at the execution of this *non-terminating path*.

  CleanCut provides to the programmer a way to find if *non-terminating paths* exists inside the code, and a tool able to place checkpoints in a way which avoids them. It is not conceived to account for inconsistencies, and thus it does not provide any technique for verifying the presence of inconsistencies, or for identifying if the state produced after a power reset is equivalent to the one that the sequential execution of the same code produces.

All the available solutions are not conceived to focus on testing all the possible combinations of intermittent executions. EDB [13] and Siren [17] are designed to address the debugging problem, and thus they are the tools that are closer to our problem. Even if we can adapt EDB and Sirens for verifying the presence of inconsistencies, using Siren will limit the user to use an MSP430 [2] architecture, and using EDB requires a significant hardware intervention on the micro-controller unit, since it must be physically connected to the MCU.

Moreover, to extend these tools for recognizing inconsistencies, we still require a general workflow for analyzing where inconsistency may happen. Some checkpoint mechanisms such as DINO [1] and Ratchet [10] provide an analysis to automatically resolve the inconsistency problem. Unfortunately, such workflows are only applicable with respect to the respective checkpoint mechanism's memory configurations, and they do not consider environment interactions.

Finally, none of these tools recognize or classify different types of inconsistencies, and they also do not provide any guideline on how to avoid the unwanted behaviors introduced by frequent power resets.

1.3 Contribution

As we previously described, the literature lacks both a general workflow to analyze inconsistencies, and a tool for finding them. This thesis addresses these problems, and its contributions can be divided into two different parts.

1.3.1 Classification and Analysis of Inconsistencies

We considered the inconsistency problem from both the point of view of memory and environment interactions. For each one of these two classes, we analyzed all the possible scenarios that can cause an unwanted behavior of a program, and we provided guidelines on how to avoid such unwanted behaviors. For analyzing inconsistencies, we considered a low level of abstraction (i.e., machine code). This enables us to focus on details which otherwise we would not be able to consider.

Memory. From the memory standpoint, we identified three different types of unwanted behaviors:

- *Data Access Inconsistency*: it can happen whenever an access into NVM is performed, independently of the memory section (i.e., global variables, stack, and heap). It is the *generic* inconsistency considered in the literature [1, 10, 13, 14, 15, 16], and we analyzed it from a lower level of abstraction (i.e., machine code). As we will see, such inconsistency is caused by a power reset which happens after a sequence of instructions containing a read and write operation for the same memory location, that are not separated by any checkpoint. In fact, when the read operation will be re-executed, it will read the value written by the subsequent write operation before the previous reset.

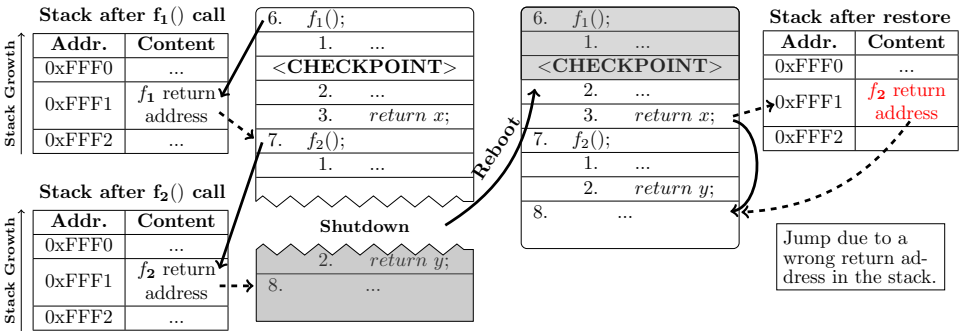

Figure 1.4: Example of an *Activation Record Inconsistency*. The stack is allocated into NVM, and thus every write is persistent across power resets. Once the function f_1 returns, it pops the stack content. When the function f_2 is called, it pushes onto the stack the return address, which goes to the same position of where the return address of f_1 was stored. As consequence, when the checkpoint is restored due to a power reset, the function f_1 will pop the wrong return address from the stack, since it was overwritten by $f_2()$ call in the previous power cycle. For this reason, the program counter is set to the address of line 8 and not to the one of line 7, entirely skipping the call of f_2.

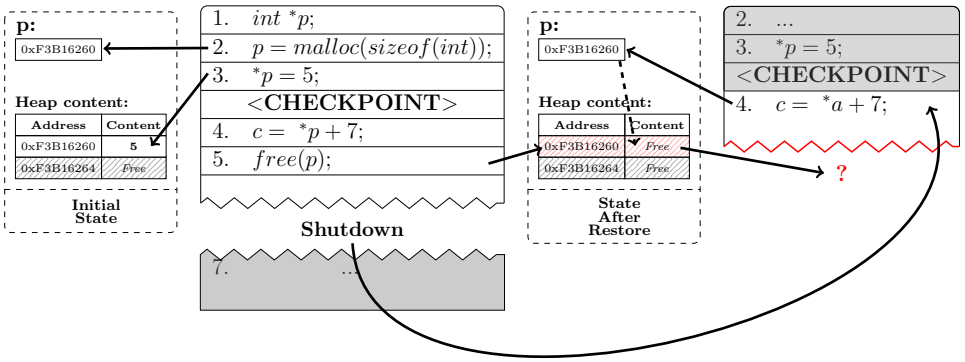

Figure 1.5: Example of an intermittent execution leading to an unexpected behavior due to the allocation of the heap segment in NVM. The variable p points to a cell which was dynamically allocated into the heap. Once the cell is freed, it is removed from the heap. In such a scenario, if a shutdown happens, the code after the checkpoint is re-executed. The following instructions expect the cell pointed by p to be available, but it does not exist anymore.

- *Activation Record Inconsistency*: it is a type of inconsistency which can happen only if the stack is allocated into NVM, and it is caused by function calls.
  Figure 1.4 shows an example of this kind of inconsistency. The call of $f_1()$ at line 6 pushes onto the stack a set of elements which includes the return address, which is the address of the instruction at line 7. When we perform the checkpoint, we are inside the context of $f1()$. When $f1()$ returns, it pops from the stack the return address, and the execution continues from line 7, which calls $f2()$. As consequence, it pushes onto the stack its return address, which is the address of the instruction at line 8. If we consider the data present in the stack when the checkpoint was taken, this operation has effectively replaced the return address of $f1()$ with the one of $f2()$. When the checkpoint is restored, the execution continues until it reaches the return instruction at line 3 of $f1()$. The return address present in the stack is the one of $f2()$, and thus as next instruction it will be executed the one of line 8 instead of line 7, resulting in an unexpected program behavior.
- *Memory Map Inconsistency*: it is a type of inconsistency which can happen only if the heap is allocated into NVM.
  As we will see, this type of inconsistency can be caused by the execution or re-execution of a function which can dynamically allocate, move or delete memory elements inside the heap (i.e., *malloc, calloc, realloc, free*). We analyzed multiple combinations of instructions which generates a Memory Map Inconsistency. Figure 1.5 shows an

example of this kind of inconsistency. The variable p is a pointer which addresses a cell inside the heap, which is in NVM. Once we execute the $free(p)$ instruction of line 5, the cell pointed by p is de-allocated. Let us suppose that, after the execution of such instruction, a shutdown happens due to a low energy buffer. When there is enough energy to restart the computation, we restore the checkpoint, and then we continue the execution from the instruction at line 4. The execution of such operation accesses the cell pointed by p, which is no longer available, leading to an unexpected behavior of the program. In fact, depending on how heap is managed, the program might crash.

For each one of these inconsistency types, we provide a general workflow for verifying their presence, and guidelines on how to solve them.

Environment. From the environment standpoint, we found and analyzed the problems of *Input Access Inconsistencies* and *Output Inconsistencies*. The intermittent execution characterizing TPCs does not only introduce an unwanted behavior from the memory standpoint, but also from environment interactions.

In the sequential execution of a program, whenever we execute an instruction that retrieves data from the environment, such as reading a sensor, we get a value that reflects the current state of the environment. For this reason, all the subsequent computation that uses directly or indirectly such data produces the expected results.

If instead we consider an intermittent execution, this same behavior is not granted. Let us suppose that we read the temperature of the environment, we perform a checkpoint and then a shut down happens due to a low energy buffer. Let us now suppose that the temperature of the environment changes while the devices is powered off. When there is enough energy to restart the computation, the previously taken checkpoint is restored, and the device continues the execution of the program. All the subsequent computation will use a temperature value which does not correspond to the current state of the environment. In fact, the state of the device obtained after the restoration of the checkpoint contains also the data retrieved from the environment during the previous power cycle. Depending on the program requirements, the computation can be considered either incorrect or correct. If we want to perform the computation using this previous version of the temperature, we can consider the results correct. Instead, if we want to use a value which corresponds to the current state of the environment, we must consider the results to be incorrect, and we have an *Input Access Inconsistency*.

Due to the presence of an intermittent execution, we can classify uses of environment-dependent data with two types of *input access models*:

- *Most Recent*: all the data computed from an input is not produced using the value of such input from a previous power cycle. In other

words, whenever the execution restarts from a checkpoint, the value of the input has to be taken from the environment, with the effect of using the most possible recent one.

- *Saved*: all the data computed from an input can be produced using the value of the input from a previous power cycle. In other words, whenever the execution restarts from a checkpoint, the value of the input may be taken from the restored state, with the effect of using the last sensed value.

The decision of using a *Saved* or *Most Recent* access model depends on the application requirements. Once the access model of an environment input is set, it is important that the data computed using the value produced by such input is consistent with the required access model over it, otherwise the program will produce incorrect results.

Environment interactions do not only consist in sensing data, but also in sending data to the environment and changing its state. We refer to these actions as *environment outputs*.

In the sequential execution of a program, whenever we execute an instruction that modifies the environment state, such as moving a servo, we obtain the expected environment state. In an intermittent execution scenario, this same behavior is not granted, due to the presence of frequent power resets. Let us suppose we have a program which incrementally moves a servo by 15°, and that its initial position is 0°. Moreover, let us suppose we perform a checkpoint, we move a servo by 15° and then a shut down happens due to a low energy buffer. When there is enough energy to restart the computation, the previously taken checkpoint is restored, and the device continues the execution of the program. As next operation, we re-execute the movement of a servo by 15°, and its current position becomes 30°. This environment state is not the same we would achieve in the equivalent sequential execution of the code. Depending on the application requirements, we can consider the obtained environment state to be either correct or incorrect. In this last case, we have an *Ouput Inconsistency* which leads to an unwanted behavior of our program.

In this thesis we provide a general workflow for verifying the presence of both *Input Access Inconsistencies* and *Ouput Inconsistencies*. Moreover, we also provide guidelines on how to make the program consistent, both from the standpoints of input accesses and environment outputs.

Intermittence as Program Input. The literature considers inconsistencies only as an unwanted behavior introduced by the intermittence property characterizing TPCs. The in-depth analysis we performed over the two different classes of inconsistencies enables us to consider them also from a different point of view. In fact, in this thesis we also consider the possibility of using inconsistencies as an input for our programs, and we also provide a way to verify the correctness of such new class of inputs.

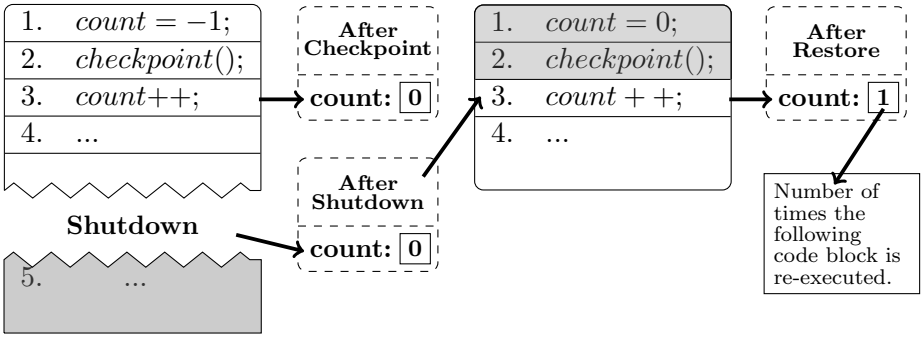

Figure 1.6: Example of an intermittence-based input, which helps us in keeping track the number of times we re-execute a certain code region due to power resets. Moreover, from another point of view, it tells us the number of happened power resets.

Figure 1.3 shows a way to use intermittence for tracking the number of times a certain code region is re-executed due to the presence of frequent power resets. For doing so, it exploits the combination of two elements: the variable *count* allocated into NVM and the instruction at line 3, which increments such variable. In this way, the variable *count* is updated whenever the checkpoint at line 2 is restored, since the computation will restart from such operation. As consequence, the variable *count* will contain the number of shutdowns that happened due to a low energy buffer. In other words, it contains the number of times we re-executed the code region after line 3.

If we take a closer look at what happens inside the data, we can see that we intentionally introduced a *data access inconsistency*. In fact, when we re-execute our code region, our variable contains a value which is different from the equivalent sequential execution of the same code. In this scenario, this inconsistency permits us to use the number of re-executions across power resets and use this data as an input of our program, and thus it is not an unwanted behavior. It is important to note that this can be achieved only if we permit the presence of such inconsistency.

Until now, the literature treated inconsistencies as an unwanted behavior of a program which can lead to the computation of incorrect results. In this thesis, we show that by permitting the presence of *Data Access Inconsistencies* in specific sections of our code and *Output Inconsistencies*, we are able to adapt the behavior of our program in a way which considers the intermittence property of TPCs as an input of our program, such as the number of resets or a function of it. Considering the possibility of interacting with the environment, this opens to scenarios in which it is possible to signal subsequent power failures, or in which is possible to perform compensation actions after a certain amount of power resets.

1.3.2 Inconsistency-oriented Debugging Tool

With the knowledge provided by the first part of this thesis, we developed `ScEpTIC`, an inconsistency-oriented debugging tool.

At the moment of writing this thesis, the most commonly used MCU in TPC is the MSP430 [2], especially thanks to the presence of an internal NVM. Due to the increasing popularity of this novel field, it is likely that other architectures with this same feature will be developed. In such a scenario, all the work developed considering the MSP430 must be adapted to account for the architectural differences. One of our goal is to address the problem of testing without sticking to a particular architecture, in such a way that none or very little modifications must be done to our tool for supporting newer architectures, which are not available at the moment in which we developed `ScEpTIC`. For this reason, we designed our tool to be architecture-independent, which makes us also able to test different configurations and architectures, without having to modify or recompile our code. To achieve this feature, `ScEpTIC` exploits these design choices:

- It runs LLVM IR [22] code, which is an intermediate representation of the source code, similar to the assembly language. Such language does not contain references to elements of the Instruction Set Architecture, nor it refers to specific memory locations, and thus we can run a simulation without the need of knowing a priori which device will be used. Moreover, it does not refer to architectural registers, and instead uses a virtual representation of them.

  Furthermore, in this way we avoid the register allocation step which is done by the compiler. This could lead to an inaccurate analysis, since this step can introduce memory accesses. For this reason, `ScEpTIC` permits us to introduce and customize a register allocation process, which will be performed over the LLVM IR and will introduce memory operations similar to the one introduced by the compiler.

  All these features permits an architectural-independent interpretation of the user provided code.

- It abstracts the memory in a way which does not require us to specify addresses nor dimensions. For doing so it only differentiates between Volatile Memory and Non Volatile Memory (NVM), and permits us to arbitrarily allocate different sections into them (i.e., global variables, stack, and heap). All the memory accesses are then automatically managed by `ScEpTIC`, transparently with respect to the user and the memory type.

- It abstracts the checkpoint/restore logic and implements the workflow of the most common checkpoint mechanisms. Usually, a checkpoint

saves into NVM the register file and the content of the volatile memory. The source code of a checkpoint mechanism contains the reference to such elements, and thus can be considered architecture-dependent. With this abstraction, we avoid the emulation of architecture-specific elements, such as interrupt service routines, voltage comparators, and other elements used to eventually trigger checkpoints during the execution of a program.

Moreover, this abstraction also enables us to test the logic of a checkpoint mechanism with a different architecture with respect to the one it was developed on. In this way, if we want to use a different architecture from the supported one, we are able to verify if the checkpoint mechanism suits our needs, without having to re-implement it over such architecture.

- It abstracts I/O ports as if they are simple user-defined functions, so that we can ignore how they are accessed and initialized. In this way, we can also specify the value returned by an input port, without having to emulate the actual component which will be connected to it when the application is deployed.

- It omits all the architecture-dependent functionalities (e.g. specific APIs), and threats them as if they are simple user-defined functions. To use such functionalities, we have to specify an implementation for them, using the abstraction methods exposed by `ScEpTIC`.

Once we configure `ScEpTIC`, and we provide the inputs used by our program, we can run four different type of analysis:

- **Memory Inconsistencies**: in this analysis `ScEpTIC` automatically verifies the presence of memory inconsistencies (i.e., *Data Access Inconsistency*, *Activation Record Inconsistency*, and *Memory Map Inconsistency*) by testing all the possible intermittent execution scenarios of the code. To reproduce the intermittence characterizing TPCs, it simulates multiple *power resets* during the execution of the code, which consist in a shut down immediately followed by the consequent restart of the computation. This analysis is optimized thanks to the knowledge provided by the previous part of this thesis. Power resets are generated only in relevant positions, leading to a reduction of the number of instructions executed and the reduction of the amount of time required to run the analysis. It is important to note that set of optimizations implemented does not cause the loss of any information about the inconsistencies, and thus the analysis is kept exhaustive.

  Once `ScEpTIC` completes this analysis, it returns us the list of all the found inconsistencies, identifying the causes and the code position where they happen.

- **Input Access Inconsistencies**: before running this analysis, we must provide an input access model for each input element. Then, `ScEpTIC` automatically verifies all the input accesses by simulating an intermittent execution of the code, and it verifies that accesses over inputs correspond to the access models we required.

  In this analysis we are interested in identifying if input accesses happen before or after a checkpoint. For doing so, `ScEpTIC` keeps track of when a checkpoint is executed, and with this information it is able to verify if the input accesses are consistent with respect to the required access model.

  We optimized this analysis in a way which does not require generating any power reset, and thus the intermittent analysis is reduced into a sequential execution of the code. As we will see in Chapter 5, for identifying *input access inconsistencies* we do not require an inconsistent memory state, and we only require information about when checkpoints happen. For this reason, we were able to omit the generation of power resets, leading to a drastic reduction of the amount of time required to run this analysis, without loosing any information on *input access inconsistencies*.

  Once `ScEpTIC` completes this analysis, it returns us the list of all the found inconsistencies, identifying the causes and the code position where they happen.

- **Profiling**: this type of analysis permits us to interact with `ScEpTIC` as if it was a semi-interactive debugging environment. Using this analysis, we can verify the correctness of intermittence-dependent inputs inside our program and environment interactions, we can verify the presence of output inconsistencies, or we can simply debug a specific intermittent execution of our code.

  `ScEpTIC` exposes two groups of debugging primitives: one to log debug information, and one to specify where a power reset should happen. Using such methods, we can fine-tune where power resets should happen and thus we can generate specific intermittent execution scenarios. In this way, we can recreate particular conditions which generate a certain value of an intermittence-dependent input and, in combination with the knowledge provided by the previous part of this thesis, we can recreate all the scenarios which enables us to exhaustively verify the correctness of our code.

  Once we set up the environment, `ScEpTIC` runs the code reproducing the intermittent execution scenario we created. When the execution finishes, `ScEpTIC` returns us the *execution trace* of the tested program. It contains the list of environment interactions executed and the con-

tent of debug information we decided to log, grouped by the power cycle in which they happen.

Finally, with the result of this analysis we can verify the correctness of the intermittent execution we recreated, and eventually we can modify our code to account for the errors we found.

- **Output Profiling**: as we will see in Chapter 5, testing output inconsistencies is a particular case of verifying intermittence-based inputs. As consequence, we can analyze output inconsistencies using the *Profiling* analysis we previously described. In fact, we can set the generation of power resets after each environment output access, and we only need to track the execution of output actions.

  For reducing the user intervention required to analyze output inconsistencies, we provide a lighter version of the profiling analysis. It automatically generates power resets after the execution of each output routine, and it only keeps track of the execution of output actions. In this way, we are not required to fine-tune where resets should happen, and we only get the information about outputs, which is the one we require for analyzing output inconsistencies.

  Once `ScEpTIC` completes this analysis, it returns us the *execution trace* of the output actions executed. With the result of this analysis we can verify output actions that are re-executed, and thus are able to establish if their re-execution can lead to *output inconsistencies*.

  This analysis recreates a set of specific intermittent scenarios, that presents power resets after the execution of output routines. If we require more control on where power resets should happen, we have to use the *Profiling* analysis.

1.4 Thesis Structure

This structure of the thesis can be divided into four different parts:

1. **Domain knowledge and problem description**

   The aim of this part of the thesis is to provide all the knowledge and basic concepts required to understand the new challenges introduced by Transiently Powered Computing and to understand the problem of testing an intermittent scenario.

   - In Chapter 2 we describe the TPC domain, and we introduce the different approaches used for preserving the work done in presence of frequent shutdowns. We consider both task-based and checkpoint-based solutions, with an in-depth description of this

last group [1,3,4,10,11,12]. In this chapter we also provide an introduction to the unwanted behavior caused by checkpoint-based solutions (i.e., inconsistencies), describing it from both control-flow and data-flow standpoints.

- In Chapter 3 we provide an in-depth description of the problem of testing intermittent execution and finding inconsistencies. We also describe the available debugging environments [13,17,18,20], and we discuss if and how they can be adapted for exhaustively analyzing intermittent executions for finding inconsistencies.

2. **Classification and analysis of inconsistencies**

   The aim of this part of the thesis is to provide complete knowledge over inconsistencies and their causes. For doing so, we analyze the inconsistency problem, and we classify different types of inconsistencies. For each one of those types, we provide an algorithm able to find the inconsistency, and we also describe how to solve such unwanted behaviors. Moreover, for doing so, we deepen into the machine level, and we analyze what the intermittent execution causes inside the register file, the memory and on the control flow.

   - In Chapter 4 we analyze and classify memory inconsistencies, which are inconsistencies introduced due to the allocation of memory sections (i.e., global variables, stack, or heap) into Non Volatile Memory (NVM). In this chapter we describe *Data Access Inconsistencies*, *Activation Record Inconsistencies*, and *Memory Map Inconsistencies*. The first type of inconsistency can happen if a memory element is allocated into NVM, independently of the memory section. Instead, the other two types of inconsistencies can happen only if respectively the stack and the heap are allocated into NVM.

     For each one of these three types of inconsistencies, we provide guidelines for analyzing their effects and verifying their presence. We also discuss how to avoid and compensate for them.

   - In Chapter 5 we analyze environment interactions, and we discuss the possibility of treating intermittence as an input of our program.

     In the first part of this chapter, we consider both the possibilities of retrieving data from the environment, and changing its state. We classify *Input Access Inconsistencies* and *Output Inconsistencies*. The first type of inconsistency may happen due to a missed re-execution of an input action, such as reading a sensor. It can lead the program to use a wrong value during the computation of input-dependent data. The second type of inconsistency

may happen due to the re-execution of an output action, such as the movement of a servo. It can produce a wrong environment state, resulting in an unwanted behavior of the program. For each type of environment inconsistency, we analyze its cause, we discuss how to avoid such unwanted behavior, and we provide a technique for finding their presence and analyzing their effects.

In the second part of this chapter, we consider the possibility of exploiting the intermittence as a new input for our programs. For introducing this possibility, we provide some examples, and we also provide a technique for verifying the correctness of this new type of input.

3. **Implementation of ScEpTIC**

The aim of this part of the thesis is to provide a description of the implementation details of `ScEpTIC` and its test mechanism implementations. `ScEpTIC` is the inconsistency-oriented debugging environment we developed using the concepts we described in the previous chapters, and it is designed to abstract its analysis in a way that permits extending or modifying them without an excessive effort.

- In Chapter 6 we describe the implementation of the debugging environment, its architectural components, and its work flow. `ScEpTIC`'s architecture is composed by different objects and recalls the structure of a Micro Controller Unit. The entire state is included into a *VMState* class, which contains the reference to the objects representing register file and memory. The intermittent execution is generated using both the *Checkpoint Manager*, which is in charge of generating and restoring checkpoints, and the *Interruption Manager*, which is in charge of generating power resets.

  Moreover, `ScEpTIC` takes as input a user-provided configuration and the LLVM IR [22] of the source file to be analyzed. Then, it converts the provided LLVM IR into an abstract syntax tree, which is then analyzed and used to initialize `ScEpTIC`'s internal components. Finally, `ScEpTIC` performs the analysis under an intermittent execution scenario by running the *Interruption Manager* specified in the configuration.

- In Chapter 7 we describe the implementation of the different analysis performed by `ScEpTIC`. Each analysis can be executed using the respective implementation of the *Interruption Manager*:
  - *Data Interruption Manager*: it runs the analysis able to find *Memory Inconsistencies*.

  - *Input Interruption Manager*: it runs the analysis able to find *Input Access Inconsistencies*.
  - *Interaction Interruption Manager*: it permits us to use `ScEpTIC` as a semi-interactive debugging environment, and thus it enables us to verify the correctness of intermittence-based inputs.

  For each one of these implementations of the *Interruption Manager*, we provide an in-depth description of how the analysis is performed, which inputs are required from the user, and which optimizations are applied to the analysis.

4. **Evaluation**

   The aim of this final part of the thesis is to evaluate the performance of `ScEpTIC` and the analysis guidelines we described in previous chapters. Since no previous work addresses the problems solved by this thesis, in Chapter 8 we consider two different approaches for performing such evaluation:

   - A *quantitative evaluation*, which compares the algorithms implemented by `ScEpTIC` and presented in this thesis with the most simple ones which anyone can think of. For doing so, we considered a broad range of quantitative metrics, such as the number of instructions executed and the number of power resets generated. Moreover, we selected different types of benchmarks that represent a set of heterogeneous use cases which are common in TPC domain.

     For performing such evaluation, we also consider different scenarios, including one in which we show how `ScEpTIC` can help a developer in the selection of a checkpoint mechanism, with the related memory configuration, for his program.
   - A *qualitative evaluation*, which compares `ScEpTIC` with other existing tools [13, 17] that are designed to account for other faces of the testing problem. To perform this evaluation, we considered the alterations required for such tools to perform the same analysis of `ScEpTIC`, and we analyzed the efforts required for achieving so. Moreover, we compared the obtained analysis capabilities with the ones of `ScEpTIC`.

Chapter 2

Transiently Powered Computing

Transiently Powered Computing (TPC) consists in using the energy harvested from the environment, such as sun light, as the only power source for devices. This chapter aims to provide a background over this novel field, so to understand some challenges that the usage of free energy introduces.

TPC is becoming more and more popular in the Internet of Things (IoT) area, especially for powering smart sensors. Batteries increases the physical

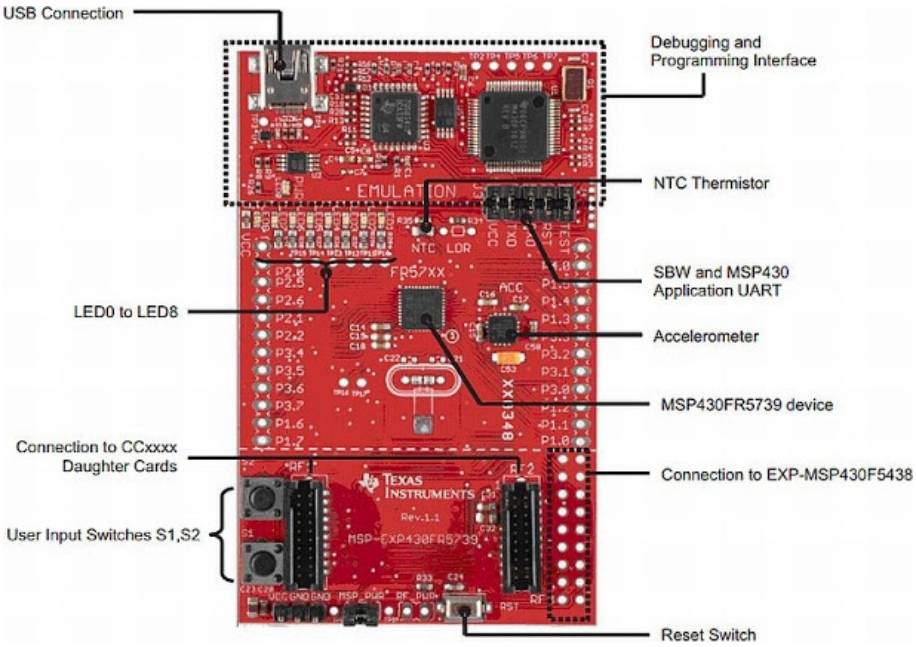

Figure 2.1: Example of a device based on the *MSP430* architecture [2].

dimension of a sensor, they must be used within certain intervals of temperature, and they have a limited lifespan. For these reasons, using batteryless systems has the potential to enable the deployment of sensors in environments considered to be impractical due to battery limitations. Furthermore, the absence of batteries reduces maintenance costs, since there is no need to replace and dispose them.

Typical devices used in TPC are the ones based on the *MSP430* [2] Micro Controller Unit (MCU). Usually they also present components for interacting with the environment, such as an accelerometer and a thermistor. Figure 2.1 shows an example of these devices.

2.1 Saving Work Done

2.1.1 Overview

Due to its nature, the energy harvested from the environment is an unpredictable power source. Powering a device with it causes an intermittent execution, that alternates periods of time in which the device is running, with periods of time in which it is completely powered off. When a shutdown happens, the content of the main memory used by the device is lost, and within it all the results of the computation produced so far. For this reason, having an intermittent execution creates the need of saving the work done, so to not start the entire computation over again.

There exist different approaches to solve this problem, and they can be identified within two classes: *checkpoint-based solutions* and *task-based solutions*. Furthermore, independently of the class, they exploit a non-volatile memory (NVM) present on the device, which is persistent and thus maintain its content even in presence of shutdowns. For example, MSP430-based boards uses a Ferroelectric RAM (FeRAM or FRAM) as NVM. It is a memory similar to a *DRAM*, and it is composed by a ferroelectric material that grants the non volatility property.

Depending on the device configuration, we can choose to use the NVM as support memory, or we can use it also as main memory.

2.1.2 Checkpoint-based Solutions

Checkpoint-based solutions save the work done using the combination of two routines:

- *Checkpoint*: this routine saves the current state of the device (i.e., registers and main memory) into a non-volatile memory.
- *Restore*: this routine restores the state of the device from the data saved by a previous checkpoint.

The general idea behind this class of solutions is to take checkpoints during runtime, so to restore them when the device restarts after a power failure.

In this way, restoring the saved state permits to resume the computation from where the checkpoint was saved, and not all over again.

There exists different implementations of checkpoint mechanisms, which we discuss in Section 2.3, and they all differ on how and where the state is saved, and on data that a checkpoint contains.

Independently of the data to be saved, we can classify checkpoint mechanisms considering how they take checkpoints:

- **Static Checkpoint Mechanism**: checkpoint routines are statically placed inside the code, and thus checkpoints are taken in fixed positions.

- **Dynamic Checkpoint Mechanism**: this category of checkpoint mechanisms exploit interrupts to take checkpoints. When an interrupt is generated, it pauses the current execution of the program and start executing the checkpoint routine. For this reason, checkpoint routines are not placed inside the code, and thus the point in which they are taken is not predictable.

2.1.3 Task-based Solutions

Other solutions exists for preserving the work done which does not involve checkpoints, and are based on the definition of tasks.

The program's code is divided into groups called tasks, which are then executed in sequence for granting the correct control flow of the program. This logical division is necessary to preserve the work done. In fact, each task uses as input the data produced by a previous one, and when it is executed, its results are stored in NVM. In this way, if a shutdown happens, when there is enough energy to resume the computation, it can restart from the task which uses as input the data stored in NVM.

To maintain this behavior, usually only input/output data of tasks is stored into NVM, and the intermediate results of each task are allocated into the volatile memory. In this way tasks are atomic, and a task can modify the values stored in NVM only when it terminates. Let us suppose a shutdown happens during the execution of a task, due to a low energy buffer. When there is enough energy, the execution restarts from the first instruction of the task, and uses as input data the one present in NVM. Since such data was not modified during the previous execution, the computation performed during the re-execution of the task produces a correct result. Alpaca [15] and Chain [16] are examples of this approach, and they differ in how data is managed and how a task can communicate results to the next one.

For how data is managed among tasks, the computation will always use correct values when it restarts, independently of where a power failure

happens. This is not the case of checkpoint-based solutions, which may present some problems from a data consistency standpoint.

The work done in this thesis focuses on the behavior of the execution in presence of inconsistent states, and for this reason we will only consider checkpoint-based solutions.

2.2 Checkpoints and Inconsistencies

We use checkpoints for saving the work done, so to not restart the computation all over again. Furthermore, we expect to obtain the same results as if the program was never interrupted, otherwise an inconsistent behavior is manifested, and such results are not usable.

To understand what an inconsistency is and how it happens, let us focus on the execution of Example 2.1. Its sequential execution sets the variable a to 4, and finally prints out "*Condition ok: a is 4*".

Let us now focus on how the same code is executed in an intermittent environment. Furthermore, let us suppose that variable a is stored in NVM and that we use a *Static Checkpoint Mechanism*. We start the execution of the *main* function, and the first instruction sets the variable a to 3. As next operation, we take a checkpoint, which saves the state of the main memory, but not the NVM content. Then, we execute the instruction at line 3, which increments variable a to 4. Let us now suppose that a shutdown happens due to a low energy buffer. When there is enough energy to restart the computation, we restore the content of the checkpoint, and thus we resume the execution from the instruction after it, which is the one at line 6. Since we did not restore the content of the NVM, the variable a is still set to 4, and thus the execution of the instruction at line 6 sets a to 5, which is an incorrect value. For this reason, the condition of the if statement is not met, and thus we take the else branch. This has the effect of printing out "*Something is wrong: a is 5*".

Not only the data produced by the intermittent execution differs from the one of the sequential execution, but also the control flow is different. In fact, during the intermittent execution is executed the *else* branch of the *if* statement. Due to these differences, the result produced by the intermittent execution is incorrect.

When the checkpoint is taken at line 5, the runtime state comprehend the value *3* for variable *a*. Instead, when it is restored, the variable a has a different value (i.e., 4), and thus the two runtime states are different. As consequence, the computation uses an inconsistent value, producing a wrong result.

In fact, whenever we restore a checkpoint, we expect that the runtime state is the same of when such checkpoint was taken. If this condition is not met, the restored checkpoint produces a state which is not consistent

```
1  int a; // stored in NVM
2
3  int main() {
4      a = 3;
5      checkpoint();
6      a++;
7      if(a == 4) {
8          printf("Condition ok: a is %d", a);
9      }
10     else {
11         printf("Something is wrong: a is %d", a);
12     }
13 }
```

Example 2.1: Example of a C program which may present an inconsistent runtime state.

with respect to the work done. As consequence, it is possible that the computation performed over an inconsistent state produces wrong results, making the entire checkpoint mechanism useless, since we are not able to use the produced data.

Having an inconsistent runtime state may cause unexpected behaviors from two different perspectives:

- **Data Flow Inconsistency**: a portion of data assumes an incorrect value with respect to the equivalent sequential execution of the program. This is the case of the value assumed by variable a in Example 2.1.
- **Control Flow Inconsistency**: a portion of code is executed a different number of times with respect to the equivalent sequential execution of the program. In this case, three cases are possible:
  - *New execution*: a portion of code that was not executed after the checkpoint is executed after the checkpoint is restored. This is the case of the *else* branch in Example 2.1.
  - *Lack of execution*: a portion of code which was executed after the checkpoint is not executed after the checkpoint is restored. This is the case of the *if* branch in Example 2.1.
  - *Re-execution*: a portion of code is re-executed after restoring the checkpoint. This scenario does not necessary cause a *Control Flow Inconsistency*, since it can be considered a property of checkpoint mechanisms. In fact, when the execution resumes, and we restore a checkpoint, the restore routine is re-executed. Depending on where the execution is restored, we may also re-execute other portions of code.

Furthermore, we can note that the presence of a *Data Flow Inconsistency* implies a *Control Flow Inconsistency*, and vice versa:

1. The first two cases of a *Control Flow Inconsistency* can only be caused by the re-evaluation of a condition in a control-flow based statement (e.g. if, for, and while) in the presence of a *Data Flow Inconsistency*. In fact, if the runtime state is consistent after a checkpoint is restored, the condition will always be evaluated with the same outcome, no matter how many times it is re-evaluated. For this reason, the absence of a *Data Flow Inconsistency* implies the absence of these two cases of the *Control Flow Inconsistency*.

2. The re-execution of a portion of code can not be always considered as a *Control Flow Inconsistency*. In fact, let us suppose that the runtime state restored after a checkpoint is consistent. The re-execution of instructions placed after the checkpoint will produce a result which is equal to the sequential execution of the code, and thus can be considered consistent. Instead, if the runtime state restored after a checkpoint is inconsistent, then the re-execution of instructions placed after the checkpoint may produce an inconsistent result. This last case is the one present in Example 2.1, that happens when we re-execute the instruction of line 6. For these reasons, a re-execution of a portion of code can be considered as a *Control Flow Inconsistency* only in the presence of a *Data Flow Inconsistency*.

Finding an inconsistent behavior by analyzing the presence of *Control Flow Inconsistencies* requires also verifying the presence of *Data Flow Inconsistencies*. In fact, for the re-execution case we must consider the consistency of produced data.

Thanks to the above property, we can infer that *Data Flow Inconsistencies* are the cause of *Control Flow Inconsistencies*. For this reason, we can verify the presence of an inconsistent behavior by only focusing on the produced data, thus omitting the analysis of the control flow.

Finally, in Example 2.1 the inconsistency of runtime state is caused by the fact that checkpoints do not include the data present in NVM. One may think to solve the inconsistency problem once for all by including the NVM inside the data saved by the checkpoint. Even if this is a valid option, it is not always applicable. In fact, performing checkpoints consumes energy, and the more data it saves, the more energy it consumes. This increased energy consumption reduces the number of instruction that can be executed in between checkpoints, especially when the energy buffer is not refilled by the energy source. If, instead of a single variable, we have a large array, it may be more efficient to allocate it in the NVM and to solve data inconsistencies.

2.3 Checkpoint Mechanisms

The aim of this section is to provide a basic knowledge of some checkpoint mechanism, which introduce relevant concept for the analysis we do in this thesis. For each one of them we will specify the checkpoint placement, the checkpoint work flow, the saved data, and the presence of Data Flow Inconsistencies. Furthermore, for DINO and Ratchet we will also focus on the key-concepts they introduce on *Data Flow Inconsistencies*, which we exploit in this thesis for an in-depth analysis of such problem.

2.3.1 MementOS

MementOS [3] consists in a combination of two elements:

- A library exposed to the user, containing checkpoint and restore routines.
- A pass for the LLVM compiler framework [23], which can be used to automatically place checkpoints inside the code.

It is a *Static Checkpoint Mechanism*, but for the way in which checkpoints are taken, it exposes a dynamic-like behavior. In fact, when a checkpoint routine is encountered, it measures the current voltage of the energy buffer, which is used to estimate the remaining energy. The checkpoint is taken only if such voltage is below a certain threshold, otherwise it is ignored.

The LLVM pass can place checkpoints in two different positions:

- *loop-latch*: a checkpoint routine is placed at the end of each loop body, with the effect of calling such routine at every iteration of the loop. An example is shown in Example 2.2.
- *function-return*: a checkpoint routine is placed after each instruction which calls a function, with the effect of calling such routine every time a function returns. An example is shown in Example 2.3.

```
1 [...]
2 for(i = 0; i < 10; i++) {
3     [...]
4     checkpoint();
5 }
6 [...]
```

Example 2.2: Example of a loop-latch placement of MementOS [3].

```
1 [...]
2 function1();
3 checkpoint();
4 [...]
```

Example 2.3: Example of a function-return placement of MementOS [3].

Furthermore, the user can choose to not use the LLVM pass to place checkpoint routines, and instead to manually place them.

MementOS does not consider the possibility of allocating elements into NVM, and thus it does not include such memory into the checkpoint content. For this reason, it may present data flow inconsistencies.

2.3.2 DINO

DINO [1] can be considered as a solution in-between a *Static Checkpoint Mechanism* and a *Task-based approach* for saving the work done. It mainly focuses on maintaining data consistency and considers the possibility of allocating global variables into NVM.

DINO does not provide an automatic checkpoint placement mechanism and it consists in a set of passes for the LLVM compiler framework [23], that analyzes and modifies the code provided by the user, in combination to a runtime library.

The general idea behind DINO is requiring the user to split the program into tasks, by specifying a series of task boundaries, using the function *DINO_task()*. An example is shown in Example 2.4, in which we have three different tasks: one at line 5, one comprehending lines 7 and 7, and one at line 10.

Each task is executed as if it corresponds to an *atomic* instruction, and thus the data alteration produced by it is seen by other tasks only if it completes. To achieve that, DINO saves a snapshot of the runtime state every time it reaches a task boundary.

This is the reason why we can consider DINO as both a *Checkpoint-based* and *Task-based* solution:

- If we think *DINO_task()* as a marker indicating the end of a task and the beginning of another one, we have a *task-based* approach.
- If we think *DINO_task()* as a marker indicating where a checkpoint takes place, we have a *checkpoint-based* approach.

Saving only volatile information during checkpoints does not ensure data consistency. In fact, if a DINO task modifies the content of a variable addresses in NVM, such alteration is seen even if the task does not complete. To overcome this problem, DINO performs *versioning* of NVM variables at each task boundary.

To better understand the analysis performed by DINO for versioning variables, let us focus on Example 2.4 and let us suppose only the variable a is allocated in NVM. The only operation changing the value of variable a is the $a{+}{+}$ instruction at line 8. If such operation is executed and suddenly a shutdown happens, we will encounter an inconsistent runtime state. In fact, when the execution restarts, it is restored the checkpoint taken at the task boundary of line 6, and the first instruction executed is the one at line 7. It uses the value of variable a, which was modified to 4 by the execution of

```
1  int a = 3; // Allocated in NVM
2  int b;
3
4  int main() {
5      b = 1;
6      DINO_task();
7      b = b - a;
8      a++;
9      DINO_task();
10     return b % a;
11 }
```

Example 2.4: Example of a program using DINO.

line 8, before the shutdown. This alteration is seen even if the task has not completed before, and thus we have an *atomicity violation* which causes an inconsistent runtime state. To account for this problem, DINO saves also the value of variable *a* within the checkpoint, when it encounters the task boundary at line 6. In this way, when the checkpoint is restored, within it is also restored the previous value of variable *a*, obtaining a consistent runtime state.

DINO achieves the described solution by performing the following analysis:

1. It analyzes the control flow graph (CFG) of the program for finding all the operations writing a variable allocated in NVM.

2. For each write operation found:

    2.1. For all the possible paths, it travels backwards the CFG for finding the most recent *DINO_task()* in each path.

    2.2. For each task boundary found, it searches the operations in the interval between it and the considered write operation. If, in such interval, is present an instruction reading the same variable addressed by the write operation, DINO is required to save the value of such variable (i.e., *versioning* it) when it executes the checkpoint associated to the task boundary.

Furthermore, when DINO complies the program, it replaces the *DINO_task()* call with two operations of its runtime library:

- *dino_version()*: it saves into NVM a copy of NVM variables for which the analysis has found a potential inconsistency.

- *dino_checkpooint()*: it saves the content of the stack and register file into NVM.

Finally, whenever the execution restarts after a power failure, DINO restores the value of the versioned NVM variables, the stack content, and the register file content. In this way, even if a power failure happens after a write operation modifies a NVM variable, the combination of *dino_version()* and restore routine ensures data consistency, because such variable will be restored to a previous version.

DINO only considers the possibility of allocating variables into NVM, and for this reason it may present *Data Flow Inconsistencies* when the stack and/or heap are allocated into NVM. The entire analysis it performs can be easily adapted to perform versioning of stack elements, and thus it does not present inconsistencies when the entire stack is allocated into NVM. Unfortunately, this is not the case for heap: DINO analyzes the data flow graph and the control flow graph, but heap can be arbitrarily modified during runtime, and thus analysis of data accesses becomes very difficult. Furthermore, once a heap mapping action is performed (i.e., *malloc*, *ralloc*, *free*), it is not possible reverting the heap state only with versioning. DINO can be certainly adapted to work with heap, but at the current state can show inconsistencies when it is allocated into NVM.

2.3.3 Ratchet

Ratchet [10] is a *Static Checkpoint Mechanism* which uses NVM as main memory. It consists in a set of passes for the LLVM compiler framework [23], which analyze the code to automatically place checkpoint in a way which avoids data flow inconsistencies.

To achieve that, Ratchet exploits the idea of *idempotency*: it considers a sequence of instruction idempotent if its re-execution does not produce an inconsistent runtime state. The implemented LLVM pass analyzes the code and splits it into idempotent groups. Then, between each group it places a call to the checkpoint routine.

Considering that Ratchet uses NVM as main memory, an inconsistent runtime state is obtained whenever write operations are performed after a read operation to the same memory location (i.e., write after read, WAR). Let us consider Example 2.5: instruction at line 5 reads the value of variable a, and the one at line 6 writes it. This is a non-idempotent sequence of code, since exists a *write after read* within it. In fact, let us suppose a shutdown happens after the execution of line 6. When the computation restarts, the obtained runtime state is inconsistent, since the value of a is 7 and not 3, and thus a wrong result is produced.

For avoiding the described problem, Ratchet analyzes the code for finding all the WAR hazards, and thus it places a checkpoint between each write and read operation. In this way, each sequence of instructions delimited by checkpoints is idempotent. In Example 2.5, Ratchet puts a checkpoint between line 5 and 6, solving the inconsistency.

```
1 int a = 3;
2 int b;
3
4 int main() {
5     b = a + 1;
6     a = 7;
7     return a + b;
8 }
```

Example 2.5: Example of a program presenting a WAR hazard.

Ratchet works on a lower level of abstraction with respect to the C language. It runs the analysis using the LLVM intermediate representation, which is a language similar to assembly. This is required because ratchet stores also the stack into NVM, and thus all the machine-level instructions which interact with the stack must be taken into account. Furthermore, the analysis considers each memory read operation in the program, and for each of them verifies the presence of an instruction that writes same read memory location. If so, it places a checkpoint between the read ans write operations, so to create an idempotent sequence of instructions.

Since Ratchet uses NVM as main memory, performing a checkpoint consists in saving only the register file into NVM (i.e., program counter, stack pointer, and general purpose registers). Furthermore, since the stack is allocated into NVM, the number of placed checkpoints is considerably high. To overcome checkpoint overhead, Ratchet optimizes the number of registers that are saved by a checkpoint, by saving only the ones which are required for the computation. This is achieved by analyzing each idempotent sequence, so to find the registers used by it. Then, it tunes accordingly the checkpoint routine placed before each idempotent sequence, so to save only such registers.

Ratchet does not present any data flow inconsistency for stack and global variables, but it does not consider the heap segment. Its analysis can not be performed over such dynamic memory, because memory addresses are not available during compile time, and thus is not possible to verify accurately the presence of WAR hazards for this segment. A conservative approach would be to place checkpoints between each pair of instructions which respectively reads and write the heap, but it would introduce even more overhead, especially if heap is accessed frequently. Furthermore, heap can be dynamically re-mapped during runtime, and this complicates the conditions required for granting consistency. For these reasons, Ratchet may present inconsistencies if heap is used in the program.

We may think to overcome the inconsistency problem by modifying Ratchet to allocate heap in SRAM. Unfortunately, even if this action solves the inconsistencies, it would increase a lot the overhead introduced by each

checkpoint, and it would be more advantageous using another checkpoint mechanism.

Finally, the idea behind the analysis Ratchet performs for placing checkpoints is similar to the one we will describe in Chapter 4 for analyzing and finding data flow inconsistencies, which is implemented by `ScEpTIC`. Our tool is able to analyze also different memory configurations with respect to the one used by Ratchet, but if we consider having all the main memory allocated into NVM, Ratchet and `ScEpTIC` seems to perform the same analysis which leads to the same results. They both focus on pairs of memory reads and writes for performing the analysis, but the information returned to us is completely different. In fact, Ratchet places checkpoints inside the program to avoid inconsistencies, but it does not tell us where they can happen and what is the cause. Instead, `ScEpTIC` returns the list of inconsistencies with their causes, but it does not produce a checkpoint placement.

2.3.4 Hibernus / Hibernus++

Hibernus [11] is a *Dynamic Checkpoint Mechanism* implemented over a MSP430 architecture. The entire system relies on a library exposed to the user, and on the internal on-chip comparator for generating the interrupt which triggers the checkpoint.

This checkpoint mechanism is based on two actions:

- *hibernate*: it saves a snapshot of the system state (i.e., registers and main memory) and puts the MCU into low-power mode.
- *restore*: it restores the system state from the previously-saved snapshot.

The system is initialized by configuring the interrupt handler to perform the *hibernate* action, and by setting the voltage threshold for performing hibernation (V_H). When the voltage level of the energy buffer goes below V_h, the on-chip comparator triggers an interrupt, which calls the *hibernate()* function. Once a snapshot of the system is saved, Hibernus sets the voltage threshold for performing the restore action (V_R), and re-configures the interrupt handler to perform the *restore* action. Finally, it sets the system to sleep. When the voltage level of the energy buffer goes above V_R, the on-chip comparator triggers an interrupt, which calls the *restore()* function.

To use Hibernus, the user is required to put the initialization routine into its code, and to specify the voltage thresholds for hibernate and restore actions, which are very critical for the stability of this checkpoint mechanism.

To overcome the problem of finding valid voltage thresholds, Hibernus++ [4] extends the functionalities provided by Hibernus with automatic calibration of such thresholds. The general idea of this calibration is to increment V_H if an invalid snapshot is found by the *restore()* routine, since it means that the energy was not enough for saving the entire system state. We will not enter into details of this functionality, since it is not relevant

from the point of view of the analysis we perform in this thesis.

Even if Hibernus does not take into consideration the possibility of allocating elements directly into NVM, it does not present any *Data Flow Inconsistency*. In fact, when the *hibernate* routine is called, the MSP430 is put into a low-power mode, which disables the CPU. This means that no further computation is performed until there is enough energy, and thus consistency is granted over the elements present in NVM.

2.3.5 QuickRecall

QuickRecall [12] is a *Dynamic Checkpoint Mechanism* that uses only NVM as main memory, and thus allocates in it global variables, stack, and heap. To achieve this result, it exploits a modified linker, which maps all the memory sections of the object file (i.e., .text, .bss, .data, and .stack) into NVM. Since these sections are persistent, performing a snapshot of the system state requires only to save the content of the register file (i.e., general purpose registers, stack pointer, and program counter).

As for Hibernus, QuickRecall relies on an on-chip comparator which triggers an interrupt whenever the voltage level of the energy buffer goes below a certain voltage threshold V_{Tr}. This threshold must be specified by the user, and the entire checkpoint mechanism of QuickRecall is implemented in the *Interrupt Service Routine* (ISR) associated with the interrupt signaling a low energy buffer.

When the execution starts, QuickRecall verifies if the *checkpoint flag* is set, that is a global variable. If this condition is not met, it means that no checkpoint is available, and thus the program starts from the beginning. When the voltage goes below V_{Tr}, the comparator triggers the interrupt that stops the normal execution and changes the context to the one of the ISR. Firstly, it pushes onto the stack the general purpose registers and the stack pointer, then it sets the *checkpoint flag*, and finally it pushes into the stack the current program counter. Note that the pushed *pc* refers to the ISR and not to user's program, since the program counter of where the interrupt is generated was saved during the changing of the context. Now, QuickRecall waits until the voltage level of the energy buffer is higher than V_{Tr}. If this condition is met, the ISR simply returns, since no restore action is required. Otherwise, at a certain point the MCU will shutdown, since the energy buffer will be totally emptied. When there is enough energy to restart the computation, QuickRecall verifies if the *checkpoint flag* is set, and thus it restores the saved stack pointer, the general purpose registers, and the program counter. The restored context is inside the ISR, and the instruction to be run is the one that waits for the voltage to be higher than V_{Tr}. When this happens, the ISR returns and the program resumes the execution from where it was stopped.

Finally, QuickRecall does not present any *Data Flow Inconsistency*. In

fact, it pauses the computation whenever a checkpoint is taken, and thus no alteration to the NVM can be made. Furthermore, the way in which registers are saved keeps them synchronized with the state of the NVM, and thus the runtime state is always consistent.

Chapter 3

The Problem of Testing

3.1 Overview

As we explained in Section 2.2, the intermittence characterizing TPCs causes an unexpected behavior of the execution flow, which will differ from the sequential run of the same code. The presence of frequent shutdowns causes the re-execution of some portion of code, even if we are using a checkpoint mechanism, and it might cause the computation of incorrect results [14]. However, the fact that the intermittent execution of a program presents an execution flow different from the equivalent sequential execution, does not imply that results are incorrect.

To better understand this statement, let us focus on Example 3.1. It represents a program which uses a static checkpoint mechanism to preserve its state between power outages. Furthermore, let us suppose that this example uses MementOS [3] as checkpoint mechanism. The *checkpoint_check()* function call present at line 4 consists in verifying the presence of a previously saved checkpoint, and if it is present, the *checkpoint_check()* function restores it. Depending on the checkpoint mechanism, such routine could be set explicitly as in the example or it can be omitted. In this second case, the final executable is modified in a way that runs such routine before the *main*, and if a checkpoint is not present, it transfers the control to the *main* function.

Let us now verify what happens during the execution of the program. The execution starts from *checkpoint_check()* and, since a checkpoint is not available, it does not perform any action. Now, the execution continues until it reaches the *checkpoint()* call at line 7, and thus a checkpoint is saved. Let us suppose there is enough energy to execute the instruction at line 8, and then a shut down happens due to a low energy buffer. When there is enough energy to restart the computation, it executes the first instruction of the *main* function. Now, *checkpoint_check()* is re-executed and it restores the previously saved checkpoint. Note that we say this instruction is re-executed

```
1  int a;
2
3  int main() {
4      checkpoint_check();
5      a = 3;
6      [...]
7      checkpoint();
8      a = a + 1;
9      [...]
10     return a == 4;
11 }
```

Example 3.1: Example of a code with a static checkpoint mechanism.

```
1  int a;
2
3  int main() {
4      Hibernus();
5      a = 3;
6      [...]
7      a = a + 1;
8      [...]
9      return a == 4;
10 }
```

Example 3.2: Example of a code with a dynamic checkpoint mechanism.

because in a sequential execution it would be executed only once, and in this case it is executed for a second time. Since this operation does not alter the computed data, and just restores it, it does not affect the result. Now, the execution continues from line 8, which is re-executed for a second time, and two scenarios are possible:

1. **Variable *a* is in NVM**: since we are using MementOS as checkpoint mechanism, such variable is not included within the checkpoint data. This means that the first execution of line 8 modified a to *4*, and the second execution of such instruction incremented it to *5*. This is an incorrect result.
2. **Variable *a* is not in NVM**: such variable is included within the checkpoint data, and thus no problem happens since its value is kept consistent thanks to the checkpoint mechanism.

We can also note that the execution flow of the intermittent execution of the program is different from the sequential one [14]. If we consider the line numbers executed, we see the following execution traces:

- **Sequential**: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow ...$
- **Intermittent**: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow \lightning \rightarrow 1 \rightarrow 8 \rightarrow ...$

There is nothing we can do to avoid this difference between the execution flows, since power outages are a characteristic of TPCs. As we can see, the re-execution of an instruction does not necessarily invalidate the computation done so far, but it depends on the type of instruction, the checkpoint mechanism used, and the memory configuration.

A similar reasoning can be applied with a dynamic checkpoint mechanism. For doing so, let us focus on Example 3.2, which is the same program used in the previous example, but without the static checkpoint placement. It uses Hibernus [11] as checkpoint mechanism, and the *Hibernus()* func-

tion performs the same actions described for *checkpoint_check()*, with the addition that it also enables the interrupt which triggers the checkpoint action. Let us suppose that we reach the instruction at line 7 and then the energy buffer is not sufficient to continue the execution. In this case, it is triggered the interrupt that saves the checkpoint. Then, the execution is paused until the energy buffer is either restored or totally emptied. In the first case, no re-execution happens, since it simply resumes from where it paused. In the second case, the MCU shuts down and, when there is enough energy to restart the computation, it resumes from the first instruction. The *Hibernus()* function is re-executed and then the program continues from instruction at line 8, since a checkpoint is present. As for the previous example, the re-execution of the *Hibernus()* routine does not interfere with the result, since it does not change the computed data. Furthermore, since no instruction after the checkpoint is re-executed, no inconsistency happens.

The described scenarios create the needs of analyzing if and where inconsistencies may happen due to the intermittent execution of a program, and if such inconsistencies can interfere with the computed results. To verify the correctness of a sequential execution of a program, the literature provides different types of code testing methodologies (e.g. unit testing), but unfortunately we are not able to apply such methodologies with an intermittent execution, since they are not conceived to account for power outages. Furthermore, very little work exists for analyzing the presence of inconsistencies in an intermittent execution environment.

Different checkpoint mechanisms try to solve the inconsistency problem, mainly with two different approaches:

- Static checkpoint mechanisms such as DINO [1] and Ratchet [10] provide ways to overcome the inconsistency problem, by analyzing where *Data Flow Inconsistencies* might happen, so to place compensation actions accordingly.
- Dynamic checkpoint mechanisms such as Hibernus [11] and QuickRecall [12] are designed to not present inconsistencies. In fact, when they save a checkpoint, the execution pauses and thus the re-execution of instruction is limited to the startup function which verifies the presence of a checkpoint and restores it.

As we can see, the objective of checkpoint mechanisms is also preserving data consistency, to produce an intermittent execution of the code which yields a result equivalent to the sequential execution. Unfortunately, the concepts provided within checkpoint mechanisms applies for their memory configuration and target architecture. They can be certainly adapted to work with other architectures, and their inconsistency analysis can be used as starting point to produce generic guidelines and a tool able to analyze the presence of inconsistencies.

Moreover, the higher is the number of memory elements allocated into NVM, the higher is the number of inconsistencies. When the user selects a

certain checkpoint mechanism for his program, he is also selecting the associated memory profile, without knowing which kind of inconsistencies it introduces. Let us suppose we want to select a checkpoint mechanism for a program, and that we want to allocate into NVM a certain variable. If we choose Ratchet [10] as checkpoint mechanism, the entire NVM is used as main memory, and thus each memory operation can potentially cause an inconsistency. As we explained in Section 2.3.3, to keep the state consistent Ratchet inserts a checkpoint before each write instruction, and thus it introduces a high overhead, in terms of both energy consumed and instructions executed for performing checkpoints. If, instead, we choose MementOS [3], no memory element is allocated into NVM, apart from our variable, and thus the only inconsistencies which can be present are the one related to the accesses of such variable. In this configuration, the number of checkpoints we must insert to keep the state consistent is drastically reduced with respect to the one of Ratchet, and thus it is also reduced the checkpoint overhead. Depending on our requirements, the overhead introduced by Ratchet can be justified or not. This recurrent problem creates the need of being able to test different memory profiles, so to understand the overhead introduced by moving a certain memory section (i.e., global variables, stack, heap) into NVM, and thus justify or not the usage of a checkpoint mechanism which allocates more memory into NVM with respect to our requirements.

Furthermore, the available work done on inconsistencies by DINO [1] and Ratchet [10] describes them at a source-language level, but then they

```
1  int a = 0;
2
3  int main() {
4      a = a + 1;
5      return a;
6  }
```

Example 3.3: Example of a variable increment in C.

```
1      .globl  a
2      .data
3      .type   a, @object
4      .size   a, 4
5  a:
6      .long   1
7      .text
8      .globl  main
9      .type   main, @function
10 main:
11     pushq   %rbp
12     movq    %rsp, %rbp
13     movl    a(%rip), %eax
14     addl    1, %eax
15     movl    %eax, a(%rip)
16     movl    a(%rip), %eax
17     popq    %rbp
18     ret
```

Example 3.4: Assembler version of Example 3.3.

perform their analysis using a lower level of abstraction. As we will see in Chapter 4, we need to go deeper toward machine level to actually describe and recognize inconsistencies. In fact, let us focus on Example 3.3, which represents a variable increment. The equivalent assembler version is shown in Example 3.4. As we can see, the variable increment is translated into different machine operations: line 13 loads the value of *a* into a register, line 14 increments the register by 1, and finally line 15 saves the content of the register back into *a*.

If we reconsider the intermittent scenario described for Example 3.1, the result is inconsistent due to the re-execution of the instruction at line 8, which increments the variable *a* by one. Such increment is converted into the operations described in Example 3.3, and thus is executed as three different operations:

1. *movl*, which loads the content of the memory location associated to variable *a* into a register.
2. *addl*, which increments the register value by 1.
3. *movl*, which stores the value of the register into the memory location associated to variable *a*.

The cause of the inconsistent result is the re-execution of the two *movl* instructions. If, for example, we can move the checkpoint just after the first *movl*, the inconsistency would not happen, since the value of *a* would be included in the checkpoint. In this case, the re-execution of the second *movl* changes the data saved into the memory, but such data is never re-loaded, since the operation which performs such action is before the checkpoint, and thus the inconsistency does not happen.

As we saw in the previous example, if we do not consider the machine level, we are not able to perform an accurate analysis of the problems introduced by an intermittent execution. Furthermore, we also need to find a general workflow for verifying, analyzing, and testing the results of an intermittent execution, independently of the checkpoint mechanism adopted, the micro controller unit used, and the nature of the harvested energy.

The lack of a well-defined testing workflow for intermittent executions makes it not possible to recognize where hazards to the data consistency may happen in the code. This leads the programmer to the selection of a checkpoint mechanism without taking into account which kind of problems the code may present and where, with the potential result of poor performance and poor energy optimization, due to checkpoint placements and consequently introduced overhead.

Apart from the general workflow for finding inconsistencies, we also require an environment able to perform such analysis. Accordingly to Hester et al. [19], the problem of testing intermittent execution is open and, due to the unstable and unpredictable nature of the harvested energy, no available tool is able to predict and explore all the possible combinations of intermittent executions. Also, no testing environment or tool is provided to verify

```
1 checkpoint();
2 // critical code
3 [...]
```

Example 3.5: Code section on which we want to track the number of re-executions.

```
1 a = 0; // in NVM
2 checkpoint();
3 a = a + 1;
4 // critical code
5 [...]
```

Example 3.6: Solution for tracking the number of re-executions of Example 3.5.

```
1 read temperature sensor
2 checkpoint()
3 if temperature > 30 then
4     activate cooling
```

Example 3.7: Example of an environment interaction.

```
1 [...]
2 checkpoint()
3 move antenna
4 measure signal intensity
5 [...]
```

Example 3.8: Example of an environment interaction.

the presence of inconsistencies, and the available tools present in the literature consist mostly in debugging environments with energy reproduction capabilities.

Until now, all the previous works present in the literature considers inconsistencies as an unwanted behavior. As we will se in Section 5.3, we can look at the intermittence characterizing TPCs as a feature granting new inputs for our program. Let us focus on Example 3.5, and let us suppose we want to track the number of times the code portion between line 2 and line 3 is executed. Furthermore, let us suppose for a moment that we are not focusing on the presence of data inconsistencies. Example 3.6 shows how to track the number of times the code region is re-executed: before the execution of the critical code region we increment variable a, which is in NVM. If a reset happens due to a low energy buffer, when there is enough energy to restart the execution, it continues from the checkpoint at line 2. The first instruction to be executed is the increment of variable a, which is in NVM and thus retains the value of previous re-executions. Thanks to its presence in NVM, variable a contains exactly the number of times the code region at line 4 is entered. If we do not allow the presence of an inconsistent value for variable a, we are not able to track the number of executions for the code region of interest, and there is no way of doing so without permitting the presence of an inconsistency. Since no previous work considers intermittence as a device input, there is no tool or guidelines for verifying the correctness of such usage scenario.

Finally, another problem not faced by previous works concern environment interactions. The most common application of devices in TPC comprehend their usage as wireless sensors, and thus their work flow consists in sensing the environment, processing the sensed values, and then store the results or send them to the main node of the system. The presence of frequent shutdowns can cause not only data inconsistencies, but also unwanted interactions with the environment. Let us focus on the execution flow described in Example 3.7: our device reads the temperature of the environment and then, if it is higher than 30 degrees Celsius, it activates the cooling system. Let us suppose that our device senses 31 degrees Celsius from the environment and then just after the execution of the checkpoint at line 2, a shutdown happens due to a low energy buffer. Let us now suppose that a long time elapses after the execution restarts, and that the environment temperatures drops down to 20 degrees Celsius. When the execution resumes, the checkpoint at line 2 is restored and within it, the previously sensed temperature. Since the saved value was 31 degrees Celsius, the cooling system is activated as next step. This clearly shows an unwanted behavior of our program, which we can refer as environment inconsistency.

A different inconsistent behavior of an environment interaction is shown in Example 3.8, which consists in a program used to measure the intensity of a signal in different directions, by moving an antenna with a servo. Let us suppose that our servo is moved by 1 degree each time, and that its initial position is 0 degrees. When the execution reaches line 3 for the first time, the servo is moved to 1 degree. Let us suppose that a shutdown happens just after the movement of the servo, due to a low energy buffer. When the execution restarts, the checkpoint is restored and as subsequent operation the servo is moved again by 1 degree, reaching the position of 2 degrees. The measurement of the signal intensity which happens at the next instruction records the value with the antenna positioned at 2 degrees. As we can notice, due to the shutdown, we have skipped the measurement with the antenna positioned at 1 degree, and thus our measurements are incomplete.

Even if it is possible to debug environment interactions with available tools [13, 17], we still need a general analysis workflow for environment inconsistencies, otherwise we are not able to find where problems may happen and how to solve them.

3.2 Available Tools

This section will provide a general view over the main tools available in the literature that addresses the problem of debugging from different perspectives. As we will see, they are designed with different goals, and they do not address the inconsistency problem.

3.2.1 EKHO

Ekho [20] focuses on the energy perspective and solves the problem of physically recreating the characteristics of an energy harvesting environment. It provides both guidelines and tools for recording and reproducing the energy harvested from a specific source. One of its goals is also granting the possibility of conducting multiple and repeatable in-lab testing, from the point of view of energy reproduction.

Ekho records the energy by measuring the current intensity (I) and voltage level (V) supplied to the test device by the energy harvester in different load conditions. Such measurements are used for generating an I-V curve, which describes how voltage and current consumption are related, and it characterizes the energy source.

To reproduce the energy source, Ekho exploits a programmable energy environment which consists in a tool able to reproduce the same energy type of the energy source. Then, it uses the recorded I-V curve profile to regulate such programmable energy environment. For example, to reproduce the sun light it uses a lamp and regulates it accordingly to the I-V curve, by sensing the amount of energy emitted.

Ekho is very useful to conduct repeatable in-lab experiments, and to test the behavior of the program with respect to an energy source, but it does not provide any program analysis methodology or techniques for finding inconsistencies.

3.2.2 EDB

Energy-interference-free Debugger or *EDB* [13] focuses on providing a debugging environment able to not interfere with the energy state of the device to be tested. It consists in both a software and hardware solution, and it is composed by:

- **Debugging Device**: it must be connected to the device we want to test, and permits us to interact with it for debugging purposes. It maintains the energy level constant during the debugging actions, so to not introduce energy interference due to debugging operations, and can set a specified energy level in the tested device buffer. Furthermore, it is able to sense I/O events, the energy level, and application events triggered by the exposed software API, and it provides a debugging console which allows us to interact with the tested device and to gather the sensed information.

- **Software Library**: it provides a library called *libEDB* which must be included in the program to be tested. Such library permits us to insert the following debugging operations inside the code:

- *watchpoint*: it triggers a program event which is sensed by the hardware components of EDB, and it is signaled through the debug console.
- *breakpoint*: it pauses the program execution and permits us to inspect the current state while maintaining the energy level constant. EDB permits us to use three different types of breakpoints:
  - *code breakpoint*: it pauses the program execution when it is reached.
  - *energy breakpoint*: it pauses the program execution when the energy level goes below a certain value.
  - *combined breakpoint*: it pauses the program execution when it is reached and the energy level is below a certain value.
- *energy guard*: it runs a specified portion of code while imposing a constant energy level.
- *assertion*: it verifies a condition, and if it is not met, EDB enters an interactive debug mode, as if it reached a breakpoint.
- *printf*: it is an energy-interference-free version of the C *printf*, which prints in the debugging console the specified information.

When EDB performs a debugging operation, it always provides to the tested device a constant energy, so to compensate for the energy consumed by such debugging operation.

Even if EDB acknowledges the inconsistency problem as *intermittence bug*, it lacks of dedicated and automated techniques for finding or analyzing where inconsistencies may happen.

Despite the lack of an analysis methodology, through the usage of exposed software functionality, we are able to verify if computed results are equivalent to a sequential execution of the same code, and we are also able to set the energy levels causing a reset in precise points of the code. Furthermore, we could put breakpoints to trigger resets at every line of the code, and verify the partial result computed by each line, but this operation is inefficient and might not discover every inconsistency. In fact, as stated previously, we need to deepen into the machine level to better understand and analyze inconsistencies.

3.2.3 Siren

Siren [17] is a simulator built on top of MSPSim [21], an instruction level simulator for the MSP430 [2] micro-controller. It introduces energy and NVM capabilities in the features available within MSPSim, and runs directly the firmware to be uploaded on the target device.

To reproduce an energy source, Siren uses the same I-V curves concept introduced by Ekho [20], and it is both able to use an energy profile registered

using Ekho, or to generate one accordingly to the user's requirements. Once we specify such energy profile, Siren uses it in combination with the physical model of a capacitor to emulate the energy buffer level: the execution of each instruction reduces the energy available in the energy buffer, which might be refilled accordingly to the I-V curve profile of the energy source.

Siren extends the debugging features of MSPSim (i.e., breakpoints, registers and memory monitoring, and profiling of executed code) with a function named *siren_command()*, which can be called inside our source code to be tested. Its execution does not interfere with the energy buffer and it allows us to perform debugging operations. This function has a similar syntax of the C *printf* function: we specify as first argument a string containing a recognized command, and then we can specify other variables as further arguments, which will be included within the command using the same format strings of the *printf* function (e.g. *%d, %s, ...*). Siren provides two commands for debugging purposes:

- ***PRINTF***: prints out the specified string as if in its place there was a call to *printf()* with the same arguments.
  For example, `siren_command("PRINTF: %d\n", x)` prints out the value of variable *x*.

- ***GRAPH-EVENT***: signals to Siren a program event. All the events encountered in this way can be plotted over a graph with respect to the associated energy traces by using the provided command within the Siren environment.

Note that to use the debugging functionalities, we must include in our source code the library header containing the *siren_command()* definition.

Finally, to start a simulation, we must provide the compiled firmware containing the program to be tested with the additional debugging function calls, and the energy profile. Then, we are able to analyze the behavior of the simulated environment and to debug the program execution.

Unfortunately, Siren lacks of dedicated and automated techniques for analyzing the inconsistency problem. As for EDB, we are still able to verify if the computed results are equivalent to the ones that a sequential execution of the same code produces, but we must create the conditions that cause a power reset and the consequent re-execution of an instruction altering the consistency.

3.2.4 CleanCut

CleanCut [18] is a debugging environment that focuses on the energy perspective and solves the problem of *non-terminating path bugs*.

A *non-terminating path* is a sequence of instructions that does not include any checkpoint, and that for its execution requires more energy than

the device can store. During the execution of such sequence, it is possible that the device is not able to gather enough energy from the harvesting source, and thus a power failure happens. When there is enough energy to restart the computation, it will restore the latest checkpoint, and the computation will continue from the first instruction of the *non-terminating path*. This same process will be repeated over again, since the device is not able to gather energy fast enough to sustain the execution of such sequence, and the absence of a checkpoint in the *non-terminating path* makes the device unable to save the work done. As consequence, the execution is stuck at running this sequence of instructions and it is not able to continue.

To find *non-terminating paths*, CleanCut uses a statistical model of the energy consumption. To create this model, it measures the energy consumption of the code by exploiting the debugging capabilities of EDB [13]. CleanCut inserts EDB's watchpoints inside the code, and then it measures the level of the energy buffer when they are encountered during the execution. These measurements are then used to generate a statistical model of the energy consumption of the code, which is then used to estimate the energy consumption of all the paths (i.e., sequences of basic blocks) present in the code.

CleanCut is composed by two different components, which can be used independently:

- **Checker**: it verifies the absence of non-terminating paths within a given source file in which checkpoints are already placed by the user. For doing so, it takes as other inputs the capacity of the energy buffer and the statistical model of the energy consumption of the code. Then, for each path it estimates the energy consumption, and calculates the probability which such path has to exceed the capacity of the energy buffer. If such probability is higher than zero, CleanCut alerts the developer of the presence of the non-terminating path.

- **Placer**: it performs checkpoint placements over a given source file in a way which grants the absence of non-terminating paths. For doing so, it considers DINO [1] as checkpoint mechanism and uses an iterative algorithm to automatically place its checkpoint boundaries.

  The iterative algorithm estimate at each step the energy consumption of each path, accordingly to the given statistical model. Then, it selects the path with the highest energy consumption and, if it exceeds the capacity of the energy buffer, CleanCut's placer splits such path in half by placing a checkpoint boundary in the middle of it. Then, the algorithm repeats this process over again until no path exceeds the capacity of the energy buffer. In this way, it grants the absence of non-terminating paths.

> Once the iterative algorithm finishes, CleanCut's placer exploits the LLVM passes of DINO [1] to convert the checkpoint boundaries into actual checkpoints and to create the required versioning of variables.

CleanCut is a very useful tool for placing task boundaries and for finding the presence of *non-terminating paths*, but it is not designed to account for the inconsistency problem, and it does not provide any analysis methodology or technique for finding them.

3.3 Testing Environment Requirements

As previously stated, the existing tools are designed to solve different problems, and they do not directly consider inconsistencies.

Ekho [20] focuses only on the energy recording and reproduction perspective, and CleanCut [18] focuses only on the analysis of *non-terminating paths*. EDB [13] and Siren [17] are designed to address the debugging problem, but currently they are not suitable for automatically testing the presence of inconsistencies in the intermittent execution of the code. Moreover, except for CleanCut which verifies the energy consumption, all the available solutions do not analyze or verify any property of the executed code.

Even if it is possible to adapt EDB and Sirens for verifying the presence of inconsistencies, using Siren will limit the user to use an MSP430 architecture, and using EDB requires a significant physical user intervention on the micro-controller unit. Furthermore, Siren does not permit interactions with the energy buffer with the same flexibility provided by EDB, and thus creating the conditions for generating a shutdown in specific code locations becomes more complicated. On the other hand, EDB runs debugging directly on the physical element to be tested, and this certainly slows down testing. Moreover, they are not designed to be inconsistency-based testing environment, and even if we do not consider these limitations, we are still requiring a way and a general workflow to automatically find where inconsistencies might happen, and how to analyze them.

The main problem to be addressed by a testing environment is to verify the correctness of the data produced by the execution of the program, independently of where and when a shutdown due to a low energy buffer happens. Since we must conduct an exhaustive test and the intermittence characterizing TPCs is not predictable, we must consider that a shutdown may happen at any instant, and thus during the execution of any line of the machine code. For this reason, emulating an energy source becomes useless for the scope of this analysis.

The main aspect for verifying the presence of inconsistencies is to grant a result which is the same of the equivalent sequential execution of the same program. For this reason, for verifying the absence of inconsistencies, we are required to verify that the any combination of an intermittent execution

of a program produces a result which is equivalent to the one produced by the sequential execution of the same program.

To perform the required operations, we require an environment able to produce an intermittent execution of the program to be tested, and such environment must be able to automatically analyze all the produced memory states, so to find if an inconsistency is present and where.

Another important aspect is that using a software-based solution will certainly make the analysis easier. In fact, running these type of tests over a hardware-based solution creates more challenges, especially for generating resets and analyzing memory content.

As stated before, the tools already present in the literature do not meet the described requirements, since they are designed for other purposes. An inconsistency-oriented testing environment must focus on finding where inconsistencies happen and what causes them. To overcome the lack of both a general workflow and an environment for finding inconsistencies, we developed `ScEpTIC`, an inconsistency-oriented testing environment. Moreover, in Chapter 4 and Chapter 5 we will explain the methodologies we use for analyzing inconsistencies.

Chapter 4

Memory-section Specific Inconsistencies

4.1 Overview

An intermittent execution introduces the possibility of *intermittence bugs* [1, 10, 13, 14] or *inconsistencies*, where programs reach a state unattainable in a continuous execution. In this chapter we address such problems, and we analyze them from the machine-code standpoint. For simplicity and better comprehension, the examples we use in the following sections use a toy version of the assembly language, for which Table 4.1 and Table 4.2 show respectively the elements and instructions available. Moreover, in our examples we consider a static checkpoint mechanism that does not restore the NVM content.

Element Name	Description
R_i	Represents a general purpose register.
R_{rt}	Represents a register containing the return value of a subroutine call.
PC	Represents the program counter.
BP	Represents the stack base pointer, which is the address of the first cell at the base of the current stack frame.
SP	Represents the stack pointer, which is the address of the first free cell on top of the stack.

Table 4.1: Elements of the assembly language used in the examples.

Instruction	Description
PUSH R_i	Stores the value present in register R_i on top of the stack. It also increment SP.
POP R_i	Stores the value present at the top of the stack into the register R_i. It also decrements SP.
LOAD x, R_i	Reads the value stored at the memory address x and saves it into the register R_i.
STORE x, R_i	Stores the value of register R_i inside the memory cell at address x.
ADD R_x, R_y, R_z	$R_x = R_y + R_z$
MOV R_x, R_y	Saves the value of register R_x into register R_y.
CALL routine	Calls the routine named *routine*.
RET	Returns from function call.
BRANCH (*condition*), LABEL	This instruction modifies the execution flow of the program. It evaluates the condition, and if it corresponds to a logical *True*, it forces the program counter to the address corresponding to the specified label, effectively making the instruction after the label to be the next one to be executed. Otherwise, the execution continues from the next instruction, as normal.
L:	Sets a label named L on the current line.
CHECKPOINT	Executes a checkpoint.

Table 4.2: Instructions of the assembly language we use in the examples of this chapter.

4.2 Data Accesses

4.2.1 NVM and Memory Sections

We can use the NVM not only for allocating checkpoints, but alto for moving inside it some portions of the main memory of the program, which consists in global variables, stack, and heap. Even if global variables usually reside on top of the stack, it is possible to address a portion of them directly inside the NVM, without having to move in it also the entire stack.

When we take a checkpoint, the state it saves is synchronized with the

entire program data. If a portion of the program's main memory is allocated into the NVM and it is not part of the checkpointed state, each alteration to it will disrupt such synchronization. This can potentially lead to data inconsistencies and happens because the memory alteration remains visible when we restore the checkpoint. An example of this scenario follows in the next section.

4.2.2 Data Access

Let us consider the execution of the code that Example 4.1 shows, in which only the variable a resides in NVM. When we perform the checkpoint of line 2, it saves the register file, the stack content, and the heap content, but it does not save the value of variable a, since it is in NVM. We can consider the state that the checkpoint saves to be synchronized with the value of the variable a. Now, we continue the execution, and we run the instruction of line 5, which alters the value of the variable a that becomes 3. Let us now suppose that a shutdown happens due to a low energy buffer. When there is enough energy to restart the execution, the startup routine restores the state that the checkpoint previously saved, and the execution resumes from the instruction at line 3. Such instruction loads the value of variable a from the NVM, and stores it into the register R_0. Here is where the data inconsistency happens: the value that the instruction read for variable a is 3, but at the time at which the checkpoint was taken it was 7. This problem does not prevent the execution of the following instructions, but the entire result is compromised. In fact, the subsequent operations use an incorrect value of variable a, and they will produce an incorrect result.

```
1 STORE a, 7
2 CHECKPOINT
3 LOAD a, R0
4 ADD R1, R0, 7
5 STORE a, 3
6 [...]
```

Example 4.1: Data access in NVM that causes an inconsistency.

```
1 STORE a, 7
2 CHECKPOINT
3 LOAD a, R0
4 CHECKPOINT
5 ADD R1, R0, 7
6 STORE a, 3
7 [...]
```

Example 4.2: Fix to the data access inconsistency that Example 4.1 has.

4.2.3 Data Access Inconsistencies

We define that a **Data Access Inconsistency** may happen whenever there is an ordered sequence of instructions $I_1, ..., I_n$ such that:

1. I_1 is a LOAD operation that loads from an address x
2. I_n is a STORE operation that writes at the same address x
3. x is an address in the NVM space
4. no CHECKPOINT exists in the interval $I_1, ..., I_n$

If a checkpoint is not present between the LOAD and STORE instructions, the LOAD operation is re-executed when the state is restored. This means that the memory alteration done by the STORE instruction affects the value that the LOAD operation reads from memory, potentially leading to an inconsistency.

If the STORE operation does not alter the value present at the address x, the presence of such sequence can not lead to a *data access inconsistency*. In fact, if a power failure causes the LOAD instruction to be re-executed, it reads a consistent value from the NVM. We consider this case as a *false-positive*. Otherwise, if the STORE operation does alter the value present at address x, such sequence generates a *data access inconsistency*.

This type of inconsistency can happen independently of the portion of main memory we allocate into NVM. It can be just a subset of the global variables as we show in Example 4.1, the entire stack, or the entire heap.

To solve such inconsistency we may think that it is necessary to save also the NVM whenever a checkpoint happens, so to impose a synchronization between the elements composing the state and thus granting consistency. Although this solution works, it is not effective because we will get the same affects as having the variable allocated in the volatile memory, but with an increased access costs and checkpoint overhead. This has the effect of losing the benefits of addressing such variable into NVM, which is the reason why we might choose in first place to pay such increased access costs.

An effective and sufficient solution that makes us able to maintain a consistent state is to place a CHECKPOINT after every LOAD operation which reads a variable from the NVM, for which a subsequent STORE exists before the next checkpoint. In this way, the alteration of the memory that a STORE operation introduces will not be seen by the associated LOAD operation, because it will not be re-executed again if a shutdown happens. Moreover, the correct value of the variable will be included in the state the checkpoint saves, inside the target register of such LOAD operation. This is exactly the idea on which some checkpoint mechanisms such as DINO [1] and Ratchet [10] are built on, which makes them able to overcome data access inconsistencies.

By applying such guideline to Example 4.1 we obtain the code that Example 4.2 shows. Let us suppose that we reach the STORE operation at line 6, and then a shutdown happens. When there is enough energy to restart the execution, the startup routine restores the saved state and the execution resumes from the instruction at line 5. As we can see, we do not re-execute the LOAD operation, and the results that the instruction 5 produces is the one we expect.

4.3 Stack

4.3.1 Stack, Activation Record and Function Calls

The stack portion of the program's main memory contains global variables, local variables and the activation record of function calls. Allocating it into NVM may generate two class of inconsistencies: one representing the data access inconsistency we described in Section 4.2, and one involving function calls.

Since the whole stack is allocated into NVM, data access inconsistencies may happen whenever a LOAD operation reads from the stack. This means that we must be cautious whenever we access a global variable, a local variable, or the activation record of a function.

To understand the second class of inconsistency, let us initially focus on how a subroutine call works at a low level of abstraction, and which data it writes into the stack. There exists multiple calling conventions that specify how parameters, registers and return value should be managed when performing a subroutine call. Their differences reside in how and where they write data into the stack, but the type of data they store in it is more or less the same. For example, the calling convention that most of the C compilers use for the x86 architecture states that:

- Parameters are pushes onto the stack in reverse order by the caller, and then the function can be called.
- The return value of the subroutine is stored into a special register.
- The caller is in charge of removing from the stack the parameters once the function returns.

Usually this convention is slightly modified so that arguments are passed using registers to the called routine, due to a better access speed and latency. In this case, if the number of parameters exceeds the one of available registers, they must be stored onto the stack. When control is transferred to another subroutine, it can access and modify each register present in the architecture. This creates the need of preserving register values that the caller routine needs when the control is transferred back to it.

```
1  # Save two values in registers and two onto the stack
2  MOV 4, R5
3  MOV 3, R4
4  PUSH 2
5  PUSH 1
6  # Save caller-save register R0 on top of the stack
7  PUSH R0
8  # Function call
9  call MY_FUNC
10     # Function Prologue
11     # Save return address
12     PUSH PC
13     # Save current base of stack
14     PUSH BP
15     # Create a new stack frame, which starts
16     # on top of the stack.
17     MOV SP, BP
18
19     # Function Code
20     [...]
21
22     # Function Epilogue
23     # Restore old stack pointer
24     MOV BP, SP
25     # Restore saved base pointer
26     POP BP
27     # Restore saved return address
28     POP PC
29     # Return
30     RET
31 # Restore saved value of R0
32 POP R0
33 # Sum R0 and the return value
34 ADD R1, R0, Rrt
```

Example 4.3: Complete code of the call $MY\_FUNC(1, 2, 3, 4)$

Those instructions are generated by the compiler's back-end. For limiting the overhead of such operation, it splits the register file into two logical subgroups: callee-save and caller-save. They represent the groups of registers for which respectively the callee and the caller needs to preserve the value. Moreover, during the register allocation process, the compiler usually tries to minimize the number of registers saved onto the stack.

Let us consider Example 4.3, that represents the call of MY_FUNC with arguments $1, 2, 3, 4$. Let us suppose that we can only use the registers R_4 and R_5 to pass the function's arguments, and that the function MY_FUNC uses the register R_0, which the compiler considers a caller-save register. The first group of operations consists in setting the arguments of the routine we want to call, and it is usually done in reverse order of parameters. In the

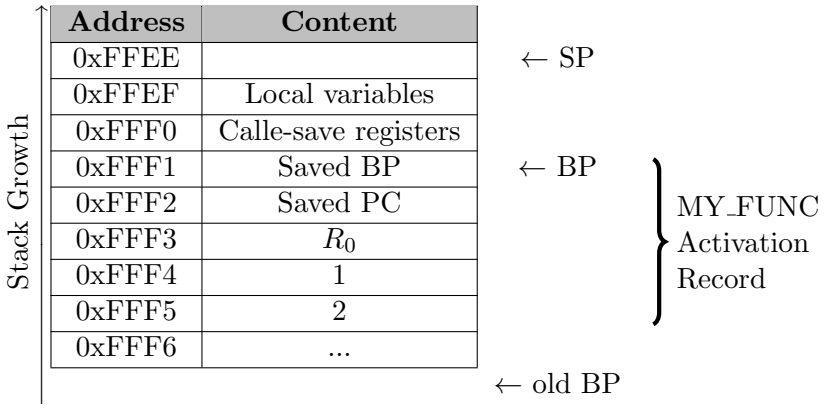

Figure 4.1: Stack content after running the function prologue in Example 4.3

example, we pass the last two parameters using registers, and the first two using the stack. The subsequent group of operations consists in saving the caller-save registers that we need after the return of the subroutine. In this case, we only save R_0. As next operation, we execute the call instruction, and then we run the *function prologue*. It saves into the stack the return address, that corresponds to the next Program Counter (PC) and the base of the stack (BP). Then, it creates a new stack frame by setting the stack base pointer (BP) equal to the current top of the stack, and it finally transfers the control to the actual function code. Figure 4.1 shows the content of the stack after running these operations. Also, note that in the *function prologue* also happens the savings of callee-save registers, if the *function code* uses any of them, and it also makes space into the stack for local variables.

Once the subroutine reaches its end, we execute *function epilogue*, which performs in revers order the operations that the *function prologue* did. In fact, it restores the saved values of the callee-save registers and the previous stack frame. Then, it returns the control back to the caller, by getting the saved return address from the stack and forcing it into the program counter.

4.3.2 Function Calls and NVM

Let us now focus on the execution of Example 4.4. Also, let us suppose that F_1 and F_2 use respectively the caller-save registers R_0 and R_1, and that we pass arguments using only the stack.

We start executing the program, and when the execution reaches line 14, we take a checkpoint inside the context of $F1$. The saved state is synchronized with the current information present in the stack, that Figure 4.2 shows, which is not included in the data that the checkpoint saves because the stack is allocated into NVM. We continue the execution sequentially until we reach the function call F_2 of line 34. Let us suppose that the execution

```
1  # Save parameters for F1
2  PUSH 2
3  PUSH 1
4  # Save register
5  PUSH R0
6  call F1
7       # Function Prologue
8       PUSH PC
9       PUSH BP
10      MOV SP, BP
11
12      # Function Code
13      [...]
14      CHECKPOINT
15      [...]
16
17      # Function Epilogue
18      MOV BP, SP
19      POP BP
20      POP PC
21      RET
22 # Restore saved register
23 POP R0
24
25
```

```
26 # Increment R0 with returned value
27 ADD R0, R0, Rrt
28
29 # Save parameters for F2
30 PUSH 4
31 PUSH 3
32 # Save register
33 PUSH R1
34 call F2
35      # Function Prologue
36      PUSH PC
37      PUSH BP
38      MOV SP, BP
39
40      # Function Code
41      [...]
42
43      # Function Epilogue
44      MOV BP, SP
45      POP BP
46      POP PC
47      RET
48 # Restore saved register
49 POP R1
50 [...]
```

Example 4.4: Code of two subsequent calls of functions F_1 and F_2

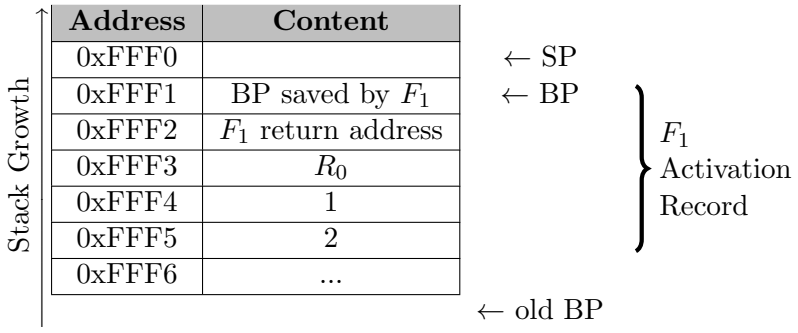

Figure 4.2: Stack content after execution of *CHECKPOINT* in Example 4.4

reaches instruction at line 37, which is inside function F_2, and then a shutdown happens due to an empty energy buffer. Figure 4.3 shows the current state of the stack, which is preserved during the power off. When there is enough energy to restart the computation, the program resumes from the instruction of line 14, which is inside the code of function F_1. Unfortunately,

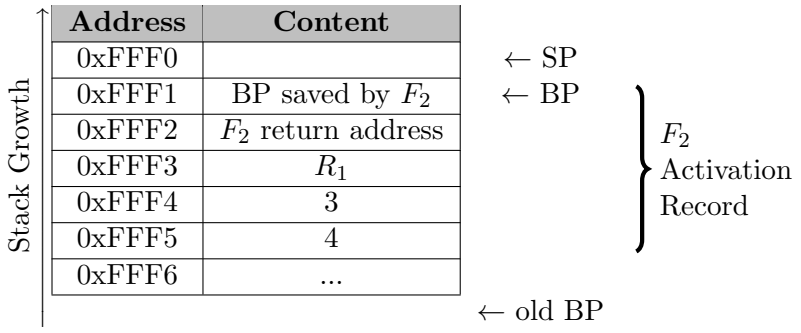

Figure 4.3: Stack content after execution of line 37 in Example 4.4

the stack contains part of the activation record of function F_2, since it is persistent due to its presence in NVM. From now on, all the computation that happens can be considered incorrect, since it uses invalid argument values, presenting a series of *data access inconsistencies*, which we described in Section 4.2. In fact, F_1 should be executed using 1 and 2 as arguments values, but due to the execution of lines 30 and 31, now it uses the values 3 and 4, which are the ones of F_2, and thus an invalid result may be produced. Moreover, when we reach the epilogue of function F_1 at line 17, it restores a wrong stack base pointer and a wrong return address, which are the ones stored into stack by function F_2 before the shutdown. As consequence, when we reach the return operation of line 21, the next instruction fetched is the one at line 47 instead of the one at line 23. The program skips a portion of code due to the wrong return address, leading to an inconsistent state.

A similar problem can happen if we configure our program with interrupts. Let us consider Example 4.4, that we previously used for describing the activation record inconsistency problem. Further, let us suppose that we reach the instruction at line 29, and then an interrupt is triggered. The MCU pauses the execution of the program, and executes the Interrupt Service Routine associated with the triggered interrupt. The ISR pushes into the stack the address where the program was paused, and it runs the code that we associated with such interrupt. When the execution of the ISR ends, it pops from the stack the address where the program was paused, and forces it into the program counter. In this way, the execution continues from where it was paused.

Let us suppose that, during the execution of the ISR, a shutdown happens due to a low energy buffer. Figure 4.2 shows the state of the stack during the checkpoint. Since the ISR was triggered after the return of the function $f1$, the activation records of $f1$ and ISR overlap. As consequence, the ISR overwrites the return address of $f1$ with the address of line 29, just as $f2$ did in our previous example. When there is enough energy to restart

the execution, we restore the checkpoint and the execution continues from line 15. When we reach the *ret* instruction at line 21, it pops the return address from the stack, which points to line 29 instead of 22, leading to the same unwanted behavior we previously described.

Ratchet [10] identifies this particular case as a potential cause of inconsistencies, but it does not analyze all the effects that *Activation Record Inconsistencies* may produce.

4.3.3 Activation Record Inconsistency

We define an **activation record inconsistency** as a particular case of a **data access inconsistency** that happens whenever there is an ordered sequence of instructions $I_1, ..., I_n$ such that:

1. I_1 is a CALL operation
2. $I_2, ..., I_x$ is the ordered sequence of instructions representing the *function prologue* of I_1
3. $I_{x+1}, ..., I_y$ is the ordered sequence of instructions representing the *function code* of I_1
4. $I_{y+1}, ..., I_z$ is the ordered sequence of instructions representing the *function epilogue* of I_1
5. Exists at least a CHECKPOINT operation I_i inside the context of the function that I_1 calls, $(2 < i < z)$.
6. I_a is a CALL operation at the same call level of I_1, with $a > z$.
   Note that two subsequent function calls C_1 and C_2 are at the same *call level* if C_2 is executed after the return of C_1 , or vice versa.
7. I_n is the *function epilogue* of I_a, $(n > a)$
8. No other CHECKPOINT exists in the interval $I_z, ..., I_n$

In other words, this kind of inconsistency may happen if two non-nested consecutive function calls exist, such that the second call can overwrite a portion of the activation record of the first one. If we have a checkpoint between the change of the context from the first to the second call, the inconsistency can not happen. In fact, whenever a power failure takes place, the execution will resume from a consistent context with respect to the entered function.

Moreover, if we configure our program for using interrupts, an **activation record inconsistency** may happen after the return of every function containing a checkpoint. In fact, the ISR may be executed after the return of a function, and it overwrites the addresses where the activation record of

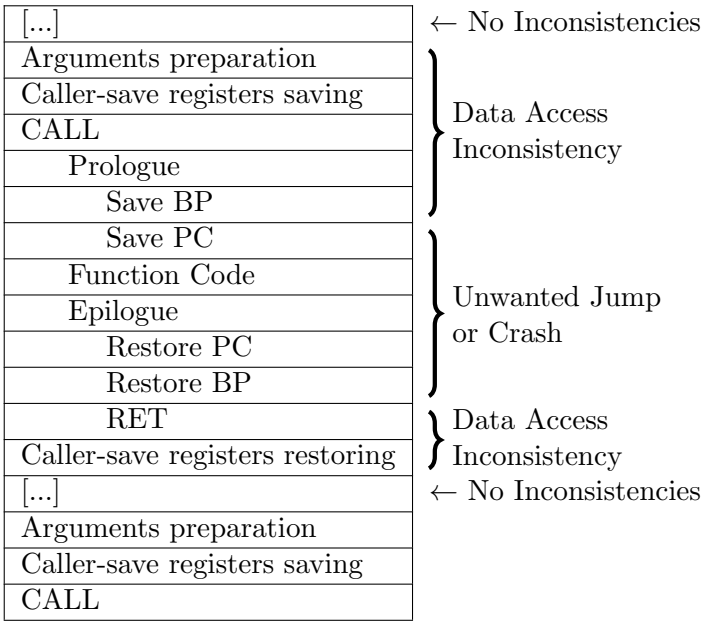

Figure 4.4: Different kind of inconsistencies given the checkpoint placement.

such function was. For this reason, a subsequent function call is no longer required for creating the conditions of this kind of inconsistency, since interrupts can happen at any instant and the ISR has the same effects of a function call. As consequence, the presence of interrupts relaxes the constraints of our previous definition, and removes the conditions 6 and 7.

After we encounter *Activation Record Inconsistency*, there is no guarantee on the possibility of continuing the execution. In fact, if the two calls have a different number of arguments, the second routine may overwrite the return address of the first one with the value of an argument. As consequence, the program may crash since the return address that the *RET* instruction forces into the program counter is not a valid one.

Figure 4.4 shows the different type of inconsistencies that may happen depending on the checkpoint placement. If we place a checkpoint before the *Arguments preparation*, no inconsistency happens, since a power failure makes the program to resume outside the context of the function.

Instead, if we place a checkpoint inside the interval from *Arguments preparation* and *Save BP*, we may experience an inconsistency, but no unexpected jump or crash. In fact, a power failure makes the program to resume from the point in which it stores into the stack the BP and PC, but the arguments and saved registers may be inconsistent.

If we place a checkpoint inside the interval from *Save BP* to *Restore BP*, we may experience an unwanted jump or a program crash. In fact, a power failure makes the program resume inside the context of the function. If a

future function call changed the return address present in the stack, when we execute the *RET* instruction, we access its return address instead of the one of the function we are in.

Lastly, another inconsistency may happen if we place the checkpoint inside the interval from *RET* to *Caller-save registers restoring*. In such case, we do not experience an unwanted jump, since the *RET* instruction accesses a consistent return address. Instead, if a future function call altered the stack position where the caller-save registers are stored, we restore them into an inconsistent state.

To solve an *Activation Record Inconsistency* is sufficient placing a checkpoint between the two calls that may generate an inconsistency. In this way, we maintain a consistent activation record with respect to the context of the function where we resume the execution.

For example, in Table 4.4 we can move the *CHECKPOINT* of line 14 to line 26. In this way, we can not resume the execution inside the context of the first function with the stack content representing the activation record of the second one.

Finally, if we configured interrupts in our program, we must modify the interrupt handler so that it takes a checkpoint before executing the actual interrupt. In this way, if a reset happens during the execution of the ISR, we resume the computation at the instruction where the interrupt was triggered. As consequence, even if the ISR overwrites the activation record of a previous function call, it can no longer cause an inconsistency. In fact, we restore our state outside the context of such function, and the data that the ISR overwrites is no longer accessed by a return instruction.

4.4 Heap

4.4.1 Heap Structure and Management

The heap is a dynamic memory section inside the program's main memory in which we can dynamically allocate or de-allocate data elements during runtime. Figure 4.5 shows how the main memory is usually partitioned. As we can see, the heap is generally placed at the opposite side of the stack, and grows from lower addresses to higher ones.

The runtime heap management depends on the environment and architecture, and usually programming languages expose abstract interfaces to interact with it. Table 4.3 shows the list of the functions that the C language exposes for managing the heap, and we will use them for generalizing the common access methods.

Whenever we execute a *malloc* instruction, it allocates a memory block inside the heap. Such memory block consists in a group of memory cells such that their overall dimension is equal to the one that we required. We can access a memory block at any position. Usually, whenever we execute

Address	Content	
0x0000	[...]	
0x0B00	**HEAP**	↓ Heap Growth
0x0B01	[...]	
[...]	[...]	
0xFFFE	[...]	↑ Stack Growth
0xFFFF	**STACK**	

Figure 4.5: Partitioning of the main memory.

Function Name	Description
$malloc(bytes)$	Allocates into the heap a given amount of bytes and returns the address of the first cell of the memory block.
$calloc(bytes)$	Allocates into the heap a given amount of bytes, initializing them to zero, and returns the address of the first cell of the memory block.
$realloc(address, bytes)$	Modifies the overall dimension of the specified memory block, eventually moving it to fit such dimension. It then returns the address of the first cell of the memory block.
$free(address)$	De-allocates and frees the given memory block.

Table 4.3: List of functions permitting heap management during runtime.

a $free$ request over the address of a memory cell, such request de-allocates the entire memory block that contains the memory cell.

All the different ways for allocating and de-allocating dynamic memory sections have the common objective to avoid memory fragmentation. When we free a memory block, it is likely marked as garbage. If an allocation request happens and a garbaged memory block able to fit the required size exists, it will be associated to such request.

For simplicity, in the examples of this chapter we will use the notation $R_i = fun(arg_1, ..., arg_n)$ for calling heap functions. This instruction means that we execute the call of fun with arguments $arg_1, ..., arg_n$ and, if the target register R_i is specified, the return value will be stored into it.

As we will see in the next sections, allocating the heap into NVM may cause a series of inconsistencies, due to the possibility of altering the heap structure during runtime.

4.4.2 Data Accesses

For understanding how the allocation of the heap into NVM may cause problems to memory accesses, let us consider the execution of Example 4.5. Table 4.4a shows the status of the heap after we execute the checkpoint of line 4. Let us suppose that we continue the execution until we run the $free$ instruction of line 7, and then a power failure happens due to a low energy buffer. Table 4.4b shows the current state of the heap. When there is enough energy to restart the computation, we restore the checkpoint and then the execution resumes from the instruction of line 5. Such instruction loads the value present at the address that the register R_0 contains, that is $0x0B00$, but during the previous execution, the $free$ instruction marked such address as garbage. We are not able to predict the outcome that the re-execution of the $LOAD$ instruction of line 5 may produce. In fact, it depends on how the architecture manages accesses to garbaged memory blocks. Such access may read the old value, may read an invalid value, or it can cause a crash of the program due to the access to an invalid memory location. All but the first hypothesis lead to a *Data Access Inconsistency* induced by the alteration of the heap state that the $free$ instruction of line 7 produced.

```
1  R0 = malloc(1)
2  R1 = malloc(1)
3  [ . . . ]
4  CHECKPOINT
5  LOAD R0 , R2
6  [ . . . ]
7  free(R0)
8  [ . . . ]
9  R3 = malloc(1)
10 [ . . . ]
```

Example 4.5: *Memory Map Inconsistency* due to a load-free sequence of function calls.

```
1  R0 = malloc(1)
2  R1 = malloc(1)
3  [ . . . ]
4  CHECKPOINT
5  STORE R0 , R2
6  [ . . . ]
7  free(R0)
8  [ . . . ]
```

Example 4.6: *Memory Map Inconsistency* due to a store-free sequence of function calls.

Address	Status
0x0B00	Allocated (ref. R_0)
0x0B01	Allocated (ref. R_1)
0x0B02	Free
[...]	Free

(a) After the *checkpoint* ot line 4.

Address	Status
0x0B00	Garbage (freed)
0x0B01	Allocated (ref. R_1)
0x0B02	Free
[...]	Free

(b) After the $free$ ot line 7.

Table 4.4: Status of the heap in Example 4.5

A similar case happens in Example 4.6, in which instead of the *LOAD* operation there is a *STORE*. In such case, the *STORE* instruction may fail since it tries to access a garbaged memory location.

Furthermore, let us instead suppose that the power failure happens after the execution of the *malloc* instruction of line 9. As we can see in Table 4.4b, the memory block that R_0 addresses is marked as garbage. During the execution of such *malloc*, the memory block $0x0B00$ will be assigned to this request, since the heap is managed for avoiding data fragmentation. When there is enough energy to restart the computation, we restore the checkpoint of line 4, and then the execution resumes from the *LOAD* instruction of line 5. Such instruction may read an inconsistent value, since it accesses the memory block that was allocated by the future *malloc* instruction of line 9.

In the previous examples, we show that a $free$ instruction may cause an inconsistency when we re-execute a *LOAD* or *STORE* operation. Example 4.7 shows a similar problem, that instead a *realloc* instruction causes. Let us analyze what happens during the execution of such code. Table 4.4a represents the status of the heap after we execute the checkpoint at line 4. Let us now suppose that we reach the *realloc* instruction of line 7, and then a power failure happens due to a low energy buffer. Table 4.5 shows the current state of the heap, in which we can see that the memory location $0xB00$ is marked as garbage, and the *realloc* instruction moved the memory block contained in such address to $0xB02$. When there is enough energy to restart the computation, we restore the checkpoint of line 4, and then the execution resumes from the *LOAD* instruction of line 5. As for the previous cases, such instruction is likely to fail, because the addressed memory block was moved by the *realloc* operation. In fact, the target address of such *LOAD* operation is $0x0B00$, which is marked as garbage.

A similar case happens if instead of a *LOAD* operation there is a *STORE*, as we can see in Example 4.8. The re-execution of the *STORE* instruction may fail since it tries to access a garbaged memory location.

```
1  R0 = malloc(1)
2  R1 = malloc(1)
3  [ . . . ]
4  CHECKPOINT
5  LOAD R0 , R2
6  [ . . . ]
7  R0 = realloc(R0, 3)
8  [ . . . ]
```

Example 4.7: *Memory Map Inconsistency* due to a load-realloc sequence.

```
1  R0 = malloc(1)
2  R1 = malloc(1)
3  [ . . . ]
4  CHECKPOINT
5  STORE R0 , R2
6  [ . . . ]
7  R0 = realloc(R0, 3)
8  [ . . . ]
```

Example 4.8: *Memory Map Inconsistency* due to a store-realloc sequence.

Address	Status
0x0B00	Garbage (reallocated)
0x0B01	Allocated (ref. R_1)
0x0B02	Allocated (old 0x0B00, ref. R_0)
0x0B03	Allocated (old 0x0B00)
0x0B04	Allocated (old 0x0B00)
[...]	Free

Table 4.5: State of the heap after the execution of the *realloc* instruction of line 7 in Example 4.7

4.4.3 Memory Maps

In the previous section we show how memory map instructions may cause unexpected behaviors to the memory accesses that *LOAD* and *STORE* instruction perform. In this section we show that inconsistencies do not happen only due to memory accesses over garbaged or remapped memory addresses.

Example 4.9 shows an example of another inconsistency that two consecutive memory map instructions may cause. Let us focus on the execution of such code. Table 4.6 shows the status of the heap after we execute the $free$ instruction of line 5, and Table 4.7 shows the status of the heap after we execute of the *realloc* operation at line 7. As we can note, the $free$ instruction marks as garbage the memory location that R_1 addresses. The *realloc* operation expands the memory block that R_0 addresses, which now includes also the address that R_1 contains. Let us suppose that now, after we execute the *realloc* of line 7, a power failure happen due to a low energy buffer. When there is enough energy to restart the execution, we restore the checkpoint, and the program resumes from the $free$ instruction of line 5. The re-execution of the $free$ operation tries to de-allocate the memory block containing the memory cell of address $0xB01$. Such memory cell is now part of the memory block that R_0 addresses, since the *realloc* operation of line 7 changed the heap state during the previous execution. As consequence, the $free$ operation marks as garbage the entire memory block that R_0 addresses. We not only loose the information we computed and stored in such memory block, but we also may experience a crash. In fact, when we execute the *realloc* instruction of line 7, it will likely fail since the target memory location is marked as garbage.

A similar problem happens if a power failure causes the re-execution of a *realloc* operation. Let us consider the execution of Example 4.10. We run the code, until we reach the *realloc* instruction of line 5, that alters the heap state into the one that Table 4.5 shows. As we can see, it moves the memory block at the end of the heap, and thus it marks the address

```
1  R0 = malloc(1)
2  R1 = malloc(1)
3  [ . . . ]
4  CHECKPOINT
5  free(R1)
6  [ . . . ]
7  R0 = realloc(R0, 3)
8  [ . . . ]
```

Example 4.9: *Memory Map Inconsistency* due to a free-realloc sequence.

Address	Status
0x0B00	Allocated (ref. R_0)
0x0B01	Garbage (freed)
0x0B02	Free
[...]	Free

Table 4.6: Heap status after the execution of $free$ at line 5 of Example 4.9

Address	Status
0x0B00	Allocated (ref. R_1)
0x0B01	Allocated (remap of 0x0B00, ref. R_0)
0x0B02	Allocated (remap of 0x0B00)
0x0B03	Free
[...]	Free

Table 4.7: Heap status after the execution of *realloc* at line 7 of Example 4.9

$0x0B00$ as garbage. Let us now suppose that a shutdown happens due to a low energy buffer. When there is enough energy to restart the computation, we restore the checkpoint and then the execution resumes from the *realloc* instruction of line 5. The re-execution of such *realloc* fails and leads to an inconsistency, since it tries to enlarge a garbaged memory location. This happens because the previous execution of the *realloc* changed the heap state. We must note that if a *realloc* does not alter the heap state, its re-execution can not cause an inconsistency.

Let us instead suppose that we do not experience a power failure after the execution of the *realloc* operation of line 5, and thus we are able to execute the *malloc* instruction of line 7. Since the heap is managed for avoiding fragmentation, the *malloc* request allocates a new memory block inside address $0x0B00$, that the previous *realloc* garbaged. Table 4.8 shows the resulting heap state. Let us now suppose that a shutdown happens due to a low energy buffer. When there is enough energy to restart the computation, we restore the checkpoint and then the execution resumes from the *realloc* instruction of line 5. Such *realloc* moves the memory block present at the address $0x0B00$, which is the one the *malloc* allocated during the previous execution. As consequence, the *realloc* targets a wrong memory block, and thus all the results produced by the previous instruction are lost. Due to this problem, all the future instructions that need such data will produce wrong results. Moreover, we also experience a memory leak, that

```
1  R0 = malloc(1)
2  R1 = malloc(1)
3  [ . . . ]
4  CHECKPOINT
5  R0 = realloc(R0, 3)
6  [ . . . ]
7  R2 = malloc(1)
8  [ . . . ]
```

Example 4.10: *Memory Map Inconsistency* due to a realloc-malloc.

```
1  R0 = malloc(1)
2  R1 = malloc(1)
3  [ . . . ]
4  CHECKPOINT
5  R0 = realloc(R0, 3)
6  [ . . . ]
7  R1 = realloc(R1, 2)
8  [ . . . ]
```

Example 4.11: *Memory Map Inconsistency* due to a realloc-realloc.

Address	**Status**
0x0B00	Allocated (ref. R_2)
0x0B01	Allocated (ref. R_1)
0x0B02	Allocated (old 0x0B00, ref. R_0)
0x0B03	Allocated (old 0x0B00)
0x0B04	Allocated (old 0x0B00)
[...]	Free

Table 4.8: Heap status after the execution of the *malloc* instruction at line 7 of Example 4.10.

eventually may cause the inability of allocating new memory blocks.

A similar problem happens also if we have two consecutive *realloc* instructions, as we show in Example 4.11. In such example, at line 7 there is a *realloc* operation instead of a *malloc*. A power failure happening after the execution of the *realloc* operation of line 7 causes the same problem we described before. In such case, we not only lose the content of memory block that R_0 addresses, but also the one of R_1.

Considering the different behaviors we describe in the above examples, we can say that the re-execution of a *realloc* likely causes an inconsistency. The effects of such unwanted behavior depends on the instructions that we execute before a power failure.

4.4.4 Inconsistencies for Heap in NVM

When we allocate the heap into NVM, all the changes to its state are preserved across power failures, even if they are not included in the state that a checkpoint saves. As consequence, all the changes to the heap that happens after a checkpoint makes the heap state no longer synchronized with the state that such checkpoint saves. Resuming the execution after a power failure restores the checkpoint, and if the heap state is not synchronized with the restored state, we experience an unexpected behavior that we identify as **Memory Map Inconsistency**.

We define a **Memory Map Inconsistency** as an inconsistency which happens whenever there is an ordered sequence of instructions $I_1, ..., I_n$ such that:

1. I_1 is an operation that changes the position or the status of the memory block x

2. I_n is an operation that causes the association of a different memory block where x was

3. x is a heap memory block inside NVM

4. no CHECKPOINT exists in the interval $I_1, ..., I_n$

In other words, this inconsistency may happen whenever there are two memory map operations not separated by a checkpoint.

This type of inconsistency might be removed with the insertion of a checkpoint after the first memory map operation, but the re-execution of instructions that make changes to the heap may lead to an inconsistency even in presence of such checkpoints. For example, the re-execution of a *realloc* causes a **Memory Map Inconsistency**, provided that its first execution changes the location of the target memory block.

Moreover, all the instructions that manipulate the heap state, such as $free$ and *realloc*, are not atomic. Shutdowns that happen in the middle of the execution of such instructions leave the heap state partially changed, and the computation will resume with an inconsistent state. From an inconsistency standpoint, the effects of such power failures are similar to the ones that power resets happening after the execution of such instructions cause. Using a transactional memory controller [24] ensures the atomicity of such instructions, but it does not offer other guarantee about the consistency of the heap state.

Another element we must consider is that the re-execution of a memory allocation certainly leads to a memory leak due to multiple blocks allocated but not used. This scenario may cause runtime failures for not having enough memory.

For these reasons, we should prefer not allocating the heap into NVM. If we require non-volatility for data structures that are present in the heap, we should prefer using an approach that allocates such data structures in the stack or in the global variable sections. This leads us losing the possibility of dynamically allocating memory elements, but it reduces unwanted behaviors that dynamically managing the heap introduces.

Moreover, a *Memory Map Inconsistency* may also cause *Data Access Inconsistencies*. In fact, we say that a **Data Access Inconsistency** can happen as consequence of a *Memory Map Inconsistency* whenever there is an ordered sequence of instructions $I_1, ..., I_n$ such that:

1. I_1 is an operation that accesses the memory block x
2. I_n is an operation that changes the position or the status of the memory block x
3. x is an heap memory block inside NVM
4. no CHECKPOINT exists in the interval $I_1, ..., I_n$

The dynamic management of the heap introduces new elements that may cause a *Data Access Inconsistency*. The analysis we describe in Section 4.2 remains valid and already covers it. For this reason, we can fix it as we did for *Data Access Inconsistencies*, that is with a checkpoint after the data access operation I_1.

4.5 Analyzing Memory Inconsistencies

In this section we describe how we can find the presence of the inconsistencies we described in the previous sections, and how we can analyze the effects that they may cause to the behavior of the program. Inconsistencies happen whenever a power reset causes the re-execution of an instruction that accesses the value produced by a *future* operation. Being able to identify their presence and analyzing their effects permits us not only to understand the effects they may cause, but also to avoid them.

4.5.1 Sequential-equivalence Algorithm

The presence of an inconsistency makes an intermittent execution producing results that differs from the ones of an equivalent sequential execution of the same code. In fact, a consistent intermittent computation should produce the same results that an equivalent sequential execution produces.

We can exploit such notion for verifying if our program presents an inconsistency, and for analyzing the effects that it causes. Let us consider the execution of Example 4.1, which we know has a *Data Access Inconsistency*. When we execute the checkpoint of line 2, we also save a snapshot of the

state, that includes the value 7 for the variable a. We continue the execution, and we reach the *STORE* instruction of line 5, which alters the value of variable a to 3. Let us now suppose that we do not have enough energy to continue, and thus a shutdown happens. When there is enough energy to restart the computation, we restore the checkpoint. If we now compare the runtime state with the snapshot we saved, we can see that variable a has now a value of 3, but in the snapshot it has a value of 7. As consequence, we found an inconsistency. If we continue the execution, we are able to verify what effects it causes in the program behavior.

The described example shows the behavior of Algorithm 1. It specifies how to analyze the presence of inconsistencies with a **static checkpoint mechanism**, in which checkpoints are statically placed inside the code. The algorithm shows us how to analyze the execution of the instructions that are placed after a checkpoint.

We run the program and when we reach a checkpoint, we execute the operations that Algorithm 1 specifies. As first operation, we save a snapshot of the entire runtime state. Such snapshot provide us a consistent memory state, since we save it alongside with the checkpoint. We can use such snapshot for identifying the presence of inconsistencies. Then, we execute the code and for obtaining a complete coverage of all the possible execution scenarios, we simulate a power failure after the execution of every line of code, until we reach the next checkpoint. Whenever we restore a checkpoint, we compare the obtained runtime state with the one saved in the snapshot, and if there is a discrepancy, we find an inconsistency.

In this way, we are able to analyze if an inconsistency may happen with respect to the current checkpoint positions. As consequence, we require verifying n different intermittent executions, with n equal to the number of instructions between the two checkpoints. In fact, testing a power outage in a specific point requires us to run all the previous instructions sequentially, and then to generate a shutdown.

We repeat this process until we cover all the checkpoints. If we do not find any inconsistency, then our program is consistent with respect to the checkpoint placement it has.

If we want to analyze a **dynamic checkpoint mechanism**, we no longer have the checkpoints placed statically inside the code, since they are saved during runtime when the energy buffer is low enough. With a *dynamic checkpoint mechanism*, whenever we save a checkpoint, the execution can either pause or continue until there is no energy remaining. As we stated in Section 2.3, if the execution does not continue after we take a checkpoint, no inconsistency can happen. In fact, since we do not execute any operation, the state can not be modified by future operations, and thus it will always be synchronized with respect to the data that the checkpoint saves. Instead, if the execution continues after we take a checkpoint, we must verify the instructions we run, so to analyze the presence of inconsistencies.

Algorithm 1 Static sequential-equivalence algorithm to be applied after we take a checkpoint.

```
1: Save a complete snapshot of the environment memories.
2:
3: // Simulate intermittent execution until next checkpoint is reached
4: while current instruction ≠ CHECKPOINT do
5:     run current instruction
6:     target_instruction ← next instruction
7:     reset machine state
8:     restore saved checkpoint
9:
10:    if current state ≠ snapshot then
11:        return inconsistency found
12:    else
13:        // return to the operation done before state reset
14:        while current instruction ≠ target_instruction do
15:            run current instruction
```

Since checkpoints can happen in arbitrary positions inside the code, we can not apply the same testing algorithm we used for a *static checkpoint mechanism* Whenever we take a checkpoint, the number of operation we can perform after it is limited by the remaining portion of the energy buffer, and by the possibility of the energy source to maintain such level. If we perform a checkpoint, and then the energy source maintains the level of the energy buffer constant, we will not perform another checkpoint, but the number of executed instructions increments.

To cover all the possible scenarios, we should consider a checkpoint happening at every line of code, since they can happen at any moment during the execution. For each possible checkpoint, we should run a test similar to the one we perform for a *static checkpoint mechanism*. As consequence, we should generate a shutdown at every line after the one we considered for the checkpoint. Such test case has a significant complexity, that is $O(n^3)$, with n equal to the total number of lines in the code. Moreover, in real-case scenarios, it is unlikely that we execute a considerable amount of instructions without triggering a new checkpoint. In fact, a checkpoint is triggered whenever the level of the energy buffer goes below a certain threshold. If we execute a substantial number of instructions after the checkpoint, the energy source must have refilled our energy buffer, and thus a subsequent checkpoint will happen. Instead, if the energy source does not refill our buffer, the number of instructions we can execute is limited by the energy present in our buffer.

For these reasons, we model an **Execution Depth** parameter, that specifies the number of operations we can execute after a checkpoint, before

Algorithm 2 Dynamic sequential-equivalence algorithm to be applied after we take a checkpoint.

```
1: Save a complete snapshot of the environment memories.
2: executed_instructions ← 0
3:
4: // Simulate intermittent execution for execution_depth instructions
5: while executed_instructions < execution_depth do
6:     run current instruction
7:     target_instruction ← next instruction
8:     executed_instructions ← executed_instructions + 1
9:     reset machine state
10:    restore saved checkpoint
11:
12:    if current state ≠ snapshot then
13:        return inconsistency found
14:    else
15:        // return to the operation done before state reset
16:        while current instruction ≠ target_instruction do
17:            run current instruction
```

experiencing a power failure. It depends on the energy source, the energy buffer capacity, and on the architecture we use, since those are the factors which affect energy usage and availability. If the dynamic checkpoint mechanism pauses the execution after it takes a checkpoint, the *Execution Depth* is equal to *0*.

Once we provide such parameter, we can use it for analyzing the presence of inconsistencies in a dynamic scenario. As for the static case, we must generate n different shutdowns, but in this case n is equal to the *Execution Depth* parameter.

Algorithm 2 shows how we can analyze the presence of inconsistencies after we take a checkpoint in a dynamic execution scenario. For having a complete coverage of all the possible execution scenarios, we must consider a checkpoint to happen at every line of code, and we must consider shutdowns to happen at every line of code within the specified execution depth.

For using such algorithm, we start running our program. After the first instruction, we generate a checkpoint, and we apply Algorithm 2. When it finishes, we restore the snapshot that the algorithm saved, so to obtain a consistent memory state. Then, we repeat the entire analysis all over again for the next instruction, until we reach the end of the program. If we do not find any inconsistency, then our program is consistent with respect to a dynamic checkpoint mechanism and the considered execution depth.

Static and Dynamic Equivalence. We can note that we can test a static checkpoint mechanism as a particular case of a dynamic one, in which check-

```
1  STORE a, 7
2  CHECKPOINT
3  STORE a, 5
4  LOAD a, R0
5  STORE a, 3
6  [...]
```

Example 4.12: Data access in NVM wrongly detected as an inconsistency.

points are placed in fixed positions and the *execution depth* is fixed inside the code, and corresponds to the distance between checkpoints. This property permits us testing all the possible static checkpoint placements without having to select a specific checkpoint mechanism. For obtaining so, we can consider a generic dynamic checkpoint mechanism which continues the execution, and we set the *execution depth* equal to the estimated maximum distance between two checkpoints. The analysis will analyze a checkpoint at any position inside the code, and it will also test all the possible reset points within the execution depth range, that is equal to the distance from the next checkpoint if a static checkpoint mechanism would be used instead. The result of such analysis comprehend all the possible inconsistencies present in the program, since it analyzes each possible checkpoint placement.

The *Sequential-Equivalence Algorithm* has a very high running time, especially in dynamic cases. It is not able to identify false-positive inconsistencies, that are inconsistencies for which the content of the memory cell is not changed. The value that instructions write in a memory cell may depend on the input data, and recognizing false-positive inconsistencies may permit us knowing where critical parts of the program are, without the need of running multiple tests with different input data.

Moreover, this algorithm has some problems in recognizing if we must classify a state inconsistent. Let us consider Example 4.12, with a representing the memory location of a variable in NVM. We run the Static Sequential-Equivalence Algorithm on the checkpoint of line 2. When we generate a shutdown after the operation of line 3, we detect an inconsistent state. In fact, the runtime value of a is 5, and it is different from the one present in the snapshot, that is 7. The only instruction that can access a is the LOAD operation of line 4, but it always reads the value that the STORE instruction ot line 3 produces, independently of where a power failure happens. This means that the LOAD instruction can not read an inconsistent value, since the instruction of line 3 always overwrites the variable a with a consistent value, and thus the inconsistency that the algorithm detects is not a real one.

For these reasons, we must modify this algorithm as we describe in the following sections, so to account for this problem and for optimizing its performance.

4.5.2 Data Access and Activation Record

As we stated in Section 4.2, a *Data Access Inconsistency* can happen only between a pair of *LOAD* and *STORE* operations that access the same memory location inside the NVM. This means that:

- During testing, the generation of a shutdown after an instruction that is not part of such pair is useless, since it will certainly lead to a consistent state.
- In a dynamic scenario, generating a checkpoint after a *LOAD* operation can not lead to an inconsistency, since the eventually inconsistent value will not be re-loaded from the NVM. Thus, to optimize the number of tested checkpoints, we should generate them only before *LOAD* instructions that reads data from NVM.

For applying such optimizations, and thus generating checkpoints and shutdowns only on relevant positions, we require two elements:

1. A **Logical Clock** which we increase after the execution of every instruction, and thus makes possible to establish an ordering of events. Every time we perform a checkpoint, we also save with it the current value of the logical clock, so to restore it alongside the checkpoint. This makes us able to verify if a future operation produces a value that a memory location contains.
2. A **Lookup Table**, which is a data structure containing pairs of tuples $(address, write\_clock)$. It keeps track of the addresses we access in NVM, and the time at which we alter their content, in terms of the logical clock.

Address	Write Clock	Info
a	3	STORE of line 3

Table 4.9: Lookup Table of Example 4.12, after the execution of the instruction at line 3.

Address	Write Clock	Info
a	5	STORE of line 5

Table 4.10: Lookup Table of Example 4.12, after the execution of the instruction at line 5.

Algorithm 3 Optimization of the dynamic sequential-equivalence algorithm to be applied after we take a checkpoint.

```
1: Save a complete snapshot of the environment memories.
2: Save current clock
3: executed_instructions ← 0
4: lookup_table ← {}
5:
6: // Simulate intermittent execution for execution_depth instructions
7: while executed_instructions < execution_depth do
8:     // Verify for inconsistency
9:     if current instruction is a MEMORY LOAD then
10:        if address ∈ lookup_table then
11:            if write_clock > current_clock then
12:                if memory_value ≠ snapshoted_value then
13:                    return inconsistency found
14:        else
15:            // Initialize lookup table's record for loaded address
16:            lookup_table ← (address, 0)
17:
18:    run current instruction
19:    increment current_clock
20:
21:    // Verify if a shutdown is needed
22:    if executed instruction is a MEMORY STORE then
23:        if address ∈ lookup_table then
24:            // Update lookup table's record for target address
25:            lookup_table[address] ← current_clock
26:
27:            // Generate a shutdown if this store has not been tested, yet
28:            if executed instruction not tested then
29:                reset machine state
30:                restore saved checkpoint
31:                restore saved clock
32:                continue
33:
34:    executed_instructions ← executed_instructions + 1
```

Let us consider Example 4.12, and let us analyze it using the Sequential-Equivalence Algorithm with the optimization we described. The initial state of the lookup table is empty, and the initial value of the logical clock is 0. We execute the first instruction, and we increase the logical clock to 1. Such instruction alters the content of variable a, we insert a pair $(a, 1)$ into the lookup table. As next operation, we execute the checkpoint, and

thus we save a snapshot of the runtime state. Then, we increase the logical clock to 2, and we save it with the state that the checkpoint saved. Now we execute the STORE instruction of line 3, which alters the content of variable a. We increment the logical clock to 3, and then we update the record in lookup table associated to variable a, as Table 4.9 shows. We continue the execution, and we reach the instruction at line 5, which also alters the content of variable a. We increment the logical clock to 5, and then we update the lookup table, as Table 4.10 shows. Then, we generate a shutdown, since the current operation alters the value that the LOAD instruction of line 4 reads. We restore the checkpoint, alongside the logical clock, which now has a value of 2. We resume the execution from the STORE instruction of line 3, and we execute it. Now, we update the logical clock to 3, and we also update the content of the lookup table, since such instruction alters the content of variable a. Table 4.9 shows the new state of the lookup table. Now, we should run the LOAD instruction of line 4. Since it accesses a memory location that has a record associated into the lookup table, we verify if a *future* operation has produced the value that we are going to read. As we can see in Table 4.9, the *write clock* of variable a is 3 and the current clock is 4. This means that no future instruction has modified the value of a, and thus it no inconsistency can happen. The non-optimized version of the Sequential-Equivalence Algorithm would consider such access to be inconsistent.

Algorithm 3 shows the optimized version of the Sequential-Equivalence Algorithm for the dynamic case. We can easily adapt it to account for a static scenario, by replacing the cycle condition. In this version of the algorithm, we significantly decreased both the number of shutdowns we require for an exhaustive analysis and the number of checkpoint we require in the dynamic scenario. Moreover, we no longer have the problem of consistent states recognized as inconsistent.

The main aspects of Algorithm 3 are:

- To know which *STORE* instruction requires a shutdown, each time a *LOAD* operation happens in NVM, it tracks the address in the *lookup table*. When it executes a *STORE* instruction for which the target address is present in the *lookup table*, the algorithm updates *write_clock* of the target address in the *lookup table* with the current clock. Then, it generates a shutdown.

- To know if the re-execution of a *LOAD* operation may generate a *data access inconsistency*, each time an instruction accesses a memory location, the algorithm verifies the record of the lookup table that corresponds to the memory location. If the *write_clock* is higher than the *current clock*, the *LOAD* instruction may access an inconsistent value, since a future operation produced it. In such case, the algorithm compares the current value of the memory location with the

one present in the snapshot. If they differ, it returns a *data access inconsistency*, otherwise we encountered a false-positive.

Moreover, as we stated in Section 4.3, we can consider an *Activation Record Inconsistency* as a specific case of a data access inconsistency, in which a future function call overwrites the elements in NVM that compose the activation record of the current function. For this reason, Algorithm 3 is also able to recognize activation record inconsistencies, and the only thing we must consider is the different type of instructions which can access the NVM.

For demonstrating this statement, let us analyze Example 4.13 in a dynamic execution scenario. The first instruction which reads from the NVM is the *POP* of line 12, and thus we generate a checkpoint and a snapshot

```
1  call F1
2      # Function Prologue
3      PUSH PC
4      PUSH BP
5      MOV SP, BP
6
7      # Function Code
8      [...]
9
10     # Function Epilogue
11     MOV BP, SP
12     POP BP
13     POP PC
14     RET
15 [...]
```

```
16 call F1
17     # Function Prologue
18     PUSH PC
19     PUSH BP
20     MOV SP, BP
21
22     # Function Code
23     [...]
24
25     # Function Epilogue
26     MOV BP, SP
27     POP BP
28     POP PC
29     RET
30     [...]
```

Example 4.13: Code of two subsequent function calls.

Address	Write Clock	Info
0xF010	0	POP BP of line 12
0xF011	0	POP PC of line 13

Table 4.11: Lookup Table of Example 4.13, after the execution of line 13.

Address	Write Clock	Info
0xF010	0	POP BP of line 12
0xF011	**10**	PUSH PC of **line 18**

Table 4.12: Lookup Table of Example 4.13, after the execution of line 18.

before executing it. Since we do not generate any checkpoint before this one, the lookup table is still empty. From now on, the execution starts as Algorithm 3 specifies, and we increase the logical clock at the execution of each instruction. Once we execute the *POP* instruction of line 12, we update the logical clock to 6. Since such instruction reads from the NVM, we insert into the lookup table a record containing the target address of the read, and we set the associated *write_clock* to 0. We require such operation for keeping track of which memory addresses we accessed, so that we are then able to know after which memory write instructions we must generate a shutdown. We perform a similar action for the instruction of line 13, and the lookup table assumes the content that Table 4.11 shows. We continue the execution, until we reach the instruction at line 18. Such operation writes the NVM, and thus we update in *lookup table* the *write_clock* of the target address with the current value logical clock, that is 10. Table 4.12 shows the state of the lookup table after such operation. Moreover, the lookup table already contains a record for the target address that the current operation writes. This means that a previous instruction may read the content of the memory cell we write, and thus we generate a shutdown. As next operation, we restore the checkpoint, and we bring back the logical clock to 5. Now we resume the execution from the *POP* instruction of line 12, which accesses the NVM. We access the lookup table, and we do not find any inconsistency. We perform the same operation for the *POP* instruction of line 13, which instead reads an inconsistent value. In fact, if we look the lookup table, we notice that the memory cell it reads was written at clock 10, but the current clock is 6. For this reason, we compare the current value of the memory cell with the one present in the snapshot, and they differ. This discrepancy tells us that the *POP* instruction of line 13 reads a wrong return address, that is the one that the future *PUSH* instruction of line 18 wrote.

As we demonstrated, Algorithm 3 has already the tools for identifying activation record inconsistencies. Depending on the Instruction Set Architecture, it may be possible that other instructions apart from *LOAD* and *STORE* are able to interact with the NVM, as happen for the *POP* and *PUSH* instructions in our previous example. As consequence, for recognizing activation record inconsistencies, it must apply to all the memory reading instructions all the actions that it performs for the *LOAD* instruction. The same applies to the actions for the *STORE* instruction, that it must apply to all the memory writing instructions.

4.5.3 Memory Map Inconsistencies

As we stated in Section 4.4, a Memory Map Inconsistency can happen only in two cases:

1. If two subsequent memory map instructions modify the same memory block.

2. If a memory map instruction modifies a memory block that a previous instruction accesses.

In such scenario, the instruction potentially causing an inconsistency is always the one which alters the heap state. For this reason, we can alter Algorithm 3 so to account also for *Memory Map Inconsistencies*:

- We must generate a shutdown after each memory map instruction, so to analyze the effects that it has on the program re-execution.

- We must use another lookup table called **Memory Map Table**, so to track the changes in the heap configuration. It associates to each address the clock in which its mapping has changed, and the old address from which the content of the memory is moved. We must verify the *memory map table* before executing any memory access.

- In a dynamic checkpoint scenario, we must generate a checkpoint before any memory access (i.e., *LOAD* and *STORE*), and before each instruction that changes the heap configuration. In this way, we cover the checkpoint positions that may present a *memory map inconsistency*.

Let us now analyze Example 4.14 using Algorithm 3 with the set of the new optimizations, and let us focus only on the *memory map table*. We start the execution, and we run the *malloc* operation of line 1. We increase the logical clock to 1, and we insert a record inside the *memory map table*, since the instruction changed the heap configuration. We continue the execution, and we perform the same actions for the *malloc* instruction of line 2. Table 4.14 shows the content of the *memory map table* after such operations. Now we execute the checkpoint of line 3, and thus we also save

```
1  R_0 = malloc(1)
2  R_1 = malloc(1)
3  CHECKPOINT
4  STORE R_0 , 7
5  R_0 = realloc(R_0, 2)
6  [ . . . ]
```

Example 4.14: Example of a Memory Map Inconsistency.

Address	Status
0x0A00	Allocated (ref. R_0)
0x0A01	Allocated (ref. R_1)
0x0A02	Free
[...]	Free

(a) Before the execution of the *realloc* instruction of line 5.

Address	Status
0x0A00	Garbage (reallocated)
0x0A01	Allocated (ref. R_1)
0x0A02	Allocated (ref. R_0)
0x0A03	Allocated (ref. R_0)
0x0A04	Free
[...]	Free

(b) After the execution of the *realloc* instruction of line 5.

Table 4.13: Status of the heap in Example 4.14.

Address	Map Clock	Old Address	Info
0x0A00	1	0x0A00	*malloc* of line 1
0x0A01	2	0x0A01	*malloc* of line 2

Table 4.14: Memory Map Table of Example 4.14, before the execution of the *realloc* instruction at line 5.

Address	Map Clock	Old Address	Info
0x0A00	**5**	**Null**	*realloc* of line 5
0x0A01	2	0x0A01	*malloc* of line 2
0x0A02	**5**	**0x0A02**	*realloc* of line 5

Table 4.15: Memory Map Table of Example 4.14, after the execution of the *realloc* instruction at line 5.

a snapshot of the runtime state. As next operation, we execute the *STORE* instruction of line 4, and we increase the logical clock to 4. Table 4.13a shows the current state of the heap. Now we execute the *realloc* instruction of line 5, that relocates the memory block that R_0 addresses to another location, since there is not enough space to extend it in place. Table 4.13b shows the new state of the heap. As next operation, we increase the logical clock to 5, and since the operation changed the heap state, we also update the *memory map table* accordingly. Table 4.15 shows the new content of the *memory map table*. We just executed an instruction that changed the configuration of the heap, and thus we generate a shutdown. Now we restore the checkpoint, we revert the logical clock back to 3, and we resume the execution from the *STORE* instruction of line 4. We increase the logical clock to 4, and since we are going to execute a memory access, we verify the *memory map table*. Our instruction accesses the memory cell referred by R_0, that has address $0x0A00$. The corresponding record in the *memory map table* has a map clock of 5, but the current clock is 4. For this reason, we verify the old address of the memory cell. Since it differs from the current one, we

can state that the content of such cell was moved or marked as garbage by a future operation, and thus this memory access represents a *memory map inconsistency*.

Algorithm 4 shows the version of Algorithm 3 modified with the set of the optimization that we described.

The main aspects of such changes are:

- Each time it executes a memory map function, it updates the *memory map table* accordingly:
  - When it executes a *malloc* or *calloc*, it inserts a new triple (*address*, *current_clock*, *current_address*).
  - When it executes a $free$, it updates the *map_clock* of the involved memory cells, and removes their old address.
  - We can se a *realloc* as the combination of a $free$ and a *malloc*. For this reason, when it executes a *realloc*, it performs both the two above actions. It updates the record associated to the source memory cells with the new *clock*, and removes the associated address. Then, it inserts a new triple (*address*, *current_clock*, *old_address*) for each new allocated memory cell.
- For verifying if an alteration to the heap state leads to an inconsistency, it verifies the *memory map table* before executing any *LOAD* or *STORE* operation that targets the heap. If the map clock of the cell is higher than the current clock, then a future operation changed the state of the memory cell. For verifying the presence of an inconsistency, it compares the current address of the cell with the old address contained in the *memory map table*. If they differ, the cell was moved by a future operation, and thus it finds a *memory map inconsistency*.
- Every time it executes an operation that changes the configuration of the heap, it generates a shutdown, so to verify if the re-execution of the program may produce a *memory map inconsistency*.
- In a dynamic checkpoint scenario, it analyzes a checkpoint before each *STORE* operation that addresses the heap, so to verify if its re-execution causes an unwanted behavior due to a future alteration of the heap configuration.

  It also analyzes a checkpoint before each operation that alters the heap configuration, such as *realloc* and $free$. Their re-execution may lead to an inconsistency, especially if there exists a subsequent operation that alters the heap configuration.

  Note that it does not analyze a checkpoint before *malloc* or *calloc* operations, since their re-execution do not generate any inconsistency.

Algorithm 4 Final optimization of the dynamic sequential-equivalence algorithm to be applied after a checkpoint is taken.

```
1: Save a complete snapshot of the environment memories.
2: Save current_clock
3: executed_instructions ← 0
4: lookup_table ← {}
5: memory_map_table ← {}
6: do_reset ← False
7:
8: // Simulate intermittent execution for execution_depth instructions
9: while executed_instructions < execution_depth do
10:     // Verify for data access and memory map inconsistencies
11:     if current instruction is a MEMORY LOAD then
12:         if address ∈ lookup_table or address ∈ memory_map_table then
13:             if write_clock > current_clock or map_clock > current_clock
    then
14:                 if memory_value ≠ snapshotted_value then
15:                     return inconsistency found
16:         else
17:             // Initialize lookup table's record for loaded address
18:             lookup_table ← (address, 0)
19:     // Verify if a memory map inconsistency is present
20:     if executed instruction is a MEMORY STORE then
21:         if address ∈ memory_map_table then
22:             if map_clock > current_clock then
23:                 if current_address ≠ old_address then
24:                     return inconsistency found
25:
26:     run current instruction
27:     increment current_clock
28:
29:     // Verify if a shutdown is required
30:     if executed instruction is a MEMORY STORE then
31:         if address ∈ lookup_table then
32:             // Update lookup table's record for target address
33:             lookup_table[address] ← current_clock
34:             do_reset ← True
35:
```

```
36:     // Account for memory map inconsistencies
37:     if executed instruction is a MEMORY MAPPING then
38:         // Update memory map table
39:         if executed instruction = free() or executed instruction =
    realloc() then
40:             memory_map_table[address] ← (current_clock, Null)
41:
42:         if executed instruction = malloc() or executed instruction =
    calloc() or executed instruction = realloc() then
43:             memory_map_table[address] ← (current_clock, old_address)
44:
45:         do_reset ← True
46:
47:     // Generate a required reset only if it has not been tested, yet
48:     if do_reset = True and executed instruction not tested then
49:         reset machine state
50:         restore saved checkpoint
51:         restore saved clock
52:         do_reset ← False
53:         set executed instruction as tested
54:         continue
55:
56:     executed_instructions ← executed_instructions + 1
```

4.6 Finding Inconsistencies

The analysis technique of Algorithm 4 permits us to identify where inconsistencies may happen in the code. It also permits us to analyze the effects that an inconsistent state has on the behavior that the execution of subsequent instructions produces.

Depending on our requirements, we may be only interested in identifying where inconsistencies happen, without analyzing the effects that they may cause. We generate power resets for reproducing an inconsistent memory state, and for analyzing the consequent behavior of the program that re-executed instructions cause. Since we are not interested in verifying the consequence of inconsistencies, we can limit our analysis in verifying only the instructions executed, and we can omit the generation of power resets and the re-execution of instructions.

Let us consider Example 4.16, that is the machine-code version of Example 4.15. From the knowledge we provide in this chapter, we are able to see

that it contains two *data access inconsistencies*. If a power reset happens after the execution of line 10, the instructions at line 3 and 8 will read an incorrect value when the device resumes the execution.

For verifying only the presence of inconsistencies, we execute the program sequentially until we reach its end, and we keep track of all the instructions executed. We need the target address of each instruction, and the value that it reads or writes into the memory, and such information is available only during runtime.

Let us suppose that the variable a has an initial value of 1 and it is allocated at the memory address $0xFF10$. The $LOAD$ instruction of line 2 accesses the NVM, and thus we keep track of its target address and the value that it reads from the memory, that are respectively $0xFF10$ and 1. We perform the same action when we execute the instruction at line 8. Finally, we execute the instruction at line 10, that is a $STORE$ targeting the NVM. For this reason, we track its target address and the value it writes in the memory, that respectively are $0xFF10$ and 2. Table 4.16 shows the resulting execution trace and the information that we tracked for each instruction.

Now we can analyze the execution trace for finding the pairs of instructions that leads to inconsistencies. The $LOAD$ instruction at line 2 accesses the NVM and has tracked data associated to it. Considering such instruction, we search for a $STORE$ instruction that writes the same memory address $0xFF10$. We find that the $STORE$ instruction at line 10 writes the value 2 at the same memory address of the $LOAD$. Now we need to compare the value that the $STORE$ writes with the one that the $LOAD$ reads. In this case, the $LOAD$ instruction reads 1 and $STORE$ writes 2. Since those values differ, we find an inconsistency. In fact, if a power reset happens after the execution of the $STORE$, the $LOAD$ instruction would read the

```
1 // a in NVM
2 checkpoint();
3 b = a + 3;
4 if(b > 4) {
5       ...
6 }
7 a++;
```

Example 4.15: Example of a data access inconsistency that the instruction at line 7 may cause.

```
1  CHECKPOINT
2  LOAD a, R0
3  ADD R1, R0, 3
4  STORE b, R1
5  BRANCH(R1 <= 4), ENDIF
6  ...
7  ENDIF:
8  LOAD a, R0
9  ADD R1, R0, 1
10 STORE a, R1
```

Example 4.16: Equivalent machine code of Example 4.15.

#	**Instruction Executed**	**Tracked Data**
1	CHECKPOINT	-
2	LOAD	(0xFF10, 1)
3	ADD	-
4	STORE	-
5	BRANCH	-
6	...	-
7	ENDIF:	-
8	LOAD	(0xFF10, 1)
9	ADD	-
10	STORE	(0xFF10, 2)

Table 4.16: Execution trace of Example 4.15, that permits us to identify if and where inconsistencies may happen.

value that such *STORE* produced, leading to an inconsistent memory state. Now we completed the analysis for the *LOAD* instruction of line 2, and we continue searching for an operation that reads the NVM. We find the *LOAD* instruction of line 8, which accesses the memory at the address $0xFF10$. As we did for the previous *LOAD*, we search for a *STORE* that writes into the same memory location. We find the *STORE* instruction of line 10, which writes 2 into the memory cell. Since this value differs from the one that the *LOAD* reads, we find another inconsistency.

We can see the analysis we described as a lighter version of Algorithm 4, in which we do not require any power reset. The data we tracked for the previous example is exactly the one that the lookup table of Algorithm 4 tracks. Moreover, the execution trace has an implicit ordering of events, and thus we do not require a logical clock.

Considering the definitions of inconsistencies that we provide in this chapter, we can say that an inconsistency may happen in a sequence of instructions $I_1, ..., I_n$ if I_n modifies the non-volatile state that I_1 accesses, and no checkpoint exists in such sequence. In fact, a power reset happening after I_n makes I_1 read an inconsistent value with respect to the restored state. We define I_1 as the *consumer* of the inconsistent value, and I_n as its *producer*. Table 4.17 summarizes for each inconsistency the instructions that consumes and produce the inconsistent value. Considering the analysis technique that we described in the previous sections, we verified a checkpoint placement before each *consumer*, and we evaluated the effects of shutdowns happening after each *producer*.

Algorithm 5 formalizes the work flow we previously described for finding the presence of inconsistencies in Example 4.16. It shows the algorithm we can use for finding the presence of inconsistencies, without analyzing the effects that they may cause over the behavior of the program.

	Data Access Inconsistency	Activation Record Inconsistency	Memory Map Inconsistency
Consumer	*load*	*pop*, *ret*	*load*, *store*, *realloc*, *free*
Producer	*store*	*push*, *call*	*malloc*, *realloc*, *free*

Table 4.17: Subset of checkpoint and reset locations.

Firstly, we execute the program sequentially, and for each instruction accessing the NVM we save the associated lookup information:

- **load** and **pop**: we save the target address and the value that it reads from NVM.
- **store** and **push**: we save the target address and the value that it writes on NVM.
- **call**: we decompose the call instruction into the different **push** that it performs. We save the lookup information for each one of them.
- **ret**: we decompose the ret instruction into the different **pop** that it performs. We save the lookup information for each one of them.
- **malloc** or **calloc**: we save the addresses of all the memory cells contained in the memory block it allocates.
- **free**: we save the addresses of all the memory cells contained in the memory block it garbages/deallocates.
- **realloc**: we save the addresses of all the memory cells that this instruction alters, and their content.

Then, we start analyzing the execution trace we produced. Whenever we find an instruction that we identify as a *consumer*, we append it to the consumer queue. Whenever we find an instruction that we identify as a *producer*, we compare its lookup information with the ones of the consumers present in the consumer queue. For each consumer c that is in the *executiondepth* range with the producer p, we verify whether p alters the address that c accesses. In such case, we find an inconsistency that p causes to the access that c performs. As next operation, we remove from the consumer queue all the consumers that are no longer within the *executiondepth* range.

We must note that this analysis works also on instructions that we both classify as consumer and producer. For example, it is able to verify if the re-execution of a *realloc* may cause a *Memory Map Inconsistency*.

Moreover, Algorithm 5 performs the analysis using a dynamic configuration, and we can easily adapt it to work in a static configuration of checkpoints. We only need to alter how we remove consumers from the consumer queue: instead of verifying if consumers are within the *executiondepth* range for removing them from the queue, we have to entirely flush the consumer queue when we reach a checkpoint instruction. Algorithm 6 shows the resulting algorithm that we can use for finding inconsistencies in a static configuration.

Finally, this analysis techniques does not verify the effects that inconsistencies have over the behavior of the program. For this reason, it analyzes a significant lower amount of instructions with respect to Algorithm 4, reducing the instructions it executes to $O(n)$. This results in a lower execution time, but with this algorithm we are not able to identify the effects that inconsistencies have over the program. Depending on our requirements, we can select the algorithm that best suits our needs.

Algorithm 5 Searching inconsistencies in a dynamic configuration

Require: ED: execution depth
Require: *trace*: the execution trace of the program, containing lookup data of instruction

```
1:  // Queue of consumers to be analyzed
2:  consumers_queue ← [ ]
3:  expired ← 0
4:  i ← 0
5:  inconsistencies ← [ ]
6:
7:  while i < len(trace) do
8:      // if the instruction is a consumer, add it to consumer queue
9:      if trace[i] instruction ∈ CONSUMERS then
10:         append i to consumers_queue
11:
12:     // if the instruction is a producer, compare it with consumer queue
13:     if trace[i] instruction ∈ PRODUCERS then
14:         for each c ∈ consumers_queue do
15:             // if distance higher than ED, mark consumer for removing
16:             // else analyze consumer and producer lookup data
17:             if c − ED > i then
18:                 expired ← expired + 1
19:             else
20:                 // we compare the lookup data of trace[i] and trace[c]
21:                 if trace[c] alters trace[i] target cells then
22:                     append (i, c) to inconsistencies
23:
24:         // remove expired consumers
25:         while expired > 0 do
26:             pop from consumers_queue
27:             expired ← expired − 1
28:
29:     i ← i + 1
30:
31: return inconsistencies
```

Algorithm 6 Searching inconsistencies in a static configuration

Require: *trace*: the execution trace of the program, containing lookup data of instruction

```
1: // Queue of consumers to be analyzed
2: consumers_queue ← [ ]
3: i ← 0
4: inconsistencies ← [ ]
5:
6: while i < len(trace) do
7:     // if the instruction is a consumer, add it to consumer queue
8:     if trace[i] instruction ∈ CONSUMERS then
9:         append i to consumers_queue
10:
11:    // if the instruction is a producer, compare it with consumer queue
12:    if trace[i] instruction ∈ PRODUCERS then
13:        for each c ∈ consumers_queue do
14:            // if distance higher than ED, mark consumer for removing
15:            // else analyze consumer and producer lookup data
16:            if trace[i] instruction = CHECKPOINT then
17:                // reset consumer queue
18:                consumers_queue ← [ ]
19:            else
20:                // we compare the lookup data of trace[i] and trace[c]
21:                if trace[c] alters trace[i] target cells then
22:                    append (i, c) to inconsistencies
23:
24:    i ← i + 1
25:
26: return inconsistencies
```

Chapter 5

Environment Interactions and Inconsistencies

5.1 Overview

Transiently Powered Computing is usually applied to devices used as sensors, which can interact with the environment by both sending and receiving data from it. The possibility of interacting with the environment does not cause any new type of inconsistency, but the way in which the checkpoints happen inside the code may lead to unexpected results from the environment sensing standpoint.

Furthermore, the analysis that we describe in Chapter 4 treats inconsistencies only as an unwanted behavior, but we can exploit them for making the program aware of intermittence. This new kind of input opens to new possibilities, such as tracking the number of shutdowns.

In this chapter we analyze both these two cases, and we introduce environment interactions in the inconsistency model described so far. For doing so, we use in the examples of this chapter the assembly language that we described in Section 4.1, with the addition of the instructions that Table 5.1 shows.

We must note that in a dynamic execution scenario it is not possible to analyze the properties that we describe in this chapter, since they are strictly dependent on the positioning of checkpoints.

Instruction	Description
$R_x = input(I_i)$	Reads the value present on the input port I_i, and then stores it into the register R_x.
$output(O_i, R_x)$	Sets the value of the output port O_i to the one that the register R_x contains.

Table 5.1: List of functions permitting environment interactions.

5.2 Environment Inputs

5.2.1 Input Accesses

The intermittence characterizing TPCs not only may introduce memory inconsistencies, but it may also alter the behavior of the program with respect to the data sensed from the environment.

Let us consider Example 5.1, in which a power reset makes the computation resume after the checkpoint at line 1. As consequence, the instruction at line 2 will always be executed before all the usages of the sensed value that R_0 contains. This example describes a **Most Recent** input access model: whenever the execution restarts from a checkpoint, the value of the input is sensed again from the environment, with the effect of using the most possible recent one.

Instead, Example 5.2 shows a different behavior: when a power reset happens, the computation resumes after the checkpoint at line 2. As consequence, the instruction that reads the sensor data is not re-executed, and the value that R_0 contains may not reflect the current environment state. This second example describes a **Saved** input access model: whenever the execution resumes, the value of the input may be contained in the state saved by the checkpoint. In such a case, it is not re-sensed from the environment and the device uses the last sensed value, even if it may not represent the current environment state.

There is no input access model that we can consider correct a priori, since it depends on the application requirements. For this reason, we need to establish and specify which is the input access model that we want on

```
1 CHECKPOINT
2 R0 = input(I1)
3 ADD R1, R0, 3
4 [...]
```

Example 5.1: Access to input I_1 with a *most recent* input access model.

```
1 R0 = input(I1)
2 CHECKPOINT
3 ADD R1, R0, 3
4 [...]
```

Example 5.2: Access to input I_1 with a *saved* input access model.

```
1 [...]
2 R0 = input(I1)
3 ADD R1, R0, 3
4 [...]
```

Example 5.3: Access to input I_1 with a dynamic checkpoint mechanism.

```
1 CHECKPOINT
2 R0 = input(I1)
3 ADD R1, R0, 3
4 [...]
5 CHECKPOINT
6 SUB R7, R1, 3
7 [...]
```

Example 5.4: Access to input I_1 with a *saved* input access model.

each input, so to verify if any interaction behaves unexpectedly. We can say that if the program accesses an input with a wrong input access model, an **Input Access Inconsistency** happens.

The tests for analyzing the input access model associated with the usage of each input is not applicable with dynamic checkpoint mechanisms. Let us consider Example 5.3, that shows a simple input usage with a dynamic checkpoint mechanism. If a checkpoint happens before line 2, the program accesses the input with a *most recent* access model. Instead, if a checkpoint happens before line 3, the program accesses the input with a *saved* access model. In such a scenario, a checkpoint can happen at any moment during the execution, and thus we can not set an input access model for our input.

Testing input access models requires also tracking the propagation of input dependencies among memory locations and registers, so to verify indirect usages of the sensed data. Let us focus on Example 5.1, and let us suppose the we require a *Most Recent* access model for input I_1. Furthermore, let us analyze the input access model of input I_1 without tracking input dependencies. Register R_0 contains the input value, and only the instruction at line 3 accesses it. For this reason, we measure a *Most Recent* access model for the input I_1, and thus we do not find any *Input Access Inconsistency*. Instead, if we track and propagate input dependencies, we can notice that register R_1 depends on input I_1, and that the instruction at line 6 accesses such register. The value that R_1 contains is produced before the checkpoint, and thus we are able to establish that such operation accesses the input I_1 with a *Saved* access model. For this reason, we identify an *Input Access Inconsistency* that we would not be able to recognize if we did not track the propagation of input dependencies.

5.2.2 Testing Input Access Models

For testing input accesses, we must specify the input access model we desire for each input.

Moreover, as we previously stated, we must track the usage of input-dependent data, so to find if such usage represents an *Input Access Inconsistency*. To achieve that, we exploit two data structures:

- A **checkpoint clock**, which indicates the number of checkpoints taken. We can use such parameter to establish an ordering between different checkpoint intervals.
- An **input-dependency table**, which tracks and propagates the usage of input values. It consists in a table that associates to a memory element, a list of pairs $(input, checkpoint\_id)$. Such list contains the inputs used for producing the value contained in the memory element, and for each input it also contains am indication of when the input access happened.

```
1  R0 = input(I1)
2  R1 = input(I2)
3  ADD R2, R1, R0
4  CHECKPOINT
5  [ . . . ]
6  ADD R2, R2, 7
```

Example 5.5: Example of an input access inconsistency. We require a *Saved* access model for I_1, and a *Most Recent* access model for I_2.

```
1  R0 = input(I1)
2  CHECKPOINT
3  R1 = input(I2)
4  ADD R2, R1, R0
5  [ . . . ]
6  ADD R2, R2, 7
```

Example 5.6: A checkpoint placement which fixes the input access inconsistency that we show in Example 5.5.

Let us suppose that we want to analyze Example 5.5, and that we want a *SAVED* access model for input I_1 and a *MOST RECENT* access model for I_2. We initialize the *checkpoint clock* to 0, and we start executing the program. When we execute the instruction at line 1, we note that it directly accesses an input element. For this reason, we insert a pair $(I_1, 0)$ into the *input-dependency table*, in correspondence of the register R_0, that is where the operation stores the input data. We perform a similar operation for the execution of the instruction at line 2. Now we execute the instruction of line 3, that uses as operands R_0 and R_1. Such registers contain input-dependent data, and thus we access their record of the *input-dependency table*. We accessed both the corresponding inputs in this checkpoint interval, and thus no *Input Access Inconsistency* is present. We combine the record of R_0 and R_1, and we store it into the *input-dependency table*, in correspondence of the register R_2. Table 5.2a shows the current state of the *input-dependency table*. Now, we reach the checkpoint of line 4, and thus we increment the *checkpoint clock* to 1. As next operation, we execute the instruction of line 6, which accesses R_2 and thus we access its corresponding record in the *input-dependency table*. As we can see, it uses the values of I_1 and I_2 that were produced in the previous checkpoint interval. Since we require a *MOST RECENT* access model for the input I_2, we find an *Input Access Inconsistency*.

Algorithm 7 formalizes the described work flow, that permits us to analyze the access model of each input. As we demonstrated in the previous example, this test does not require an intermittent execution, since we can analyze access models by looking at the checkpoint interval under which the program accesses inputs.

Since we do not need to simulate power resets, whenever we encounter a checkpoint, we only increment the *checkpoint clock*. Every time the execution of an instruction alters the value of an element, we must update the corresponding record in the input-dependency table, using the lists of the

Element	Dependency List
R_0	(I_1, 0)
R_1	(I_2, 0)
R_2	(I_1, 0) (I_2, 0)

(a) In Example 5.5.

Element	Dependency List
R_0	(I_1, 0)
R_1	(I_2, 1)
R_2	(I_1, 0) (I_2, 1)

(b) In Example 5.6.

Table 5.2: Content of the input-dependency table

inputs corresponding to the operands of the instruction. When an instruction accesses an element, we must analyze the corresponding records in the input-dependency table, so to verify the access model of inputs. If an element of the input-dependency table has a *checkpoint_id* lower with respect to the current value of the *checkpoint clock*, then the program accesses the corresponding input with a *Saved* access model. Otherwise, it accesses the corresponding input with a *Most Recent* access model.

Finally, Example 5.5 has an *Input Access Inconsistency* on the input I_2. We can fix this problem by moving the checkpoint of line 4 before line 2, as we show in Example 5.6. In this way, the program no longer accesses the input I_2 with a *Saved* access model. If we re-execute the analysis that we previously described, we produce the *input-dependency table* that Table 5.2b shows. When we execute the instruction of line 6, we access the record of the *input-dependency table* that corresponds to R_2. During the execution of such instruction, the checkpoint clock is 1, since we incremented it when we executed the checkpoint of line 2. As consequence, we can see that the value we access of input I_2 is produced during the current checkpoint interval. This means that we we access the input I_2 with a *Most Recent* access model, and thus we fixed the inconsistency.

As we can see, for fixing an *Input Access Inconsistency* we have to move a checkpoint or a set of instructions, so to make the input access consistent with respect to our requirements.

Algorithm 7 Sequential execution for testing input access models.

Require: *access_models*, an array specifying the access model of each input.

```
1:  checkpoint_clock ← 0
2:  input_dep_tbl = {}
3:
4:  while program end not reached do
5:      // If checkpoint reached, increment clock and skip execution
6:      if current instruction = CHECKPOINT then
7:          checkpoint_clock = checkpoint_clock + 1
8:          skip to next instruction
9:          continue
10:
11:     if current instruction = input(I_i) then
12:         // On input reading create input dependency table's record
13:         input_dep_tbl[target_element] ← [(I_i, checkpoint_clock)]
14:     else
15:         // Re-initialize target element's record
16:         input_dep_tbl[target_element] ← []
17:
18:         // Propagate operands records to target element record,
19:         // measure access model and verify it.
20:         for each operand ∈ current instruction do
21:             for each record ∈ input_dep_tbl[operand] do
22:                 input_id ← record[0]
23:                 clock_id ← record[1]
24:                 append record to input_dep_tbl[target_element]
25:
26:                 if clock_id < checkpoint_clock then
27:                     access_model = SAVED
28:                 else
29:                     access_model = MOST RECENT
30:
31:                 if access_model ≠ access_models[input_id] then
32:                     return input access inconsistency for input_id
33:
34:     run current instruction
```

5.3 Intermittence as Program Input

5.3.1 Inconsistencies and Inputs

In Chapter 4 we described the three kinds of *memory inconsistencies* and the effects that they have on the behavior of the program. Until now, we treated inconsistencies as a bug that denies the correct execution of the program. For this reason, we focused on maintaining an execution model that produces results as similar as possible to an equivalent sequential execution of the program. Although this lead to predictable and correct results, we are limiting our possibilities: the intermittence of the execution is a characterizing property of TPCs, and we may consider it as an input for our programs.

To better understand how we can include intermittence as input for our program, let us focus on the execution of Example 5.7, in which we allocate the variable a into NVM. Let us suppose that we execute the code without any power interruption until we reach line 5, and then a shutdown happens. When there is enough energy to restart the computation, the program restores the checkpoint of line 2, and the execution resumes from the instruction at line 3, which is re-executed Here a *Data Access Inconsistency* happens: the *LOAD* operation reads 1, that is an inconsistent value for the variable a. In fact, the previous execution of the instruction of line 5 set such value, which differs from the value that variable a had during the checkpoint.

By permitting the presence of such inconsistency, we enable our program to track the number of re-executions. If another shutdown happens, the program resumes with a value of variable a equal to *2*, which corresponds to the number of re-executions.

The presence of *Data Access Inconsistencies* in specific sections of our code makes the programmer able to adapt the behavior of a program in presence of power resets. The number of applications in which we can exploit

```
1 STORE a, 0
2 CHECKPOINT
3 LOAD a, R0
4 ADD R0, R0, 1
5 STORE a, R0
6 [...]
```

Example 5.7: Inconsistency that enables the program keeping track of the number of re-executions.

```
1 a = 0;
2 checkpoint();
3 a = a + 1;
4 [...]
```

Example 5.8: Equivalent C version of Example 5.7.

```
1  STORE a, 0
2  CHECKPOINT
3  LOAD a, R0
4  BRANCH (R0 < 1), END
5  ADD R0, R0, 1
6  STORE a, R0
7  [...]
8  END:
9  CHECKPOINT
10 [...]
```

Example 5.9: Inconsistency which grants that the program executes the code block at line 7 at most once.

```
1 a = 0;
2 checkpoint();
3 if(a < 1) {
4     a = a + 1;
5     // Critical code block
6     [...]
7 }
8 checkpoint();
```

Example 5.10: Equivalent C version of Example 5.9.

such new kind of input is almost unlimited. Considering the possibility of interacting with the environment, this opens to scenarios in which we can signal subsequent power failures, or in which we can perform compensation actions after a certain amount of power resets. We can obtain a similar result by permitting the presence of *Output Inconsistencies*, as we will describe in Section 5.4.

For example, let us consider Example 5.9, that shows a program containing an inconsistency at line 3, that a power reset happening after line 6 causes. Such inconsistency grants the possibility of having the code block of line 7 to be executed at most once. In fact, let us suppose that a power failure happens after the execution of such code block. When the program resumes the execution, it jumps to the checkpoint of line 9, because the branch condition is verified. As consequence, it skips the re-execution of the code block of line 7. Instead, if a power failure happens after we execute line 6, the code block of line 7 is never executed. This execution flow is only possible if we permit the data access inconsistency of the LOAD instruction at line 3. If we move a checkpoint for fixing the inconsistency, the program no longer manifests such execution flow.

Example 5.11 shows another example of an *intermittence-based input*, that in this case makes the program able to execute the code block of line 8 after 3 or more power failures.

In fact, *intermittence-based inputs* makes the program able to track the number of times in which a portion of code is re-executed due to power resets. For achieving such scenario, it is sufficient placing a variable into NVM, and then increment it after a checkpoint. Such variable will contain the number of power resets that caused the re-execution of the checkpoint interval. Then, we can use the NVM variable inside our program for performing the actions we need.

```
1  STORE a, 0
2  CHECKPOINT
3  LOAD a, R0
4  ADD R0, R0, 1
5  STORE a, R0
6  [...]
7  BRANCH (R0 ≤ 3), END
8  [...]
9  END:
10 CHECKPOINT
11 [...]
```

Example 5.11: Inconsistency that enables the program to run the code block of line 8 after 3 or more power failures.

```
1  a = 0;
2  checkpoint();
3  a = a + 1;
4  [...]
5  if(a > 3) {
6      // Code to be executed
           after 3 resets
7      [...]
8  }
9  checkpoint();
10 [...]
```

Example 5.12: Equivalent C version of Example 5.11.

```
1  int v[10000];
2
3  [...]
4  for(i = 0; i < 10000; i++)
5  {
6      [...]
7      v[i] = ...
8  }
9  [...]
```

Example 5.13: Example of loop.

```
1  int v[10000];
2
3  [...]
4  a = 0;
5  checkpoint();
6  for(i = a; i < 10000; i++)
7  {
8      [...]
9      v[i] = ...
10     a = i;
11 }
12 checkpoint();
13 [...]
```

Example 5.14: Inconsistency that permit us to omit the insertion of a checkpoint inside the loop body of Example 5.13.

Moreover, with intermittence-based inputs we are also able to avoid checkpoint overhead, or to reduce the number of checkpoints present in our code. Let us consider Example 5.13, and let us suppose that the code block at line 6 performs a computational-intensive task whose result is stored inside *v[i]* of line 7. Let us suppose that we choose to use MementOS [3] as checkpoint mechanism, with a *loop-latch* strategy. It places a checkpoint at the end of the loop body, resulting in a high checkpoint overhead and energy waste. We can allow the presence of inconsistencies for reducing the checkpoint overhead. Example 5.14 shows the resulting code, in which we allocate a and v into NVM. The variable a is an *intermittence-based input* that keeps track of the loop iteration.

To better understand this solution, let us focus on the execution flow of the code. When we reach the instruction of line 5, we save a checkpoint, and then we enter the loop. We initialize the iteration variable i with the value of our *intermittence-based input* a, that we previously initialized to 0. When we reach the instruction of line 9, we update $v[i]$, and then we increment a. Let us suppose that after 100 iterations a shutdown happens due to a low energy buffer. When there is enough energy to restore the computation, we restore the checkpoint, and the execution resumes from line 6. The iteration variable i is initialized with the value of a, which is 100. As consequence, the execution continues from the iteration where the power failure stopped it. We might consider the first 100 values of the array v to be inconsistent, since they differ from when the checkpoint was taken. Since the variable a grants a precise work flow, we can ignore such inconsistency, since it is a form of *intermittence-based input* that allows us to continue the execution without performing a checkpoint. In this case, we can consider variable a as a *checkpoint* with a low overhead, since it produces a similar effect of a checkpoint.

In the described example we did not use the *intermittence-based input* for tracking the number of re-executions, but instead we exploited it for making us able to keep track of where the iteration stopped in the previous execution of our code.

5.3.2 Testing Intermittence-based Inputs

Testing the correctness of a program that uses *intermittence-based inputs* requires tracking environment interactions and variable values, since those are the elements that a power reset affects. For verifying the correctness of *intermittence-based inputs*, we must analyze the behavior that our program present in the presence of intermittent executions.

Performing this analysis requires us to generate specific cases of intermittent executions that recreate the conditions in which we can verify if our *intermittence-based inputs* behave as expected. Moreover, we also require an *Interaction Table*, which is data structure containing the information that we track about the execution of the program. It associates to each checkpoint a list of the corresponding events that happened after it.

Let us now suppose that we want to analyze the behavior of Example 5.15, in which the output operation at line 7 should be executed at most twice. For verifying such behavior, we can recreate power resets before the execution of line 9, and we should re-execute the program more than two times. Moreover, we are interested in tracking the value of variable a, that is our *intermittence-based input*, and of the output event O_1. We execute the code, and every time we run the output instruction of line 7, we insert a record inside the *Interaction Table*. Moreover, since we are interested in the value of variable a, every time we generate a power reset, we also insert

```
1  STORE a, 0
2  CHECKPOINT
3  LOAD a, R0
4  BRANCH (R0 < 2), END
5  ADD R0, R0, 1
6  STORE a, R0
7  output(O1, 5)
8  END:
9  CHECKPOINT
10 [...]
```

Example 5.15: Output executed at most twice.

```
1 a = 0;
2 checkpoint();
3 if(a < 2) {
4     a = a + 1;
5     output(O1, 5);
6 }
7 checkpoint();
8 [...]
```

Example 5.16: C equivalent version of Example 5.15.

Checkpoint PC	Trace	
	Reset Number	Elements
2	0	$(O_1, 5)$
	0	$(a, 1)$
	1	$(O_1, 5)$
	1	$(a, 2)$
	2	$(a, 2)$

Table 5.3: Content of Interaction Table of Example 5.15 once the execution completed.

a record inside the *Interaction Table*, containing the value of variable a. Table 5.3 shows the resulting *Interaction Table*. As we can see, the program executed the output instruction at line 7 two times: at resets number 0 and 1. Moreover, we can also see that the value of variable a is 1 before the first reset, and becomes *2* before the second one. With this result, we are able to state that the observed behavior corresponds exactly to the one we expect.

In the previous example, we described the work flow of Algorithm 8, which performs the analysis of the program behavior in an intermittent execution scenario. The algorithm executes the program sequentially, and tracks the I/O operations that it executes inside the *execution_trace* data structure, that is the interaction table we previously described. When it reaches a point where it should generate a power reset, it also tracks the variables of our interest, and then it performs such power reset. The algorithm ends the analysis when it reaches the end of the program, and then it returns us the *execution_trace* it produced. Using this result we are able to understand whether our program behaves as expected.

As final note, we must consider that the entire analysis depends on where we choose to generate power resets. There are cases in which we should run

the analysis multiple times with different reset positions, otherwise we do not have a complete coverage of all the possible scenarios. For example, in Example 5.9 we only analyzed a reset after the instruction of line 8. Such analysis told us that the program executes the output function twice, but we are not sure about its behavior if a power reset happens before line 7. For this reason, we should run two different analysis: one with two resets before the instruction at line 7, and one with two resets after line 8. In this way, we are able to verify the *at-most-once* property that we require for the code, with a complete coverage of the possible reset scenarios.

Algorithm 8 Intermittent execution algorithm for testing intermittence-based inputs.

Require: list of where we want to generate power resets.
Require: variables that we want to track.

```
1: execution_trace = {}
2: current_checkpoint ←None
3: reset_no ← 0
4:
5: while program end not reached do
6:     if current instruction = CHECKPOINT then
7:         // If checkpoint reached, keep track of its program counter
8:         // and reset the reset counter.
9:         current_checkpoint ←Program Counter
10:        reset_no ← 0
11:
12:    else if current instruction = input(I_i) then
13:        // If an input instruction is reached, keep track of the
14:        // input port, the read value, the resets counter and the
15:        // program counter of the instruction to be executed.
16:        profiling_info = (I_i, input_value, reset_no, program_counter)
17:        append profiling_info to execution_trace[current_checkpoint]
18:
19:    else if current instruction = output(O_i, R_x) then
20:        // If an output instruction is reached, keep track of the
21:        // output port, the written value, the resets counter and the
22:        // program counter of the instruction to be executed.
23:        profiling_info = (O_i, R_x, reset_no, program_counter)
24:        append profiling_info to execution_trace[current_checkpoint]
25:
26:    run current instruction
27:
28:    // If the user specified to generate a power reset here
29:    // and at the current reset number, generate it.
30:    if reset required after current instruction then
31:        for each variable to be tracked do
32:            profiling_info = (variable_name, variable_value, reset_no)
33:            append profiling_info to execution_trace[current_checkpoint]
34:
35:        reset
36:        restore checkpoint
37:        reset_no ← reset_no + 1
38:
39: return execution_trace
```

5.4 Environment Outputs

5.4.1 Ouput Inconsistencies

In Section 5.2.2 we analyzed the effects that an intermittent execution causes from the environment inputs standpoint. Environment interactions do not consist only in input actions, but they also comprehend output interactions such as sending data to another device, or moving a servo. Output actions do not change the memory state of a device, and instead they change the state of the environment.

The most common application of devices in TPC comprehend their usage as sensors for a wireless network. Their workflow consists in sensing the environment, processing the sensed data, and sending such data to the main node of the network. Let us consider Example 5.17, that represents an application which reads the temperature from two sensors, sums such values, and sends the result to the main node of a wireless sensor network. We are interested in analyzing which are the effects of the intermittent execution over the *output* operation of line 6. Let us focus on the intermittent execution of the code, without considering a specific checkpoint mechanism or memory configuration. We start the execution, and we reach the instruction at line 2, which reads the temperature of the environment from the input port I_1. Then, we execute the next instruction, which performs the same operation using the input port I_2. As next instruction we execute line 4, which sums the temperature values and stores them into the register R_2. The execution continues, and we reach the checkpoint at line 5. We save the state, and

```
1 [ . . . ]
2 R_0 = input(I_1)
3 R_1 = input(I_2)
4 ADD R_2, R_1, R_0
5 CHECKPOINT
6 output(O_1, R_2)
7 [ . . . ]
```

Example 5.17: Example of a common application in TPC, which senses the environment temperature
from the two inputs, sums the sensed values, and finally sends the result to the main node of the system.

```
1 [ . . . ]
2 CHECKPOINT
3 output(O_1, 15)
4 [ . . . ]
```

Example 5.18: Example of a code that moves a servo connected to the output port O_1 by 15° degrees.

then we execute the instruction at line 6, which sends to the main node the value of the register R_2. Let us now suppose that, after the execution of such instruction, a shutdown happen due to a low energy buffer. When there is enough energy to restart the computation, we restore the saved checkpoint and the execution resumes from the instruction at line 6. We execute this instruction for a second time, and thus the main node receive for a second time the same value of the register R_2. Depending on how the main node processes data, this can lead to an error. Let us for example suppose that the main node of the network stores into the database every data it receives, and that such data is then used for performing some analysis. In such a scenario we would have a duplicate value, and the subsequent computation would be incorrect.

This kind of problem does not happen only if the device sends data to another one, and it can also happen if the device is able to modify the environment state. Let us consider Example 5.18, which consists in a fragment of code that moves a servo by 15°, and let us focus on the intermittent execution of such code. Let us assume that the initial position of the servo is 30°. When we reach the checkpoint at line 2, we save the state, and then we continue the execution by running the instruction at line 3. As consequence, the servo is moved by 15°, and its new position is 45°. Let us now suppose there is no enough energy to continue the execution, and thus a shutdown happen due to a low energy buffer. When there is enough energy to restart the computation, we restore the checkpoint and the execution resumes from the instruction at line 3. We move the servo by 15° degrees, and its new position is 60°. From now on, all the actions that the device performs will lead to unexpected results, since it will assume that the position of the servo is 45° degrees, but instead it is at 60°.

In both these two cases, the re-execution of an environment output routine causes the environment state to be different with respect to equivalent sequential execution of the code. This unexpected behavior does not depend on the checkpoint mechanism we use or on the memory sections that we allocate into NVM. In fact, the re-execution of instructions is a direct cause of the shutdowns characterizing TPCs. For this reason, any environment output action can potentially cause an inconsistency of the environment state, since it is inevitable re-executing some instructions. We refer to this kind of inconsistency as **Output Inconsistency**.

5.4.2 Analyzing Output Inconsistencies

Before analyzing where output inconsistencies may happen, we must define the concept of **output idempotency**. We say that an output routine is *idempotent with respect to its re-execution* if we can re-execute it multiple times without changing the environment state. In other words, an idempotent output routine changes the state only the first time it is executed, and

if a power reset happens, its re-execution does not modify the environment state. As consequence, if an idempotent output routine is re-executed, it can not cause any *output inconsistency*.

For example, let us assume that we have an output function which moves the servo to a specified position. In this case, this output routine is idempotent with respect to its re-execution, and thus its re-execution can not cause any *output inconsistency*. In fact, the first execution moves the servo to the target position, but subsequent re-executions that power resets may cause would leave the servo position unchanged.

For verifying the presence of output inconsistencies inside a program, we firstly find if a non-idempotent output routine exist. For doing so, we analytically consider each output routine present in the program without considering its code, and we verify the effects of multiple subsequent re-executions. If the environment state obtained after such re-executions is the same obtained after the first re-execution, then the considered output routine is idempotent. Let us for example consider Example 5.17, which we previously described. We have an output routine which sends the processed data to another device. In this case, multiple re-execution of the output routine will send the same data to the receiving device. If the device is not able to recognize re-transmitted messages, the output routine is non-idempotent, since each re-execution will affect the data processed by the receiving device.

As next step in our analysis, we verify the effects that the re-execution of each non-idempotent output routines has over the environment state inside the program. We require such step since a non-idempotent output routine does not necessarily cause an *output inconsistency* if re-executed. In fact, let us consider Example 5.19, which has a non-idempotent output function. The code measures the current position of the servo, then it calculates the angle required to reach 45°, which is our target position. As next step, the program moves the servo using the result of such operation, and thus after the execution of the instruction at line 5 it will be at 45°, even if it is re-executed. As consequence, the re-execution of the output function does not cause an *output inconsistency*, even if it is non-idempotent.

Testing the effects of the re-executions of non-idempotent output routines is a particular case of testing *intermittence-based inputs*, which we described in Section 5.3. In fact, we only need to generate a power reset after the execution of an output routine. For this reason, we can use a simplified version of Algorithm 8, that Algorithm 9 shows. Such simplified algorithm keeps only track of the output routine executed and generates one power reset after each one of them. Moreover, the algorithm saves also a snapshot every time a checkpoint is taken, and restores it after the second execution of each output routine. In this way, not data inconsistency can happen, and thus no unexpected behavior can happen.

Algorithm 9 Intermittent execution algorithm for testing effects of non-idempotent output routines.

Require: list of the non-idempotent output routines

```
1: execution_trace = {}
2: current_checkpoint ←None
3: reset_no ← {}
4: for each non-idempotent output routine do
5:     reset_no[output_routine] ← 0
6:
7: while program end not reached do
8:     if current instruction = CHECKPOINT then
9:         // If we reach a checkpoint, we keep track of its program counter
10:        // and reset the reset counter.
11:        current_checkpoint ←Program Counter
12:        for each non-idempotent output routine do
13:            reset_no ← 0
14:        save snapshot
15:        run current_instruction
16:
17:        // Ignore if we already analyzed it
18:    else if current_instruction = output(O_i, R_x) and reset_no[O_i] < 2
   then
19:        // If we reach an output instruction, we keep track of the
20:        // output port, the written value, the resets counter, and the
21:        // program counter of the instruction to be executed.
22:        profiling_info = (O_i, R_x, reset_no, program_counter)
23:        append profiling_info to execution_trace[current_checkpoint]
24:        run current_instruction
25:
26:        // If reset not generated, then generate it
27:        if reset_no[O_i] == 0 then
28:            reset_no[O_i] ← 1
29:            reset
30:            restore checkpoint
31:        else if reset_no[O_i] == 1 then
32:            reset_no[O_i] ← 2
33:            // Restore the snapshot so to not cause other inconsistencies
34:            restore snapshot
35:    else
36:        run current instruction
37:
38: Check the content of execution_trace to verify if the behavior is the one
    we expect
```

```
1 [...]
2 checkpoint();
3 pos = input(O1);
4 mov = 45-pos;
5 output(O1, mov);
6 [...]
```

Example 5.19: Possible fix to the output inconsistency that Example 5.18 has. We use O_1 as an *intermittence-based input*.

```
1 [...]
2 checkpoint();
3 if(input(O1) == 30) {
4     output(O1, 15);
5 }
6 [...]
```

Example 5.20: Another possible solution to solve the output inconsistency present in Example 5.18, which exploits an *intermittence-based input*.

Let us suppose that we want to analyze the presence of output inconsistencies in Example 5.19. Firstly, we analyze the output function present at line 5. It is non-idempotent since moves a servo incrementally, and thus its re-execution leads to an inconsistent environment state. We initialize the *reset_no* data structure, and we start the execution of the program. We reach the checkpoint at line 2, we update the *current_checkpoint* variable, we re-initialize the *reset_no* data structure, and we save a snapshot of the device state. Then, we execute the checkpoint, and we reach the instruction at line 3. Let us suppose that the initial position of the servo is 15°, and thus the variable *pos* is set to 15. Since this instruction is not a checkpoint or an output routine, we simply execute it. Then, we execute the instruction at line 4, and the variable *mov* is set to 30. As next operation, we reach the *output* instruction of line 5. We verify $reset\_no[O_1]$, which is 0, and thus we save the profiling information associated to this operation. Then, we increment $reset\_no[O_1]$ to, and we generate a power reset. The same workflow is repeated until we reach the output operation at line 5, for which we save the profiling information. Since we reached this operation twice, we restore the snapshot, and we continue the execution. When the program ends, we see the following execution trace: $(O_1, 30, 0, 2)$ and $(O_1, 0, 1, 2)$. The first execution moves the servo by 30°, and the second one leaves it at its current position. For this reason, no *output inconsistency* happens due to the re-execution of the non-idempotent output routine at line 5.

Let us now consider how we can solve output inconsistencies. We can use three different approaches:

- We can make the non-idempotent output routine idempotent. For example, let us consider the output function of Example 5.18. It uses a relative measure, since it moves the servo by 15° from its current position. If instead we make it use an absolute measure, the output function becomes idempotent with respect to its re-execution.

In fact, the servo will be moved to a specific position, and the re-execution of the action will not change the servo state.

- We can use an intermittence-based input for evaluating the environment state during runtime , and then we can adjust the behavior of the output routine so to avoid output inconsistencies. As we stated in Section 5.3, by allowing the presence of inconsistencies we are able to consider a new set of inputs, which is a function of the intermittence. We can allow not only data inconsistencies, but also output ones.

  Let us consider Example 5.19, which we previously described. In fact, let us suppose the output function is executed for the first time, moving the servo to 45°. Moreover, let us suppose that, just after the movement of the servo, a shutdown happens due to a low energy buffer. When there is enough energy to restart the computation, the checkpoint is restored, and the servo has already a position of 45°. The obtained environment state is inconsistent, since when the checkpoint was taken, the servo was at 30°. The combination of line 3 and 4 accounts for such inconsistency, and they influence the behavior of the output routine of line 5, with the effect of keeping the environment state consistent. As result, the effect of the non-idempotent output routine is idempotent, and thus it does not cause any *output inconsistency.*

  Example 5.20 shows a different approach to the same problem. In this case, the code focuses on avoiding the re-execution of the output function, instead of making its effect idempotent. When we reach the if statement at line 3, it verifies the current position of the servo. If it is 30°, the output function is executed, otherwise it is ignored. As result, the servo will always be at the correct position, independently of how many power resets happen.

Finally, if we introduce *intermittence-based inputs* in our program, we can no longer use Algorithm 9 for verifying the presence of output inconsistencies, since it is not able to analyze them. Testing output inconsistencies is a particular scenario of testing the entire behavior of the program, and thus we can analyze them using Algorithm 8, which we described in the analysis of *intermittence-based inputs.*

Chapter 6

ScEpTIC: Testing Intermittent Computation

6.1 Overview

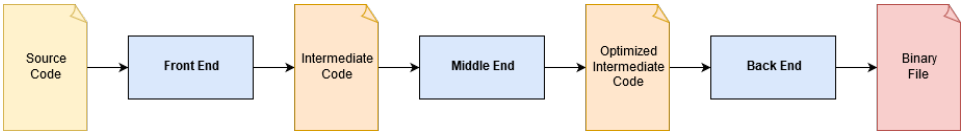

Figure 6.1: Compiler pipeline.

`ScEpTIC` stands for **S**imulator for **E**xecuting and **T**esting **I**ntermittent **C**omputation, and consists in a simulator written in python that executes **LLVM IR** code, which is an intermediate representation of the program source code.

For understanding this design choice, let us briefly focus on how a compiler is structured and works, as shown in Figure 6.1. Usually it is composed by three main modules: front-end, middle-end and back-end.

The front-end translates the source file into an intermediate representation, which is described by a language different from the source one. It uses unlimited temporary registers, abstracted memory locations, and has a structure which simplifies the work to be done in the middle-end. The middle-end analyzes the intermediate representation and then applies a predefined set of optimizations. The intermediate code produced is then analyzed by the back-end, that establishes the data layout of the program and performs register allocation. The data layout task consists in converting the abstracted memory addresses into the ones used by the architecture, or in setting their offsets with respect to the stack or frame pointers, so to obtain valid memory addresses. Instead, the register allocation is in charge of mapping the virtual registers to the physical ones available in the architec-

ture, which are limited inside the processor. Finally, the back-end converts the obtained code into the target assembly language and then produces the binary file. In doing its tasks, the back-end needs to know all the relevant information about the architecture, such as the number of available registers and the memory address space.

The choice of not executing a compiled binary permits us to have an architecture independent simulator, since the instructions are the ones from the intermediate language and not the ones from the Instruction Set Architecture. Furthermore, executing an intermediate language permits us to maintain almost a complete independence from the source language, and it is possible to create multiple front-ends which transform different languages into the intermediate representation we use. As consequence, `ScEpTIC` is able to analyze code independently of the target device or the source language used.

Maintaining this level of abstraction permits us to test a program without the need of knowing a priori which device will be used, to not specify the memory dimension and to skip tasks like register allocation and data layout. Also, the simulator abstracts checkpoint logic and implements the most common checkpoint mechanisms so to enable testing with any of them without the need of eventually porting their code to suit the used language or architecture. For static checkpoint mechanisms that uses specific compiler passes to place checkpoints, it is possible to provide the produced intermediate code without the need of modifying it. The only required action is to specify the name of the checkpoint and reset routines.

The only modification required is to replace the architecture-dependent function calls, such as I/O requests, with the one abstracted by the simulator.

Currently, `ScEpTIC` does not support functions with variable arguments, function pointers, and interrupts. It supports all the data structures and operations of the C language. We design `ScEpTIC` to make easy the process of extending the supported operations.

6.2 LLVM IR

6.2.1 LLVM and Intermediate Representation

LLVM [23] is a compiler framework which provides a set of modules that can be combined to produce compilers, and supports a broad range of source languages and architectures. The intermediate representation LLVM uses is called *LLVM IR* [22] and it is available in different fashions. `ScEpTIC` uses the assembly version, which is human readable and consists in a typed RISC-based assembly.

The input file that the simulator takes is a LLVM IR code which we can produce using clang [25], a C front-end for LLVM, specifying to output the

LLVM IR with debug information. The complete command is:

```
clang -emit-llvm -S -g source_file.c
```

The argument -emit-llvm tells clang to emit the intermediate representation of the source code and the argument -S specifies to emit a textual assembly file, instead of a binary one. To report errors during testing, it is required to map the lines of the intermediate representation with the corresponding ones of the source file. This is achieved thanks to the argument -g, that appends debugging information to the LLVM IR.

If the source program is composed by multiple compile units, we must produce a LLVM IR file for each of them. Then, it is possible to combine the obtained files using the linker provided within LLVM, using the command:

```
llvm-link -S file_1.ll file_2.ll ... file_n.ll -o file.ll
```

This command stores the result inside the file `file.ll`, that contains the source code that `ScEpTIC` takes as input.

Since LLVM IR is an intermediate representation of the source code, it uses virtual registers in each instruction, which are defined as %N}, with N representing a positive integer identifier of the virtual register. Also, LLVM IR is in a Static Single Assignment form, which means that each virtual register is defined only once and each definition of a given virtual register precedes the instructions which uses it. This simplifies the analysis `ScEpTIC` performs.

6.2.2 IR File Structure

From a practical point of view, it is possible to divide the LLVM IR file into six different sections:

- **Headers**: this section contains the specifications of the source file name, the target data layout, and the target architecture. `ScEpTIC` does not consider this section, since it does not provide useful information from a testing standpoint.
- **Custom Type Definitions**: this section contains the definitions of the custom types present in the program, such as C structures and unions. Each custom type definition is declared as a named list of LLVM *first-class types*:

  ```
  %type_name = type{type_1, type_2, ...}
  ```

  For example, a structure named car with two integer fields is declared as:

  ```
  %struct.car = type{i32, i32}
  ```

  Table 6.1 shows the most commonly used first-class types.
- **Global Variable Definitions**: this section contains the definition of all the global variables present in the code. Each variable is defined as:

  ```
  @variable_name = <type> [initial value] [, section "name"]
  ```

Type	Description
iN	represents a N-bit integer. e.g. i32 is a 32-bit integer
half	represents a 16-bit floating-point value.
float	represents a 32-bit floating-point value.
double	represents a 64-bit floating-point value.

Table 6.1: Most used LLVM IR first-class types.

The LLVM IR does not contain the information produced by the data layout process: addresses are not explicit and variable names correspond to a textual representation of the associated memory address.

Variable types are defined as elements of the first-class type or as custom one. The initial value is typed, which means that it is composed by the type and the actual initial value.

The section parameter is used to define which variables are stored into NVM, if any. There are other additional parameters in the definition of a global variable, but `ScEpTIC` does not considers them, since they are not relevant for the analysis it performs.

For example, a variable named 'test' with type i32 and initial value equal to 0 is defined as:

```
@test = i32 0
```

- **Function Definitions and Declarations**: this section contains both the definitions and declarations of the routines present in the source code. A function declaration does not include the body of the function, and it is written as:

  ```
  declare <Return Type> @<Function Name> ([args list])
  ```

  Instead, a function definition includes the body of the function, and it is written as:

  ```
  define <Return Type> @<Function Name> ([args list]) {
      <Function Body>
      ...
  }
  ```

  Definitions and declarations have the same attributes, and we represented only the ones that are relevant for the simulation that `ScEpTIC` performs.

  The argument list contains the types of each argument expected by the associated function. The function body contains an ordered sequence of instructions, organized into basic blocks and divided by labels. If only one basic block is present, the label is omitted. In the function

Metadata Type	Description
DIFile	Represents a file. Its relevant attributes are *file name* and *file directory*.
DILocation	Represents a specific location inside the source file. Its relevant attributes are *line*, *column* and *scope*. Line and columns identifies the position of the location inside the source file, and the scope refers to another metadata of type DIFile, DILexialBlock or DISubprogram.
DISubprogram	Represents the location of a function or subroutine inside the source file. Its relevant attributes are *subprogram name* and *file*. This last one refers to a metadata of type DIFile.
DILexicalBlock	Represents a basic block inside a subprogram. Its relevant attributes are *scope* and *file*.
DILocalVariable	Represents a local variable. Its relevant attributes are *variable name*, *scope*, *line*, *column*.

Table 6.2: Relevant LLVM IR meatadata and their attributes.

body, the arguments of the function are implicitly considered to be stored in virtual registers, starting from %0 to %(n−1), with n equals to the number of arguments. Virtual register ids restart from %0 in each function body.

- **Attribute Groups**: in this section are defined the attribute groups associated to each function. They represent a series of back-end directives or properties that are not useful for our analysis, and for this reason `ScEpTIC` does not consider them.

- **Metadata**: this section contains all the metadata/debug information of the program. A metadata is usually defined as:

  ```
  !metadata_id = !MetadataType(attribute: value, ...)
  ```

  There are several metadata types, each one with its specific attributes. For our analysis we consider only the ones enabling source code mapping, that are listed in Table 6.2.

  For example, a group of metadata referring to line 5 and column 3 of file '/path/test.c' is defined as:

  ```
  !DIFile(filename: 'test.c', directory: '/path')
  !DILocation(scope: !0, line: 5, column: 3)
  ```

We can link a metadata information to any element of the LLVM IR by appending to its attributes a new one defined as "*!dbg !ID*", with ID equal to the metadata identifier. *DILocation* metadata makes us able to create a mapping between the lines of the LLVM IR and the source code. For example, if the metadata *!1* of the previous example is associated to a variable, the definition of such variable will be:

```
@test = i32 0, !dbg !1
```

6.2.3 LLVM IR Labels and Basic Blocks

Inside the function body is defined a list of basic blocks, and each of them contain an ordered lists of instructions. A basic block has a single entry point, which means that the execution of the code inside it always starts from the first contained instruction. To each basic block it is associated a label, which permits to identify it and eventually start its execution, modifying the program counter.
Labels are used to separate basic blocks and are defined before the first instruction of it as:

```
; <label>:ID:
```

The label of the first basic block is implicitly defined with id equal to the number of arguments of the function in which it resides. This is done because labels are treated as temporary, and thus are identified in the same way virtual registers are: `%id`.

6.2.4 LLVM IR Instructions

Each basic block contains a list of LLVM IR instructions, that are defined as:

```
%virtual_reg_id = instructon ...
```

This notation means that the result of the instruction will be stored into the specified virtual register. If we omit the virtual register assignment, the result of the instruction will be ignored.

All the instruction operands are typed, and they can refer to virtual registers, global variables, function names, or labels.

The LLVM documentation [26] classifies instructions into different groups. The ones `ScEpTIC` supports are:

- **Binary Instructions**: a binary instruction perform a binary operation over two operands of the same type, and stores the result in the specified target register. Binary instructions are defined as:

  ```
  %target_register = <code> <type> <operand_1>, <operand_2>
  ```

The operation code specifies the binary operation type. For example, the increment of the virtual register %0 by two is written as:

```
%1 = add i32 %0, 2
```

Instead, the decrement of the virtual register %0 by two is written as:

```
%1 = sub i32 %0, 2
```

`ScEpTIC` supports all the available operation codes listed in the LLVM IR documentation [26].

- **Conversion Instructions**: those instructions perform a bit-level conversion from a source type to a target one.

  The conversion instructions are defined as:

  ```
  %target_register = <code> <type> <operand> to <target_type>
  ```

  The operation code specifies the conversion type.

  For example, the conversion of the signed integer 15 to the corresponding floating point value is written as:

  ```
  %1 = sitofp i32 15 to float
  ```

  `ScEpTIC` supports all the available operation codes listed in the LLVM IR documentation [26].

- **Memory Instructions**: the instructions of this group are the only ones able to interact with the memory. The most relevant ones are:

  - **alloca**: it allocates a given amount of memory in the stack frame, and returns the address of such memory area. Alloca operations are defined as:

    ```
    %target_register = alloca <type>
    ```

    For example, the following instruction allocates 32bits in the stack:

    ```
    %1 = alloca i32
    ```

  - **load**: it loads a value from the given memory address and stores it into the target register. Loads operations are defined as:

    ```
    %target_register = load <type>, <type>* <operand>
    ```

    For example, the following operation loads into the register %2 the value present in the memory location specified by the address contained in the virtual register %1:

    ```
    %2 = load i32, i32* %1
    ```

  - **store**: it writes a given value into a specified memory address. Store operations are defined as:

    ```
    store <type> <value>, <type>* <address>
    ```

For example, the following operation writes the value 7 into the memory location specified by the address contained in the virtual register %1:

```
store i32 7, i32* %1
```

– **getelementptr**: it returns the address calculated from a given base address and a list of offsets. This operation only performs address calculation, and does not generate any memory access. Getelementptr operations are defined as:

```
getelementptr <type>, <type>* <base address> {, <type>
    <offset>}*
```

For example, let us suppose we have a global variable @*a* which consists in an array of 10 integer elements. The following operation calculates the address of the 3rd cell:

```
getelementptr [10 x i32], [10 x i32]* @a, i32 0, i32 2
```

The first offset refers to the entire data type, that in this case is 10 x i32, and the subsequent ones refer to actual offsets which we use in the source programming language.

- **Terminator Instructions**: this group of instructions comprehends operations that changes the control flow of the program, such as *return*, *branch* and *switch*.

  From the standpoint of the analysis `ScEpTIC` performs, the most relevant one is *return*. As the name suggest, it returns from a subroutine and sets the return register equal to the specified value.

  Return operations are defined as:

  ```
  ret <type> <value>
  ```

  For example, the following operation returns 0:

  ```
  ret i32 0
  ```

- **Other Instructions**: this group of instructions contains other operations such as comparisons, selection, phi and call.

  This last one is the most relevant for our analysis, and it is defined as:

  ```
  call @<function name>([<type> <arg>, ...])
  ```

  It performs the call of the specified function. The subsequent argument list specifies the parameters that the *call* operation passes to the function. For example, the following operation calls the function *foo()*, using as its arguments two virtual registers:

  ```
  call @foo(i32 %0, i32 %1)
  ```

`ScEpTIC` does not support vectorization, LLVM vector operations, and functions with variable arguments, since at the moment of writing this thesis those features are not available or used in transiently-powered devices.

6.3 Code Representation

6.3.1 ScEpTIC Intermediate Representation

Compilers use intermediate representations for having a more suitable data structure to recognize and modify the source code. Usually, the compiler front-end parses the source code of the program and organizes it within an Abstract Syntax Tree (AST), which is a tree representation of the program structure. In such data structure, each instruction is represented as a sub tree in which its leafs represent the operands of the instruction. The AST of the program is used for verifying the correctness of the syntax of the program, and then it is used for generating an intermediate representation of the source code. It usually consists in an ordered list of abstracted instructions, that is then used by the middle-end for applying optimizations to the code. The result of the middle-end is then used by the back-end for generating the final code.

For example, Figure 6.2 represents the AST that *clang* produces from the code shown in Example 6.1. This AST is used for generating the LLVM IR code shown in Example 6.2.

`ScEpTIC` has the same approach of a compiler: all the simulation and analysis it performs is not done directly on the LLVM IR, because it would be impractical, and instead it uses a more suitable data structure. Figure 6.3 shows the process `ScEpTIC` perform for producing its own intermediate representation. The parser module of `ScEpTIC` analyzes the LLVM IR file that *clang* produces from the source code of the program. When the parser module ends its analysis, it outputs the AST that `ScEpTIC` uses for performing its analysis.

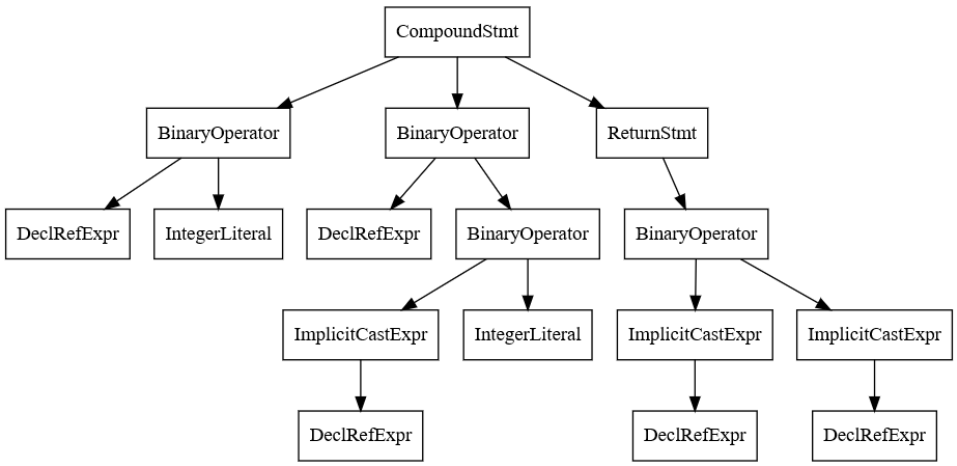

Figure 6.2: AST produced from Example 6.1.

```c
int a, b;
int main() {
    a = 1;
    b = a + 3;
    return a * b;
}
```

Example 6.1: Example of a C program.

```
@a = common global i32 0
@b = common global i32 0

define i32 @main() {
    %1 = alloca i32
    store i32 0, i32* %1
    store i32 1, i32* @a
    %2 = load i32, i32* @a
    %3 = add nsw i32 %2, 3
    store i32 %3, i32* @b
    %4 = load i32, i32* @a
    %5 = load i32, i32* @b
    %6 = mul nsw i32 %4, %5
    ret i32 %6
}
```

Example 6.2: LLVM IR produced from Example 6.1.

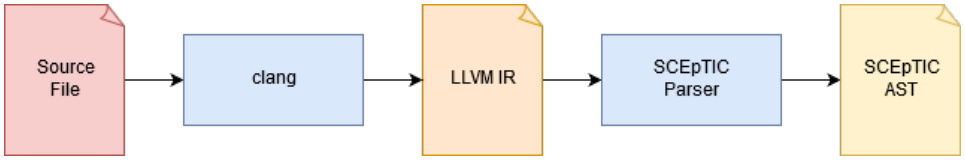

Figure 6.3: Source code parsing pipeline.

Figure 6.4 shows the components of the parser module of `ScEpTIC`, and Figure 6.5 shows its work flow. The **TokenGenerator** converts the source LLVM IR file into a list of tokens, which consists in strings representing a recognizable information of LLVM IR. Tokens do not include spacing characters. For example, `%1 = alloca double` is converted in the list:

```
['%1', '=', 'alloca', 'double']
```

The produced list of tokens is then processed by the **SectionDivider** sub module, that analyzes and splits tokens into different sub lists, one for each LLVM IR section. Each produced sub list is then processed by the corresponding **SectionParser** sub module, which extracts the relevant information of the section and creates the associated `ScEpTIC` **AST**. The *SectionParser* that parses functions uses the **InstructionParser** sub modules to extract the relevant information of instructions, and to produce the corresponding intermediate representation of the function's body.

Once the parsing process completes, the parser module returns the produced `ScEpTIC` **AST**.

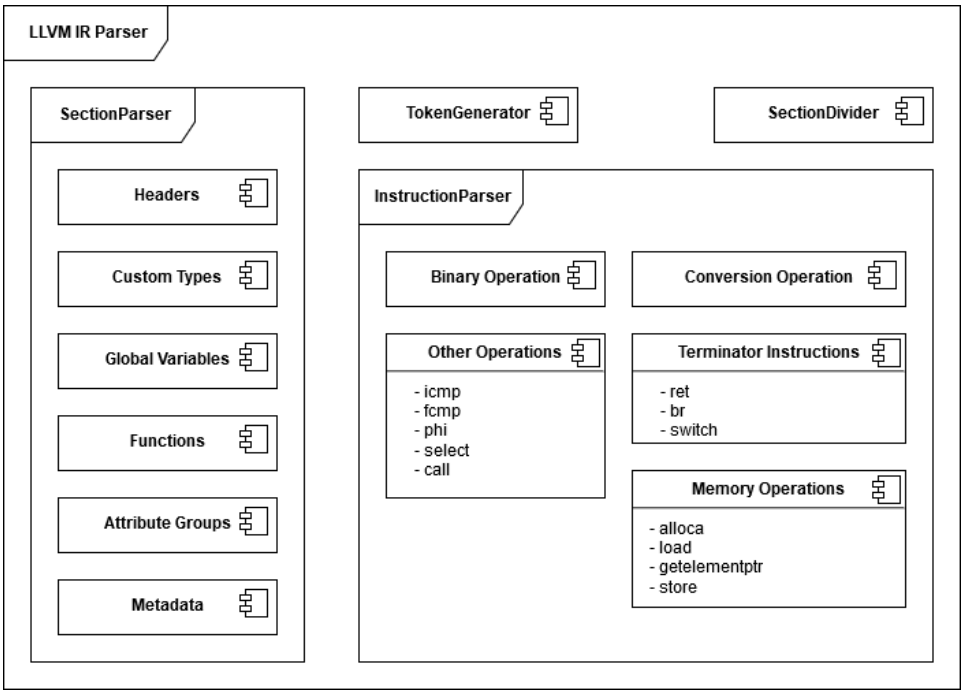

Figure 6.4: `ScEpTIC` Parser Module.

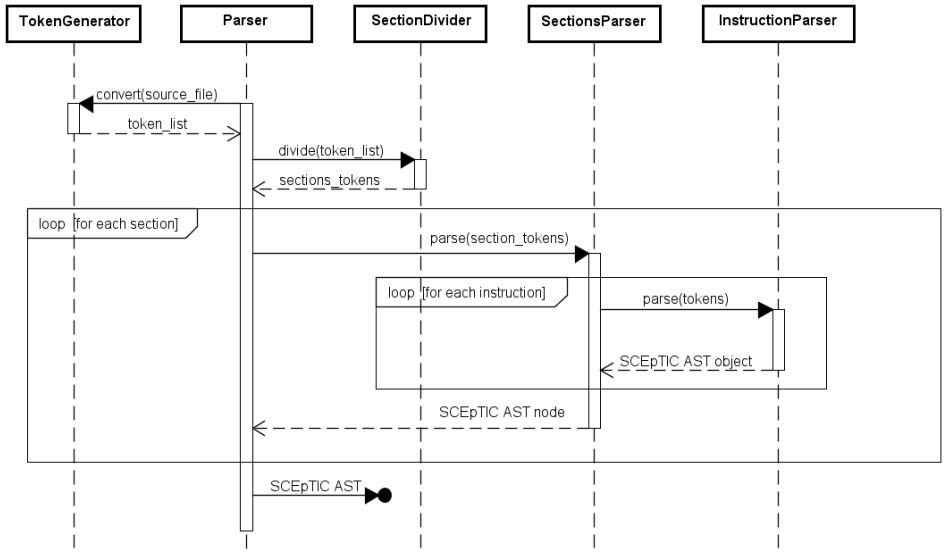

Figure 6.5: Sequence diagram representing the parsing of the source file.

Figure 6.6 shows the structure of *ScEpTIC AST*. The root node is connected to 5 sub trees, each one of them representing a group of information that is relevant for ScEpTIC. The leaf nodes of each sub trees consist in an object of the ScEpTIC **AST Class**, that contains all the relevant information about the represented element.

The available *ScEpTIC AST* Classes are:

- **GlobalVar**: it represents a global variable, and contains its name, the initial value and its type. If the type is a custom one, the node is linked to a *CustomType* object.
- **Function**: it represents both a function definition and declaration. This ScEpTIC AST class contains the function name, the return type and a list of the argument's types. If the object represents a function definition, it also contains the list of instructions composing the function's code.
- **CustomType**: it represents the definition of a custom type, and contains its name and its definition, which is represented as a non-empty list of types.
- **Metadata**: it represents the definition of a metadata, and containts the metadata type and its relevant attributes, that we describe in Table 6.2.

Furthermore, each instruction contained in the function definition is represented as an AST Instruction Class. There are one implementation of *AST Instruction Class* for each LLVM IR instruction. Each type of AST Instruction Class extends the AST Instruction base class and contains all the necessary data that makes ScEpTIC able to run the associated instruction. For example, the *BinaryOperation* class extends *Instruction* and contains the type of the binary operation to be executed, a reference to the target register, and a reference to its two operands.

Instruction operands can be represented either by an *Immediate Value*, a reference to a *Virtual Register*, or a reference to a *Global Variable*. ScEpTIC treats all those elements in the same way, by representing them using an object of the **Value** class, that during runtime is converted into the required value. In this way, ScEpTIC is able to directly resolve composite operands which uses LLVM IR sub-expressions containing *getelementptr* or any conversion instruction.

6.3.2 Libraries and ScEpTIC Built-ins

One of the latest phases of the compiler back-end consists in the linking of libraries, that are files containing a set of routines used by the program. The

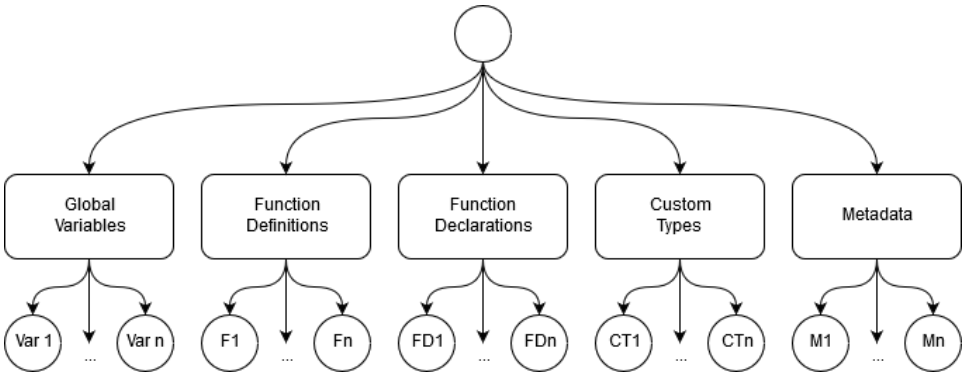

Figure 6.6: ScEpTIC Abstract Syntax Tree.

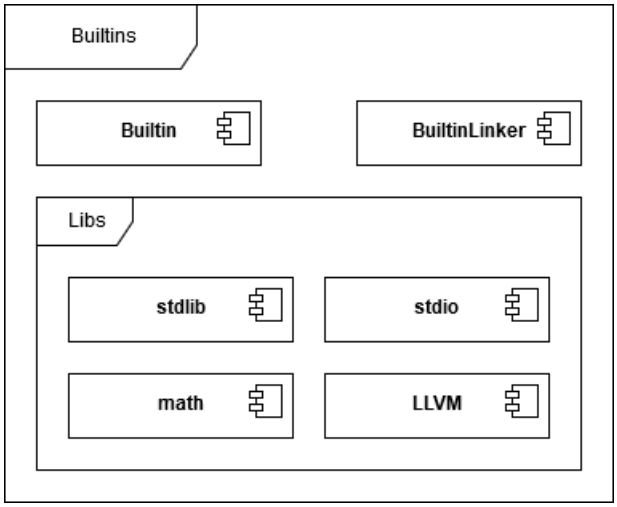

Figure 6.7: ScEpTIC Builtins Module.

linking phase extracts the relevant elements from such files and put them into the final executable binary, among the compiled code of the program.

Usually, libraries are already compiled for the host architecture (i.e., the one compiling), and thus they must be re-compiled for the target architecture if it differs from the hosting one. For this reason, the code of libraries does not appear in the LLVM IR, and instead it contains only the declaration of each library function.

To overcome the problem of not having the code of library functions in the LLVM IR, and thus in the *ScEpTIC AST*, ScEpTIC provides the **Builtins** module. It provides the implementation of the most common function libraries, and automatically performs their linking to the *ScEpTIC AST*. Figure 6.7 shows the components of this module, which are:

- **Builtin**: it is a class extending the *AST Instruction* base class, and it is used for providing the same functionality of a library function. We can use this class for implementing our own library functions. For each library function we want to implement, we must create a new class extending *Builtin*, and within it we must specify the implementation of such functionality. Then, we can call the *define_builtin* method over the created class, without instantiating a new object.

- **BuiltinLinker**: it emulates the functionality of a linker, and links the defined functions into the *ScEpTIC AST*. To achieve that, for each defined *Builtin*, it accesses the *ScEpTIC AST* and searches for the corresponding *Function Declaration*. If no declaration is found, it skips to the next *Builtin* definition. Otherwise, it firstly creates the *function prologue*, which consists in a group of instruction that only load the function arguments from the stack. Then, it appends the actual code, which is represented by the extended *Builtin* class. Finally, it creates the *function epilogue*, which contains a *ReturnOperation* that returns the value computed by the function implementation.

  With these tasks it created the list of instructions composing the function code. As final operation, it appends such list to the found declaration, transforming it into a *Function Definition*.

  The *BuiltinLinker* is automatically invoked when the parser module finishes producing the `ScEpTIC` AST.

- **Libs**: this sub module contains the definition of the functionalities provided by the most used C libraries, which are:

  - **math.h**: it provides the functionalities of acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, log, log10, pow, sin, sinh, sqrt, tan, and tanh.
  - **stdio.h**: it provides the functionalities of the printf.
  - **stdlib.h**: it provides the functionalities of calloc, free, malloc, and realloc.

  The *Builtins* module automatically tries to link the functionalities contained in the *Libs* sub module.

We can use the *Builtins* module for providing the implementation of other library functions. Let us suppose we want to provide the implementation of the function *cos*, that is included in the C math library:

```
double cos(double x)
```

Example 6.3 shows how we can implement it. Firstly, we must create the class *cos* by extending *Builtin*. Then, we must specify in the *get_val* method the code which provides the same result of the implemented function.

```
1  # import python math library
2  import math
3
4  # Define the class which provides the implementation
5  class cos(Builtin):
6  def get_val(self):
7  return math.cos(self.args[0])
8
9  # Define the builtin
10 cos.define_builtin('cos', 'double', 'double')
```

Example 6.3: Implementation of the *cos* function included in the math library.

```
1  ; Function Prologue
2  ; Save parameter into the stack
3  %1 = alloca double
4  store double %0, double* %1
5  ; Load argument from the stack
6  %2 = load double %1
7
8  ; Function Code
9  %3 = cos(double %2)
10
11 ; Function Epilogue
12 ret double %3
```

Example 6.4: LLVM IR version of the produced `ScEpTIC` AST by the *BuiltinLinker* from Example 6.3.

Such method must return the result of the operation, and it is the one that `ScEpTIC` executes during the analysis. In the implementation of the function, we can access the arguments using the variable *self.args*. It consists in a list that is automatically populated during runtime with the values of the passed arguments. As final step, we call *define_builtin* on the created class, passing as arguments the function name, the return type, and the argument type.

Before `ScEpTIC` starts the simulation, it invokes the *BuiltinLinker*, which generates the *`ScEpTIC` AST* for the *cos* function and attaches it to the corresponding function declaration.

Example 6.4 shows the LLVM IR version of the generated `ScEpTIC` AST. Each line corresponds to an extension of *`ScEpTIC` AST Instruction*, and the element at line 9 is exactly the builtin we created.

6.3.3 Register Allocation and Tests

The register allocation process maps virtual registers to the physical one available in the architecture, and it is performed by the back-end. This step may introduce memory operations that lead to inconsistencies, and ignoring it may lead to an inaccurate analysis. For this reason, `ScEpTIC` permits us to perform the register allocation over the *ScEpTIC AST*.

Let us suppose that we have a register file with only two registers, *R0* and *R1*, and that we want to perform the register allocation on Example 6.5. We start from the first instruction, and we map the virtual register *%0* into *R0*. For the second instruction, we replace the occurrence of *%0* with *R0*, and then we map the virtual register *%1* to *R1*. Now we do not have any available register, since they are all allocated. The values contained in both *R0* and *R1* are needed by future instructions, so we must preserve such values. To continue the register allocation on instruction at line 3, we must save a register into the stack. This operation is called **spill**. The value of *%1* is not required until the instruction at line 5, so we can save it into the stack by inserting a store operation. Now we can map *%2* into *R1*, and then we replace all the occurrence of *%1* with *R1*. Before performing the register allocation over instruction at line 5, we need to restore the value of register *R1*, which was saved in the stack by a previous *spill*. This operation is called *promotion*, and for performing it we insert a load before the instruction at line 5.

Example 6.6 shows the overall result of the register allocation, in which the *alloca* instruction at line 1 was inserted to show that we need some stack space to save the register.

```
1 %0 = load i32, i32* @var
2 %1 = add i32 %0, 1
3 %2 = sub i32 %0, 3
4 store i32 %2, i32* @var
5 %3 = mul i32 %1, 5
6 [...]
```

Example 6.5: LLVM IR code on which we want to perform register allocation.

```
1  %spill = alloca i32
2  R0 = load i32, i32* @var
3  R1 = add i32 R0, 1
4  ; Spill register R1
5  store i32 R1, i32* %spill
6  R1 = sub i32 R0, 3
7  store i32 R1, i32* @var
8  ; Promote register R1
9  R1 = load i32, i32* %spill
10 R0 = mul i32 R1, 5
11 [...]
```

Example 6.6: Results of the application of register allocation on Example 6.5.

```
1  define i32 @main() {
2      %1 = alloca i32
3      %2 = alloca i32
4      %3 = alloca i32
5      store i32 0, i32* %1
6      store i32 5, i32* %2
7      store i32 6, i32* %3
8      %4 = load i32, i32* %2
9      %5 = load i32, i32* %3
10     %6 = call i32 @foo(i32
           %4, i32 %5)
11     ret i32 %6
12 }
13
14 define i32 @foo(i32, i32) {
15     %3 = alloca i32
16     %4 = alloca i32
17     %5 = alloca i32
18     store i32 %0, i32* %3
19     store i32 %1, i32* %4
20     %6 = load i32, i32* %3
21     %7 = load i32, i32* %4
22     %8 = add i32 %6, %7
23     store i32 %8, i32* %5
24     %9 = load i32, i32* %5
25     ret i32 %9
26 }
```

Example 6.7: LLVM IR code on which we want to perform register allocation.

```
1  define i32 @main() {
2      %1 = alloca i32
3      %2 = alloca i32
4      %3 = alloca i32
5      store i32 0, i32* %1
6      store i32 5, i32* %2
7      store i32 6, i32* %3
8      R0 = load i32, i32* %2
9      R1 = load i32, i32* %3
10     R0 = call i32 @foo(i32
           R0, i32 R1)
11     ret i32 R0
12 }
13
14 define i32 @foo(i32, i32) {
15     %3 = alloca i32
16     %4 = alloca i32
17     %5 = alloca i32
18     store i32 %0, i32* %3
19     store i32 %1, i32* %4
20     R0 = load i32, i32* %3
21     R1 = load i32, i32* %4
22     R2 = add i32 R0, R1
23     store i32 R2, i32* %5
24     R0 = load i32, i32* %5
25     ret i32 R0
26 }
```

Example 6.8: Results of the application of register allocation on Example 6.7.

As is possible to see in the described example, after performing the register allocation, it is possible that we introduce in the code some memory load and store instructions. Such insertions may change the results of testing, making it imprecise, and thus we should perform register allocation for having an accurate result.

Register allocation should also account for function calls. Before each call, the registers that the called routine uses are saved into the stack, to preserve their value. This operation is performed only if some registers used by the caller are modified by the callee. If this step is not considered, the analysis done in `ScEpTIC` could be imprecise, due to unreal memory accesses.

Compilers tries to minimize the registers saved before function calls by partitioning them among functions, in a way which reduces usage overlaps. Example 6.8 shows the result of a register allocation performed over Example 6.7, in which we did not partition any register. Furthermore, let us suppose we have a register file with 10 registers. Note that *alloca* opera-

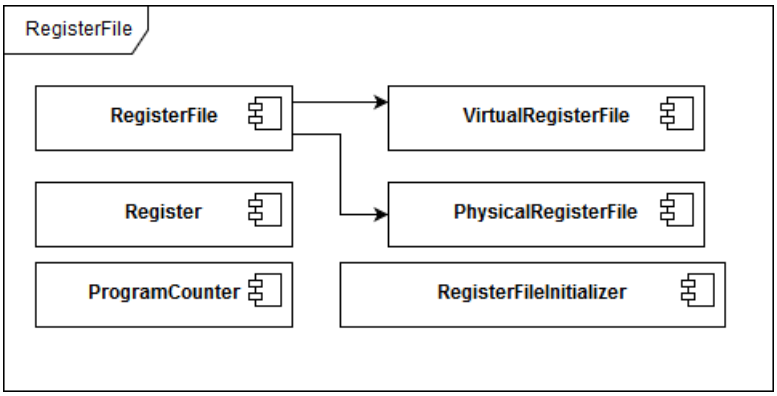

Figure 6.8: `ScEpTIC` Linear Scan Register Allocation Module.

tions do not correspond to a real instruction, and its result is a stack offset which will be explicitly set in the code during the compilation phase. For this reason, it is ignored during register allocation process that `ScEpTIC` performs.

As we can see, registers are not partitioned, and both *main* and *foo* uses the same ones. When the function *foo* is called, there is an overlapping of registers that requires saving them for preserving their values. If we consider the maximum amount of registers that each function require, we can see that *main* uses at most 2 registers, and the *foo* uses 3. The overall usage of registers is below the number of the ones available in the register file. For this reason, we can assign other registers to *foo*, so to remove the overlapping with the registers that *main* uses. For example, we can substitute R0, R1 and R2 with respectively R3, R4 and R5. As result, we obtain a partitioning which avoids registers savings, and thus reduces memory operations.

This kind of analysis is not trivial, and `ScEpTIC` uses a technique that makes it able to obtain the same result without performing register partitioning. Figure 6.8 shows the structure of the module which performs the register allocation. `ScEpTIC` applies the register allocation directly on the *`ScEpTIC` AST*, using the algorithm called *Linear Scan Register Allocation* [27], that is used by real-world compilers.

The **LinearScanRegisterAllocator** performs the register allocation, and then it estimates the partitioning of registers:

1. It allocates registers to each function individually, without considering the ones mapped in other functions. The module does this operation using the **RegisterPool** class, which helps in keeping track of available registers.

2. It calculates the overall number of registers that each function uses. Such number is equal to the actual number of registers that each function uses, plus the maximum number of registers used by a function

Algorithm 10 Iterative calculation of the overall number of registers used by each function.

Require: number of registers present in the register file (n_regs).

```
1:  // Init
2:  for each function f do
3:      calls[f] ← list of functions called by f
4:      reg_count[f] ← maximum number of registers used by f
5:      max_reg_count[f] ← reg_count[f]
6:
7:  // Iterative calculation
8:  do
9:      for each function f do
10:         // find max reg usage of called functions
11:         max_call_regs ← 0
12:         for each j in calls[f] do
13:             max_call_regs ← max(max_call_regs, max_reg_count[f])
14:
15:         // The register usage is equal to max_call_regs + reg_count[f]
16:         // It must be limited to n_regs, since if it exceeds this number
17:         // all the registers must be saved, which is exactly n_regs
18:         max_reg_count[f] ← min(max_call_regs + reg_count[f], n_regs)
19:
20: until fixed point reached
```

that is called inside the one we are considering. Recursive calls are ignored, since they do not introduce any new register usage.

Algorithm 10 shows how we can calculate this parameter. The maximum number of registers is initialized to the ones used by the function, and then it is iteratively updated. At each step, for each function we find the maximum number of registers used by a called routine. Then, we sum this number to the amount of registers used by the considered function. The computation stops when we reach a fixed point that is, when the number of registers used by each function is the same one we calculated in the previous iteration. We must compute this parameter iteratively because it is possible to have call cycles (i.e., a calls b; b calls a;). They induce cyclic dependency among the values we calculate, and we could be stuck in an infinite loop during the computation. Iterations remove this problem.

3. As next step, it performs the register partitioning. For each call instruction appearing in a function code:

   3.1. It computes the number of registers to be saved:
   $$used\_regs = |n\_resg + n\_call\_regs - reg\_file\_dim|$$
   with n_regs equal to the number of registers used by the calling function, n_call_regs equal to the number of registers used by the called function, and reg_file_dim equal to the total number of available registers.

   3.2. It inserts an instance of the **SaveRegistersOperation** class before the call instruction, initializing it with $used\_regs$. When the *SaveRegistersOperation* is executed, it saves the entire register file into its internal state, and then allocates $used\_regs$ cells into the stack, to emulate the register saving operations.

   3.3. It inserts an instance of the **RestoreRegistersOperation** class after the call instruction, initializing it with the reference to the corresponding *SaveRegistersOperation*. When *RestoreRegistersOperation* is executed, it restores the registers, except for the one containing the returne value.

   During the execution, *SaveRegistersOperation* saves the entire register file, since the register allocation is performed without considering the other routines. This, along with the $used\_regs$ parameter, permits `ScEpTIC` to emulate the registers partitioning obtained by a real-world compiler, without actually performing it.

For a better comprehension, let us apply this procedure to Example 6.7. We allocate the registers, obtaining the result that Example 6.8 shows. As next step we need to apply Algorithm 10. We initialize the data structures as shown in Table 6.3: *main* uses 2 registers (R0, and R1), and *foo* uses 3 registers (R0, R1, and R2). We start the iteration process, and it ends in just one step: we obtain a usage of 5 registers for *main* and 3 for *foo*. Now we apply the third step of the procedure. We insert *SaveRegistersOperation()* and *RestoreRegistersOperation()* respectively before and after the call present at line 10 of the *main* function. Example 6.9 shows the final result of the register allocation process.

Function Name	Reg Count	Max Reg Count	Calls
main	2	2	foo
foo	3	3	

Table 6.3: Initialization of Algorithm 10 applied to Example 6.8.

```
1  define i32 @main() {
2      %1 = alloca i32
3      %2 = alloca i32
4      %3 = alloca i32
5      store i32 0, i32* %1
6      store i32 5, i32* %2
7      store i32 6, i32* %3
8      R0 = load i32, i32* %2
9      R1 = load i32, i32* %3
10     SaveRegistersOperation()
11     R0 = call i32 @foo(i32 R0, i32 R1)
12     RestoreRegistersOperation()
13     ret i32 R0
14 }
15
16 define i32 @foo(i32, i32) {
17     %3 = alloca i32
18     %4 = alloca i32
19     %5 = alloca i32
20     store i32 %0, i32* %3
21     store i32 %1, i32* %4
22     R0 = load i32, i32* %3
23     R1 = load i32, i32* %4
24     R2 = add i32 R0, R1
25     store i32 R2, i32* %5
26     R0 = load i32, i32* %5
27     ret i32 R0
28 }
```

Example 6.9: Results of the application of the entire procedure over Example 6.8.

If a compiler for our target architecture uses a different register allocation algorithm, we may want to perform the analysis using the register allocation it produces. For this reason, `ScEpTIC` permits us to customize the register allocation algorithm we want to use. We can implement an algorithm of our choice, and then we configure `ScEpTIC` for using it, as we show in Section 6.6. The register allocation algorithm must expose a method that accept the following parameters, in this order:

- *functions*: the `ScEpTIC` AST node containing functions and their codes.
- *registers_number*: the number of available registers in the register file.
- *reg_prefix*: the prefix of the register name.
- *spill_prefix*: the prefix of the virtual register used for spilling registers.
- *spill_type*: the type of the spill registers.

Such method must apply the register allocation to the *`ScEpTIC` AST* that receives as parameter.

6.4 Architecture

6.4.1 Overview

This Section describes the components of `ScEpTIC` and how they can be configured. All the functionality of this testing tool are exposed through a **VM** object, which initializes and configures each element.

The VM object takes from the configuration the LLVM IR source file, and initializes the parser with it. Once the parser produces the initial *`ScEpTIC` AST*, the VM object initializes the **VMState** object, by passing to it the *`ScEpTIC` AST* and the configuration. The sequence diagram of Figure 6.9 shows the overall work flow of `ScEpTIC`.

The *VMState* object represents the runtime state of `ScEpTIC`. It contains the *`ScEpTIC` AST*, the **RegisterFile**, the **Memory**, and the logical clock which permits events ordering.

As first step, it initializes the register file and the memory, with the required configuration. Then, the control is passed to the *VM* object, which initializes the *CheckpointManager*. This component exposes the methods for performing checkpoints during the execution of tests. As next step, it calls the method exposed by the *VMState* object for initializing the global variables and the code to be used.

When the code is initialized, the *VMState* also calls the *BuiltinLinker* to create the missing code of library functions and user-defined ones. As next step, the *VMState* performs the register allocation, if we enabled it in the configuration.

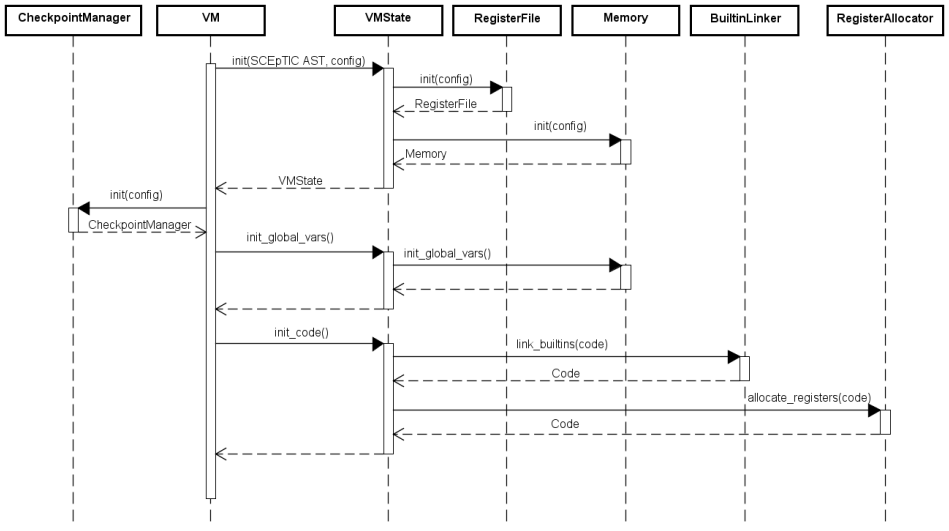

Figure 6.9: Sequence diagram representing the initialization of `ScEpTIC`.

At the end of this stage, `ScEpTIC` is completely initialized. The *VM* object executes the required analysis by loading an *InterruptionManager* for each test to be performed. The *InterruptionManager* is a component that implements the test work flow.

6.4.2 Register File

Figure 6.10 shows the components of the register file module, which are:

- **Register**: this class represents a register and contains its name and value.
- **ProgramCounter**: this class represents the program counter, and consists in a pair (*routine name*, *instruction number*) which contains the instruction to be executed. It also includes an internal stack of program counters, used to track the function calls during the execution of testing. For example, if at line 3 in the main function of the program there is a call to a function *foo*, the current program counter after having performed the call is (*foo*, 0) and the internal call stack is [(main, 2)].
- **RegisterFile**: it is the base class containing all the functionalities of the register file, and provides the methods for interacting with registers. It includes the list of registers, the program counter, and the stack base pointer (BP).
- **VirtualRegisterFile**: it is an implementation of the *RegisterFile* and uses an unlimited number of registers, as the LLVM IR does.

  All the routines in LLVM IR consider to have a separate register file, since the identifier used to track registers starts from 0 in each of them. To have a separate register file for each function call, the VirtualRegisterFile implements an internal call stack. Whenever a function call happens, the current list of registers is put in top of the internal call stack, and such list is reinitialized to accommodate the new virtual registers. When a return instruction is executed, the list of registers on top of the call stack is restored.
- **PhysicalRegisterFile**: it is an implementation of the *RegisterFile* which uses a limited number of registers, as happen in a real architecture. In a real-world scenario, the number of registers is physically limited and this could lead to more interactions with memory, as explained in Section 6.3.3. This register file can be used only if the register allocation is applied to the `ScEpTIC` AST, and apart from the list of registers, it contains also an unlimited list of address registers.

  This is required because in LLVM IR *alloca* operations store their result into virtual registers, but in real-case scenarios this value is

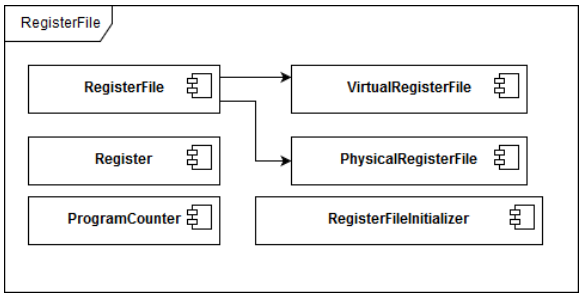

Figure 6.10: Register File module of `ScEpTIC`.

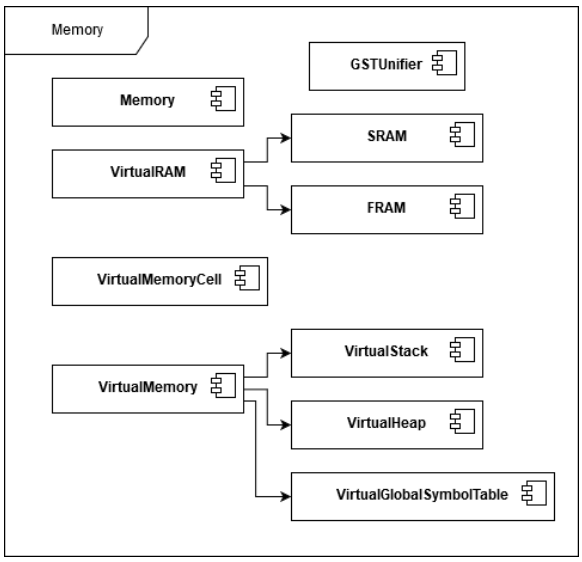

Figure 6.11: Memory module of `ScEpTIC`.

known at compile time. For this reason, the virtual register used as target by *alloca* instructions is not substituted with a physical register during the register allocation process.

- **RegisterFileInitializer**: it initializes and returns the correct implementation of the *RegisterFile* class, given the user configuration.

6.4.3 Memory

Figure 6.11 shows the components of the memory module, which are:

- **Memory**: this class is in charge of the initialization and administration of the entire memory of `ScEpTIC`. It exposes methods to interact

with the various sections of the memory, independently on their location (i.e., RAM or NVM).

- **VirtualRAM**: it is the base class that represents a RAM, and it is implemented by the classes **SRAM** and **FRAM**. The first class represents the Static RAM, which is volatile, and the second one represents the Ferroelectric RAM, which is non-volatile. A RAM class contains the memory sections allocated in the represented RAM, and exposes the functionality for performing the memory reset.

- **VirtualMemoryCell**: it represents a single addressable memory cell and contains the cell address, dimension, and content. This class exposes all the methods for interacting with a memory cell, such as read, write, free, remap, etc. The memory content is not represented in binary format, but is directly represented in numeric or string form, so to have a better view of the state during tests. Integers are by default considered as signed, and if an instruction works with unsigned integers, it converts the operands into the requested form to perform the operation. Then, the result is converted back in signed integer form.

- **VirtualMemory**: it is the base class representing a generic memory section. This class contains the definition of the address space for the represented memory section, a list of the memory cells contained in it, and exposes the methods which permits interacting with the memory cells it contains.

  Since the simulator is architecture-independent and does not require the dimension of the memory, it is not possible to have the memory sections within the same address space, otherwise there will be conflicts due to addresses overlapping.

  To overcome this problem, each memory section is considered to be totally detached from the others, and thus addresses starts from 0.

  For distinguishing different memories, the memory address is composed by two elements: a prefix characterizing the memory section and the relative address of it. For example, if we want to access the memory cell present in the stack section at the address *0x01*, the absolute address we must consider is *STACK-0x01*.

- **VirtualStack**: it is the implementation of *VirtualMemory* which represents the stack memory section. It exposes methods for the allocation and de-allocation of space into the stack, for popping and pushing values from/to the stack, and contains the stack pointer *SP*. Even if it should be a register, SP is contained here because that makes easier to manage the stack.

- **VirtualHeap**: it is the implementation of *VirtualMemory* which represents the heap memory section and exposes methods for managing it. It organizes the allocated memory cells within memory groups, and emulates the behavior of the heap.

  For example, if we perform a free request over a memory cell, all the associated memory group is freed. A free request removes *VirtualMemoryCell* objects only if it is at the end of the list containing the memory cells. Otherwise, it aggregates all the memory cells of the freed group into a single one, which is then marked as garbage. If two adjacent memory cells are marked as garbage, they are merged into a single one *VirtualMemoryCell* object. When an allocation request happens, the *VirtualHeap* firstly search for a garbaged memory cell able to fit the required size. If it finds such memory cell, the *VirtualHeap* allocates it the request, otherwise it creates a cell at the end of the heap.

- **VirtualGlobalSymbolTable**: it is the implementation of *VirtualMemory* which represents the memory section used to store global variables. It exposes methods for accessing them, and keeps track of the *VirtualMemoryCells* containing variables. For doing so, it uses a dictionary that maps the variable name with the relative address of the associated cell.

  Usually, global variables are placed on top of the stack, but since it is possible to have global variables in NVM without having the stack allocated in it, this class is required. Also, for the same reasons, during runtime there will be two instances of this class: one representing the variables in SRAM and one representing the variables in FRAM. In this way, it is more practical managing variable accesses.

- **GSTUnifier**: this class contains the instances of *VirtualGlobalSymbolTable* and provides access to global variables independently of their memory type. It does not retrain memory information, and it just provides transparent access over the Global Symbol Tables.

6.4.4 Input and Output

Transiently powered devices are usually used for interacting with the environment, and thus they may have input and output capabilities. Usually, the function permitting environment interactions are architecture-dependent. To overcome this architecture dependency, `ScEpTIC` treats I/O elements as *Builtin* functions, that the user can define.

Figure 6.12 shows the components of the **I/O** module, which is provided within `ScEpTIC` to enable the support of I/O interactions.

Input. Input functions consists in an extension of the **InputSkeleton** base class, which by itself extends the *Builtin* class and contains all the information about an input. The definition of the class defining an input is similar to the definition of a *`ScEpTIC` Builtin*, which we described in Section 6.3.2.

The *InputSkeleton* class exposes the *define_input* method, that permits mapping the input with the relative element in the *`ScEpTIC` AST*. For defining an input, we can call such method directly on the extended class, and it is defined as:

```
define_input(input_name, function_name, arguments, return_type)
```

Its arguments are:

- *input_name*: the name of the input, which is used as identifier during the analysis.
- *function_name*: the name of the function that corresponds to the created input. This parameter identifies the function that we need to put in our source code for accessing the input data.
- *arguments*: it is a string containing the types of the arguments of the input function. We must express such types as first-class LLVM IR types.
- *return_type*: it is a string defining the type of data that the input function returns. As for the arguments, we must express such type as a first-class LLVM IR type.

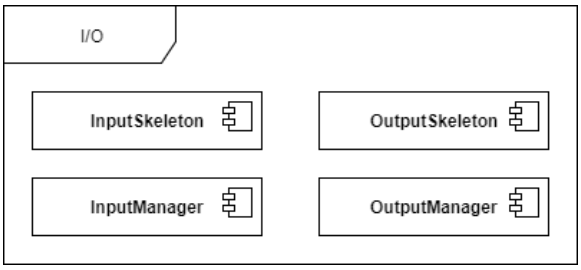

Figure 6.12: I/O module of `ScEpTIC`.

```
1 class MyPersonalyzedInput(InputSkeleton):
2     def get_val(self):
3         self.value = float(self.arg[0] *
              self.arg[1] - 7)
4         return super().get_val()
5
6 MyPersonalyzedInput.define_input('DISTANCE',
      'my_distance', 'i32, i32', 'float')
```

Example 6.10: Example of the definition of an input.

Example 6.10 shows the definition of an input named *DISTANCE*, which we can access using the function named *my_distance*. It has two 32-bit integers as arguments and returns a floating-point value.

Note that any extension of the *InputSkeleton* requires to set the object variable *self.value* to the selected value of the input, and it must call the *get_val()* method of the parent class. This is required because the *get_val()* method that *InputSkeleton* implements keeps track of the current input values, which is required by the analysis `ScEpTIC` performs. The absence of such call makes the analysis unreliable.

The **InputManager** class manages all the inputs during runtime, and exposes a set of methods for analyzing and managing them.

Even if the process of creating an input is trivial, for most use cases arguments are not required, since an input will simply return a predefined value. For simplifying the definition of inputs in this scenario, the *InputManager* exposes a *create_input* method which manages the class extension automatically. It is defined as:

```
create_input(input_name, function_name, return_type)
```

The arguments of this method are the same of *define_input*, except *arguments*, which is omitted due to the previous assumption.

Let us suppose that we want to create an input named *DHT11*, which return a 32-bit integer, and that we want to use the function *input_dht11()* in our source code. To achieve that, we can perform the following operation in the configuration file:

```
InputManager.create_input('DHT11', 'input_dht11', 'i32')
```

For specifying the values of the inputs created through the *InputManager*, we can use the *set_input_value* method it exposes, which takes as arguments the identifier of the input and the value we want to assign.

Output `ScEpTIC` manages outputs in the same way it manages inputs. We can create an output function as an extension of the **OutputSkeleton** base class which extends the *Builtin* class.

The *OutputSkeleton* class exposes the *define_output* method, which per-

```
1 class MyPersonalyzedOutput(OutputSkeleton):
2     def get_val(self):
3         self.value = [self.arg[0], self.arg[1] * 5]
4         super().get_val()
5
6 MyPersonalyzedOutput.define_output('MOTOR',
      'my_motor', 'i32, i32', 'void')
```

Example 6.11: Example of the definition of an output.

mits us mapping the output with the relative element in the *ScEpTIC AST*. We must call this method directly on the extended class, and it is defined as:

```
define_output(output_name, function_name, arguments,
                                    return_type)
```

The arguments of such method are the same ones of the *define_input* method.

Example 6.11 shows the definition of an output named *MOTOR*, which we can access in the source code using the function named *my_motor*. It has as arguments two 32-bit integers, which are sent as output values, and it does not return any value.

Note that, as for the *InputSkeleton*, any extension of the *OutputSkeleton* must set the object variable *self.value* with the selected value of the output, and must call the *get_val()* method of the parent class.

The **OutputManager** class manages all the outputs during runtime, and it exposes methods for analyzing and managing them.

For most use cases, an output function has a single argument, which will contain the data we want to send to the environment. For this reason, as the *InputManager* does, the *OutputManager* exposes a method for creating an output function without extending the *OutputSkeleton*, which is *create_output*. It is defined as:

```
create_output(output_name, function_name, argument, return_type)
```

The arguments of this method are the same of the *define_output*, except for the parameter *argument* which must contain only a single type.

Let us suppose that we want to create an output named *RELE*, which takes as input a 32-bit integer. Furthermore, suppose that it does not return any value, and that we want to use the function *output_rele()* in our source code. To achieve that, we can perform the following operation in the configuration file:

```
OutputManager.create_output('RELE', 'output_rele', 'i32',
                                    'void')
```

6.4.5 Checkpoint Manager

The implementation of a checkpoint mechanism is likely to be architecture-dependent, since it saves registers and memory. Furthermore, a checkpoint mechanism may use features that are available only on certain devices, such as an internal voltage comparators or interrupts.

For running its tests, `ScEpTIC` does not require to emulate the same behavior of a checkpoint mechanism, and it only requires to save the same data. For this reason, it provides a **CheckpointManager** class that exposes the two fundamental routines of a checkpoint mechanism: *do_checkpoint* and *do_restore*.

We do not need to extend or modify the *CheckpointManager*, and we can configure its behavior in the configuration file, as we will describe in Section 6.6.

6.4.6 Interruption Manager

Most of the test we describe in Chapter 4 and Chapter 5 requires an intermittent execution, in which we must generate power resets in precise positions.

For running tests, `ScEpTIC` uses different extensions of the **InterruptionManager** base class, and each one of them implements the work flow of a test case. They also include all the comparisons done for verifying the tested properties.

`ScEpTIC` executes a test by invoking the *run_test* method exposed by the *VM* object. For understanding how `ScEpTIC` performs a test, let us focus on Example 6.12. It consists in a part of the implementation of *run_test* that executes the code and runs a specified analysis over it.

Line 3 dynamically loads an instance of the required interruption manager, and the actual analysis is performed between lines 5 and 10. The method *intermittent_execution_required* at line 6 returns *True* if an intermittent execution flow is required. If so, `ScEpTIC` runs the code calling the *run_with_intermittent_execution* method, which implements an intermittent execution flow and the analysis logic. Otherwise, `ScEpTIC` runs the instruction normally, with the *run_step* method that the *VMState* object exposes.

We can create our own analysis by extending the *InterruptionManager* base class, and we must implement two methods:

- *intermittent_execution_required*: it must return a Boolean value indicating if the execution should continue intermittently. If the test represented by the class does not require an intermittent execution, all the test logic should be implemented in this method, since it is run before the actual execution of each instruction.

- *run_with_intermittent_execution*: it must implement the test logic and the actual intermittent execution of the code. If the test represented

```
1  def run_test(self, module_name, class_name):
2      [...]
3      int_mngr = self._load_interruption_manager(module_name,
           class_name)
4
5      while not self.state.program_end_reached:
6          if int_mngr.intermittent_execution_required():
7              int_mngr.run_with_intermittent_execution()
8
9          else:
10             self.state.run_step()
11     [...]
```

Example 6.12: Part of the *run_test* method exposed by *VM*.

by the class does not require an intermittent execution, this method can be leaved blank.

`ScEpTIC` provides the interruption managers that implements the tests we describe in Chapter 4 and Chapter 5. They are:

- *DataInterruptionManager*: implements all the tests described in Chapter 4.
- *InputInterruptionManager*: implements the tests described in Section 5.2
- *InteractionInterruptionManager*: implements the tests described in Section 5.3

We can specify the analysis that `ScEpTIC` performs inside the configuration file, without intervening on the actual code. Moreover, if we want to verify the sequential execution of our source code, we can just run a test using the *InterruptionManager* base class, that does not execute any analysis and simply runs the code sequentially.

The structure of `ScEpTIC` permits us creating new analysis easily. Once we have implemented the extension of the *InterruptionManager* that fits our needs, we can put the source file in the following folder:

ScEpTIC/emulator/intermittent_executor/interruption_managers/

Then, for running the implemented analysis, we can call the *run_test* method that the *VM* object exposes, by specifying the name of our file and the name of the extended class.

For example, if we implement a new test in the class called *NewTestInterruptionManager* contained in the file named '*new.py*', we can call:

```
run_test('new', 'NewTestInterruptionManager')
```

6.5 ScEpTIC AST Execution

`ScEpTIC` executes the nodes of the *`ScEpTIC` AST* by calling the *run_step* method that the *VMState* object exposes. It retrieves the current instruction and then runs it, as the sequence diagram of Figure 6.13 shows.

When `ScEpTIC` calls the *run_step* method, the *VMState* gets the current program counter from the register file. Then, it retrieves the corresponding instruction from the *`ScEpTIC` AST*, and it calls its *run* method.

Each node of the *`ScEpTIC` AST* that represents an instruction extends the *Instruction* base class, and implements an *get_val* method. It retrieves the value that the execution of the instruction produces, and returns it.

The *Instruction* base class exposes a *run* method, which has the effect of executing the represented instruction. It performs the following operations:

1. It calls the *get_val* method on itself, that returns the result produced by the execution of the instruction. If the instruction has a target register, the result is then stored in it.

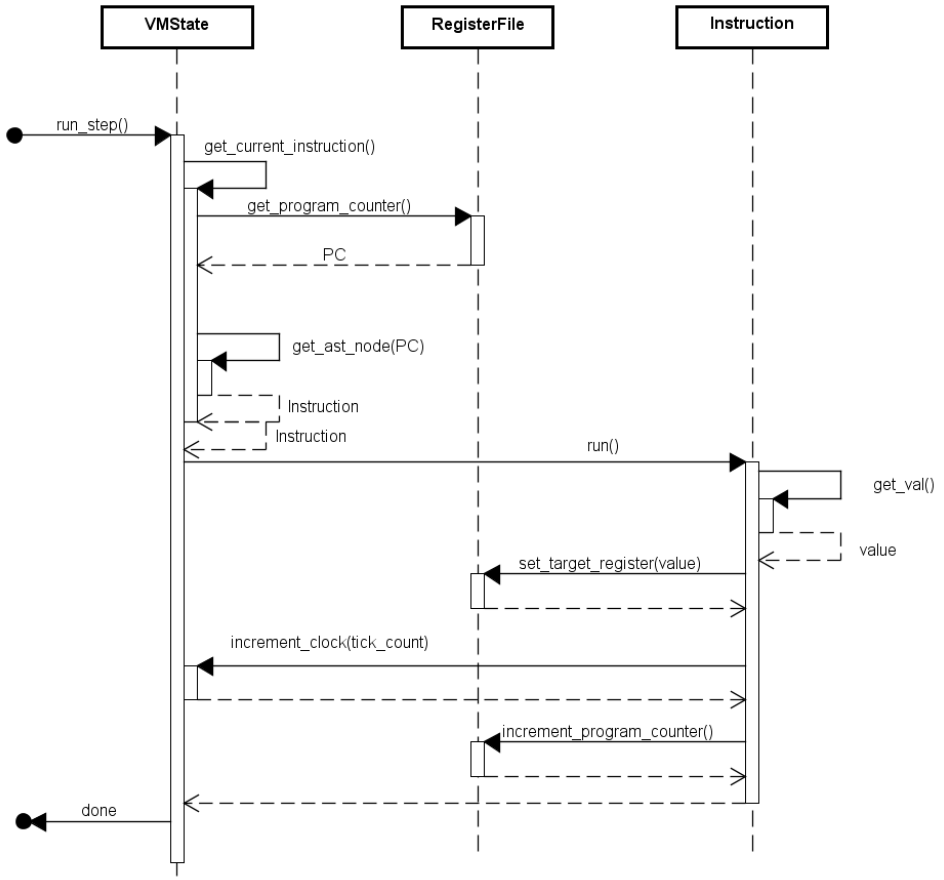

Figure 6.13: Sequence diagram representing the execution of the current instruction in the `ScEpTIC` AST.

2. It increments the global clock by *tick_count*, which is a parameter defined in the instruction. It indicates the number of machine operation that corresponds to the instruction. For example, the *alloca* operation has a *tick_count* of 0.
3. It increments the program counter.

Finally, the *run* method that the *Instruction* base class exposes perform a pre-defined sequence of operations. There are instructions that requires a specific implementation of the *run* method, since they perform other operations. In such case, their implementation overwrites the *run* method that the *Instruction* base class implements. For example, the *run* method that *CallOperation* implements changes the program counter, instead of incrementing it.

6.6 Configuration

6.6.1 Overview

In `ScEpTIC` it is possible to configure each component and to specify which analysis we want to perform over the source code. All the configuration happens in a file, which has a list of variables representing the properties that requires a configuration. In this file we must also insert the definition of custom *Builtins*, *Inputs* and *Outputs*.

We can find an example of the configuration file in the root directory of `ScEpTIC`. It is called *config.sample.py* and shows the different parameters that we can configure.

The configuration file has different parts, that we describe in the following sections.

6.6.2 Test Configuration

This section of the configuration file permits us to set up the source file of our program, the analysis we want to perform, and the output directory where the results of the analysis will be saved.

The variables permitting such actions are:

- *file*: it must contain the path of the source LLVM IR file.
- *save_test_results*: it is a Boolean value which states if tests results should be saved into a file or not.
- *save_llvmir_code*: it is a Boolean value which states if the `ScEpTIC` AST used during tests should be converted into a textual representation and saved into a file.
- *save_vm_state*: it is a Boolean value which states if the register file and memory should be saved into a file or not.

- *save_dir*: it is a string representing the directory where the produced files are saved.
- *run_sequential_program*: it is a Boolean value which states if we want to execute the source file sequentially, without running any test over it.
- *run_data_consistency_test*: it is a Boolean value which states if we want to execute the data consistency test.
- *run_input_consistency_test*: it is a Boolean value which states if we want to execute the input consistency test.
- *run_output_profiling_test*: it is a Boolean value which states if we want to execute the profiling of output interactions with the environment, for finding output inconsistencies.
- *run_interactions_profiling*: it is a Boolean value which states if we want to execute the profiling of interactions with the environment (i.e., input and output executions).
- *execution_depth*: it is an integer value representing the execution depth to test. If the checkpoint mechanisms is configured as static, it is ignored.
- *stop_on_first_inconsistency*: it is a Boolean value which states if tests should stop when an inconsistency is found.

  Depending on our goal and configuration, we might want to stop or continue the analysis at the first inconsistency found. If we are interested in finding all the inconsistencies in our program, we do not want to stop at the first one. On the other hand, if we are interested in resolving them, we may want to stop when we find the first one. In fact, let us suppose that we are testing checkpoints statically placed in our code. As we explain in Chapter 4, for fixing an inconsistency we may require moving a checkpoint or introducing a new one. Such alteration to the checkpoint position makes the analysis no longer valid, and we require to re-analyze the code.

Note that we can configure multiple tests, and `ScEpTIC` will execute all of them.

6.6.3 Register File Configuration

register_file_configuration is the variable that specifies the configuration of the register file. It consists in a python dictionary with the following keys:

- *use_physical_registers*: it is a Boolean value which states if the register file we want to use is a physical or virtual one. If it is set to True, `ScEpTIC` will perform the register allocation step.
- *physical_registers_number*: it is an integer value representing the number of available physical registers. If *use_physical_registers* is set to False, this parameter is ignored.
- *allocator_module_location*: it is a string representing the module location of the register allocator. By default, it is set equal to the module *ScEpTIC.AST.register_allocation*.
- *allocator_module_name*: it is a string representing the module name of the register allocator. By default, it is set to *linear_scan*.
- *allocator_function_name*: it is a string representing the name of the function exposed by the register allocator module, which performs the register allocation. By default, it is set to *allocate_registers*.
- *physical_registers_prefix*: it is a string representing the prefix of the name of physical register. By default, it is set to *R*, meaning that registers will be called *R0*, *R1*, etc.
- *spill_virtual_registers_prefix*: it is a string representing the prefix of the virtual registers which will contains the stack address of the stack spills.
- *spill_virtual_registers_type*: it is a string representing the type of the register spills. It must be a LLVM IR first-class type, and by default it is set to *i32*.
- *param_regs_count*: it is an integer value representing the number of registers available for passing parameters to functions. For example, for the MSP430 architecture [2] it is 4.

6.6.4 Memory Configuration

memory_configuration is the variable specifying the configuration of the memory. It consists in a python dictionary that permits configuring which memories are available and what content they have.

The configuration of a single memory consists in a python dictionary with the following keys:

- *enabled*: it is a Boolean value which states if the memory is enabled or not.

- *stack*: it is a Boolean value which states if the stack is allocated inside the represented memory.

- *heap*: it is a Boolean value which states if the heap is allocated inside the represented memory.

- *gst*: it is a Boolean value which states if the represented memory can contain global variables.

- *gst_prefix*: it is a string representing the address prefix of the gst present in this memory. If this memory does not contain any global variable, it is ignored.

- *gst_base_address*: it is an integer value representing the relative address at which the *GST* that is allocated in this memory starts. For example, if the prefix is *GST* and base address is *1*, the first memory cell allocated in the global symbol table has the address *GST-0x01*.

The keys of the *memory_configuration* variable are:

- *sram*: it must contain the single memory configuration for the static RAM.

- *fram*: it must contain the single memory configuration for the NVM, that is the Ferroelectric RAM (FRAM).

- *base_addresses*: it is a dictionary containing two keys, which are heap and stack. The associated value to each key is an integer representing the relative address at which the respective memory section starts.

- *prefixes*: it is a dictionary containing two keys, which are heap and stack. The associated value to each key is a string representing the address prefix of each memory section.

- *gst*: this key configures the default memory for the GST. It also permits to specify the section parameter present in the code that `ScEpTIC` uses to address a global variable into the non-default memory.

  It is a dictionary with the following keys:

  - *default_ram*: it is a string which can be set either to *SRAM* or *FRAM*. All the global variables will be allocated into the memory specified by this key.

  - *other_ram_section*: it is a string representing the name of the attribute *section* that we can use in the source file to allocate global variables into the non-default memory. By default, it is set to *.TI.persistent*, which is the memory section that the MSP430 [2] compiler uses for addressing elements into the FRAM.

  Let us suppose we want to allocate a variable called *my_var* in the FRAM, and we want to use SRAM as default memory. We must set *default_ram* to *SRAM*. Then, in our code we declare the variable as:

```
int my_var __attribute__((section(".TI.persistent")));
```

- *address_dimension*: it is an integer value representing the dimension in bits of an address.

6.6.5 Checkpoint Configuration

checkpoint_mechanism_configuration is the variable that permits us to specify the configuration of the checkpoint mechanism. It consists in a python dictionary, and permits us configuring the behavior of the checkpoint mechanism for each memory section.

The configuration of the checkpoint of a single memory consists in a dictionary with the following keys:

- *restore_stack*: it is a Boolean value that states if the stack in the represented memory should be saved and restored or not. If no stack is allocated in the represented memory, this key is ignored.

- *restore_heap*: it is a Boolean value which states if the heap in the represented memory should be saved and restored or not. If no heap is allocated in the represented memory, this key is ignored.

- *restore_gst*: it is a Boolean value which states if the global symbol table in the represented memory should be saved and restored or not. If no gst is allocated in the represented memory, this key is ignored.

The keys of the *checkpoint_mechanism_configuration* variable are:

- *checkpoint_placement*: it is a string representing how checkpoint are placed. We can set it equal to *dynamic* or *static*. In this last case, `ScEpTIC` expects checkpoint calls to be placed inside the LLVM IR code.

- *on_dynamic_voltage_alert*: it is a string representing the action taken when a low power interrupt is generated. It can be set to:
  - *continue*: after a low power interrupt is generated, a checkpoint will be taken and the execution will continue until *execution_dept* instructions are executed.

  - *stop*: after a low power interrupt is generated, a checkpoint will be taken and the execution will stop.

  If the checkpoint placement is not set to *dynamic*, this variable is ignored.

- *checkpoint_routine_name*: it is a string representing the name of the checkpoint routine which is present in the code.
- *restore_routine_name*: it is a string representing the name of the restore routine present in the code.
- *sram*: it is the configuration of the checkpoint mechanism for the static RAM.
- *fram*: it is the configuration of the checkpoint mechanism for the Ferroelectric RAM (FRAM).

6.6.6 Pre-defined Environment Configurations

We can configure the overall system structure from scratch, as we explained in the previous sections, or we can select a pre-defined system configuration. In this last case, `ScEpTIC` automatically configures the register file, memory and checkpoint mechanism.

For doing so, we can set the value of the variable *system*, that represents the system configuration to be adopted. If we set it to *custom*, `ScEpTIC` will use the configuration that we set in the previously described variables, otherwise it will use the pre-defined settings defined of the specified system.

If we use a pre-defined system, we still require to specify the names of checkpoint-related routines, prefixes and base addresses.

The available system configurations are the following:

- *MementOS* [3]
  - *checkpoint type*: static
  - *memory section placements*: stack in SRAM, heap in SRAM, gst in both SRAM and FRAM
  - *restored memory sections*: whole SRAM (stack, heap and gst)
- *Hibernus* [11]
  - *checkpoint type*: dynamic
  - *on power interrupt*: stop
  - *memory section placements*: stack in SRAM, heap in SRAM, gst in both SRAM and FRAM
  - *restored memory sections*: whole SRAM (stack, heap and gst)

- *DINO* [1]
  - *checkpoint type*: static
  - *memory section placements*: stack in SRAM, heap in SRAM, gst in both SRAM and FRAM
  - *restored memory sections*: whole SRAM and only gst for FRAM

  Note that FRAM is considered to be saved and restored by checkpoints, since DINO applies versioning of the variables in NVM to prevent inconsistencies.

- *Ratchet* [10]
  - *checkpoint type*: static
  - *memory section placements*: stack in FRAM, heap in FRAM, gst in FRAM
  - *restored memory sections*: None

- *QuickRecall* [12]
  - *checkpoint type*: dynamic
  - *on power interrupt*: stop
  - *memory section placements*: stack in FRAM, heap in FRAM, gst in FRAM
  - *restored memory sections*: None

To use one of the listed systems, we must set the *system* variable equal to the configuration name written in lowercase. For example, if we want to use DINO, we must set `system = 'dino'`.

We can add pre-configured systems to `ScEpTIC` by creating a new python file containing the system configuration. Then, we must place such file into the folder *ScEpTIC/emulator/configurator*, that already contains the available systems. For example, if we want to create a system called *mytest*, we need to create the file mytest.py and then we have to put it into the specified directory. Now, we can use the new pre-defined system by setting the configuration variable *system* to *mytest*.

Chapter 7

Testing Mechanisms Implementation

7.1 Data Interruption Manager

In this section we present the **DataInterruptionManager**, that implements Algorithm 4 which we described in Section 4.5. Such algorithm analyzes the effects of *Memory Inconsistencies* and finds their presence.

The *DataInterruptionManager* extends the base *InterruptionManager* class that we described in Section 6.4.6. For identifying *Data Access Inconsistencies* and *Memory Map Inconsistencies*, it exploits the *lookup table* and *memory map table* that the *VirtualMemoryCell* class directly implements. Instead, for identifying the presence of *Activation Record Inconsitencies* it exploits the method that the base *InterruptionManager* implements.

The following sections describe the work-flow of the analysis that the *DataInterruptionManager* implements.

7.1.1 Lookup and Memory Map Tables

The *VirtualMemoryCell* class contains the lookup and memory map table elements associated with the memory cell it represents. We made this implementation choice because each memory cell is an instance of this class, independently of the memory section it belongs.

All the support memory that the analysis requires resides inside the *lookup* variable, which is a dictionary containing the following keys:

- **old_content**: the associated value represents the old value of the memory cell, and it is part of the content represented by both the lookup table and the memory map table.
- **write_global_clock**: the associated value represents the logical clock at which the memory cell was written, and it is part of the content represented by the lookup table.

- **memory_mapped**: the associated value represents the logical clock at which the memory cell was mapped, and it is part of the content represented by the memory map table.
- **old_memory_address**: the associated value represents the previous address at which the memory cell was mapped, and it is part of the content represented by the memory map table.
- **old_dimension**: the associated value represents the previous dimension of the memory cell, and it is part of the content represented by the memory map table.
- **write_pc**: the associated value represents the program counter in which the memory cell is written. This data is used for signaling to the user where an inconsistency may happen.
- **memory_mapped_pc**: the associated value represents the program counter in which the memory cell is mapped. This data is used for signaling to the user where an inconsistency may happen.

The *lookup* variable of the object is automatically updated and verified each time the memory cell is accessed. Updates to the *lookup* variable of a *VirtualMemoryCell* can happen with two methods: *set_lookup()* and *set_memory_mapped()*.

- **set_lookup()**: it takes as arguments the old content of the memory cell, the current program counter, and the current logical clock. This method updates accordingly the *lookup* variable. Furthermore, it has a fourth optional argument called memory_mapped, which if is set to *True* initializes also the memory map part of the *lookup* variable. It is used when a memory cell is allocated the first time.
- **set_memory_mapped()**: it takes as arguments the current program counter, the current logical clock, the old memory address, and the old dimension. This method updates accordingly the *lookup* variable.

Whenever a *write()* method access a *VirtualMemory* object, it automatically updates accordingly the *lookup* variable.

The *VirtualHeap* class automatically updates the part of the *lookup* variable representing the memory map table, after it performs an action over the heap:

- **allocate()**: after the memory cell is allocated into the heap, this method calls the *set_lookup()* over it with the *memory_mapped* argument set to *True*. In this way, the memory map table part of the *lookup* variable is initialized.
- **reallocate()**: after the group of memory cells is reallocated, this method update the lookup variable of each memory cell of the group using the *_update_group_lookup()* method present in the *VirtualHeap* object. Such method calls the *set_memory_mapped()* method of each cell that is part of the reallocated memory group.
- **deallocate()**: after the memory cell is set to garbage, this method calls the *set_memory_mapped()* over it.

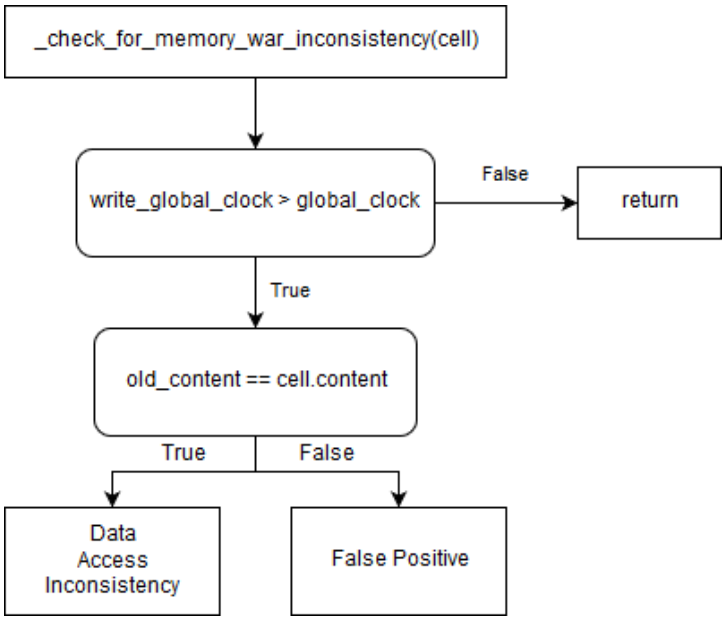

Figure 7.1: Flow chart representing how the lookup table helps in finding memory access inconsistencies over a memory cell.

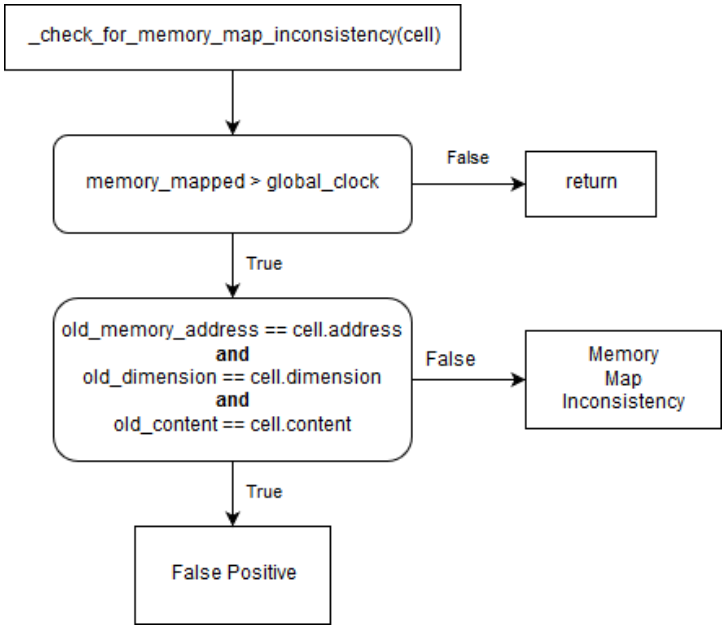

Figure 7.2: Flow chart representing how the lookup table helps in finding memory map inconsistencies over a memory cell.

Data Access Inconsistencies and Memory Map Inconsistencies are verified with the help of two methods that are internally implemented in the *DataInterruptionManager*. They both take as argument the memory cell on which we want to verify the presence of an inconsistency, and are:

- **_check_for_memory_war_inconsistency()**: it verifies the presence of a data access inconsistency on the target memory cell. Figure 7.1 shows how this process happens, and its steps are:

  1. The method verifies if *write_global_clock* is higher than *global_clock*. If the condition is not met, the method simply returns, since there is no inconsistency on the target memory cell. Otherwise, the memory cell was written by a "future" operation, and we require further analysis to establish if an inconsistent value is present.
  2. The method compares *old_content* with the cell's content. If they are equal, there exists a false-positive inconsistency over the considered memory cell. Otherwise, there exists an actual data access inconsistency over the memory cell.

  Note that we called the function *_check_for_memory_war_inconsistency()* because *Data Access Inconsistencies* can happen only if there is a memory **W**rite operation **A**fter a memory **R**ead operation over the same memory cell.

- **_check_for_memory_map_inconsistency()**: it verifies the presence of a memory map inconsistency on the target memory cell. Figure 7.1 shows how this proces happens, and its steps are:

  1. The method verifies if *memory_mapped* is higher than *global_clock*. If the condition is not met, the method simply returns, since there is no inconsistency on the target memory cell. Otherwise, the memory cell was mapped by a "future" operation, and we require further analysis to establish if an inconsistent value is present.
  2. The method compares *old_memory_address* with the current cell's address, *old_dimension* with the current dimension of the memory cell, and *old_content* with the current content of the memory cell. If any of these values is different, then we find a memory map inconsistency. Otherwise, it is a false-positive, since the memory cell was mapped by a future instruction, but it was not changed.

Whenever we access a memory cell, the associated *VirtualMemory* object automatically verifies if a memory map inconsistency is present. When a *read()* or *write()* methods is called, they access the corresponding memory cell using the *_get_cell()* method that the *VirtualMemory* object exposes. It retrieves the requested memory cell and, before returning it to the requesting operation, it calls the *_check_for_memory_map_inconsistency()* method.

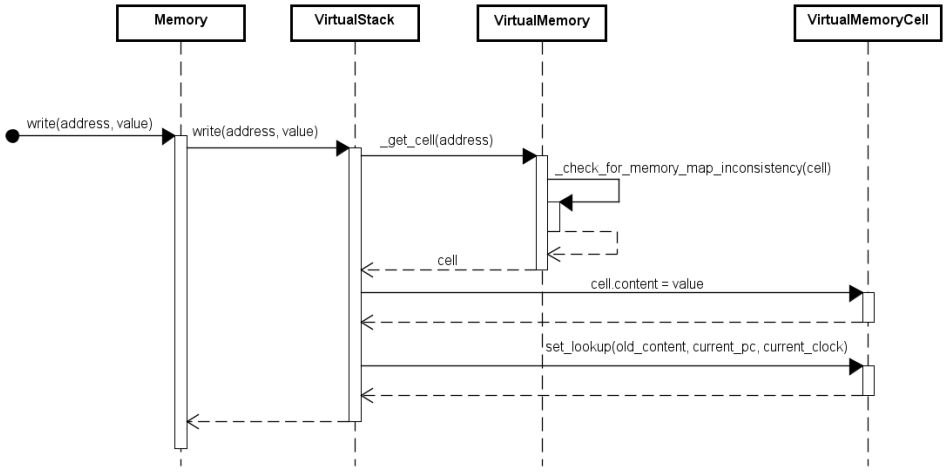

Figure 7.3: Sequence diagram representing the operations executed for performing a memory write operation.

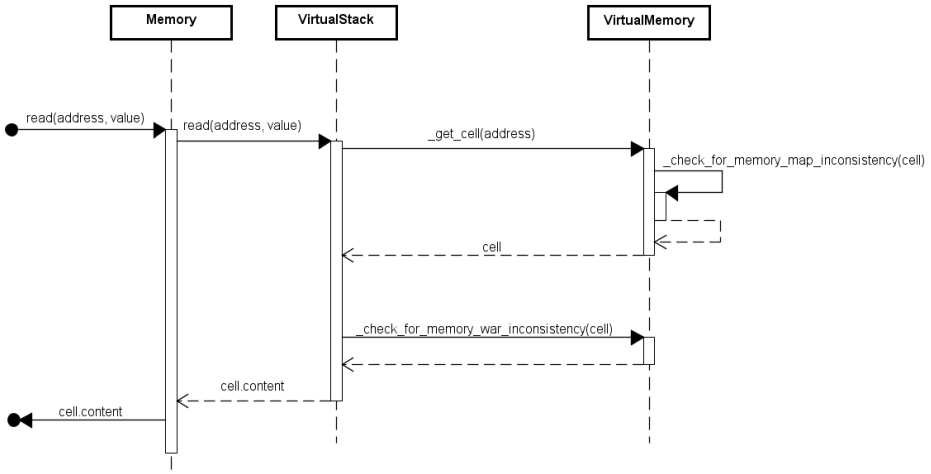

Figure 7.4: Sequence diagram representing the operations executed for performing a memory read operation.

Instead, the presence of data access inconsistencies is verified automatically whenever the *read()* method is called. In fact, before returning the cell's value, this method automatically calls *_check_for_memory_war_inconsistency()* over the retreived *VirtualMemoryCell*, so to verify the presence of data access inconsistencies.

Figure 7.4 and Figure 7.3 show respectively the execution flow of memory reads and writes.

Finally, we represent data access inconsistencies using a *WARInconsistency* object, and memory map inconsistencies using a *MemoryMapIncon-*

sistency object. Both these two classes extend the *DataInconsistency* class, that keeps track of the cell in which the inconsistency happened, the current program counter and if it represents a false-positive.

All the described controls and updates on the *lookup* variable are not tied up to a specific *InterruptionManager*, since they are implemented in the memory sub-module. Furthermore, all those controls can be enabled or disabled by setting accordingly the value of the *do_data_inconsistency_check* variable defined in the implementation of the *InterruptionManager*.

7.1.2 Activation Record Checks

As we stated in Section 4.3, an *Activation Record Inconsistency* is a particular case of a *Data Access Inconsistency*. The elements that we previously described are able to find such inconsistency without requiring any modifications. Unfortunately, they do not provide enough information for identifying which function call may cause the inconsistency, and on which kind of data (i.e., arguments, return address, saved registers, etc.). For this reason, `ScEpTIC` implements a *has_stack_activation_record_inconsistencies()* method inside the base *InterruptionManager*, so that it can be accessed by different analysis. For example, the analysis for verifying intermittence-based inputs exploits such method for preventing cases in which the execution flow crashes or performs unwanted jumps due to *Activation Record Inconsistencies*.

The *has_stack_activation_record_inconsistencies()* method exploits a data structure called *function call lookup*, which consists in a python dictionary that tracks the address of each activation record element for the current function call. It is updated automatically whenever a *CallOperation* happens. The *run()* method that the *CallOperation* class implements sets the address of each argument in the function call lookup. Then, it saves the address of the memory cells containing the stack base pointer and the return address. If we configure `ScEpTIC` to run with physical registers, before

Stack Growth ↑

Address	Content	Info	
0xFCEF	...		← SP, BP
0xFCF0	0xFCF7	Saved stack base pointer	
0xFCF1	@main: #3	Saved return address	
0xFCF2	3	1^{st} argument	
0xFCF3	5	2^{nd} argument	
0xFCF4	9	3^{rd} argument	
0xFCF5	R_0 Value	Saved register	
0xFCF6	R_1 Value	Saved register	
0xFCF7	...		← old BP

Figure 7.5: Example of a stack activation record.

every *CallOperation* it inevitably runs a *SaveRegsiterOperation* that was positioned during the register allocation phase. Such operation saves the registers onto the stack, and thus saves the addresses of the used cells into the function call lookup.

Let us suppose that we are executing a simple program and at line 3 of the *main* function we have the function call *my_func(3, 4, 9)*. Furthermore, let us suppose that we are required to save the value of register R_0 and R_1. Figure 7.5 represents the stack activation record of such function call. `ScEpTIC` populates the *function call lookup* with the content shown in Figure 7.6.

Moreover, whenever a checkpoint happens, `ScEpTIC` saves also the *function call lookup*, which is then restored alongside with the checkpoint, so to keep consistent the mappings of the *function call lookup* with the restored execution state.

Whenever a checkpoint is restored, the *DataInterruptionManager* calls the *has_stack_activation_record_inconsistencies()* method, so to verify if the computation that is going to happen uses a consistent state. This method accesses the *function call lookup*, and it calls the *_stack_check_at_address()* method over each item present in it. Figure 7.7 shows the work flow of the *_stack_check_at_address()* method, which perform the following operations:

1. It gets the *VirtualMemoryCell* object present in the dump at the considered address.

Element Type	**Element Address**	**Info**
Register	0xFCF6	Address in which the value of R_1 is saved.
	0xFCF5	Address in which the value of R_0 is saved.
Argument	0xFCF4	Address in which the 3^{rd} argument is saved.
	0xFCF3	Address in which the 2^{nd} argument is saved.
	0xFCF2	Address in which the 1^{st} argument is saved.
PC	0xFCF1	Address in which is saved the return address.
EBC	0xFCF0	Address in which is saved the old stack base pointer.

Figure 7.6: Content of the function call lookup in Figure 7.5.

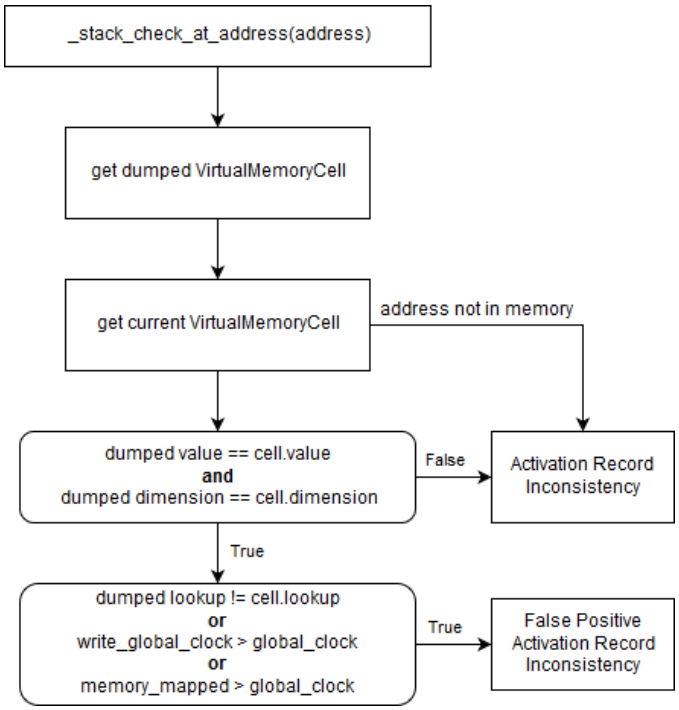

Figure 7.7: Flow chart representing the work flow for recognizing an activation record inconsistency of the *_stack_check_at_address()* method.

2. It gets the *VirtualMemoryCell* object present in the current memory state at the considered address. If no cell is allocated at the requested address, it sets an activation record inconsistency.

3. If the content or the dimension of the two considered cells is different, it sets an activation record inconsistency, since the value was changed by a "future" operation.

4. It accesses the *lookup* variable of the current memory cell, that contains the associated lookup table. If *write_global_clock* or *memory_mapped* elements are higher than the *global_clock*, it sets a false-positive activation record inconsistency. In fact, if such discrepancy is present, it means that a "future" operation rewritten the same value, and thus a false-positive inconsistency must be reported.

The class *StackARInconsistency* represents activation record inconsistencies, and contains where the checkpoint happened and the list of inconsistent elements. The *SingleStackARInconsistency* class represents each single inconsistent element, and it keeps track of the expected value, the real value, the element type, and if it represents a false-positive inconsistency.

7.1.3 Checkpoints Coverage

As we discussed in Section 6.4.6, each test must implement two methods: *intermittent_execution_required()* and *run_with_intermittent_execution()*.

The *intermittent_execution_required()* tells us if we require invoking the *run_with_intermittent_execution()* method to test the current program state. It uses three elements for establishing such information:

- *ignore_pc*: it is a list defined in the *DataInterruptionManager* which keeps track of the program counter associated to the checkpoints that may generate inconsistencies. It is populated during the intermittent execution, and it helps in avoiding the re-execution of the same tests over a checkpoint position that already returned inconsistencies.

- *do_checkpoint_on_power_interrupt*: it is a variable defined in the *DataInterruptionManager*. If it is set to *True*, a checkpoint is generated whenever an interrupt indicating a low power state happens (i.e., dynamic checkpoint mechanism). Instead, if it is set to *False*, we are analyzing a static checkpoint mechanism, or a dynamic checkpoint mechanism configured to stop the execution whenever an interrupt is generated. This last behavior is the one of QuickRecall [12], and it can not present any inconsistency, since no further computation is done after a checkpoint.

- *_consider_address_space()*: it is a private method implemented in the *DataInterruptionManager*, which takes as arguments an address and a variable stating if we are going to perform a memory read or write. This method returns if the address resides in a portion of non-volatile memory:
  - *Memory read*: it returns *True* if the address resides in FRAM, independently on the memory section (i.e., stack, heap, and gst). In fact, any read from FRAM could lead to an inconsistent state.
  - *Memory write*: it returns *True* if the heap is allocated in FRAM and the address resides in it. In fact, no memory write into FRAM can lead to an inconsistent state, unless it is into the heap.

As first instruction, the *intermittent_execution_required()* verifies if the current program counter is inside the *ignore_pc* list. If so, it returns *False*. Otherwise it continues verifying if we must run the current instruction in an intermittent execution scenario. Depending on the checkpoint mechanism we are analyzing, it verifies different conditions over the variable *current_instruction*, which corresponds to the instruction to be run:

- **Static Checkpoint Mechanism**: a static checkpoint mechanism is a particular case of a dynamic one. For this reason, this method simply

verifies if *current_instruction* corresponds to a checkpoint. If so, it returns *True*, otherwise *False*.

- **Dynamic Checkpoint Mechanism**: the method verifies the following conditions:
  1. The variable *current_instruction* corresponds to a *LoadOperation*, and the method *_consider_address_space()* returns *True* over the address targeted by the memory read operation. This condition is required for verifying *Data Access* and *Memory Map Inconsistencies*.
  2. The variable *current_instruction* corresponds to a *StoreOperation*, and the method *_consider_address_space()* returns *True* over the addresss targeted by the memory write operation. This condition is required for verifying *Memory Map Inconsistencies*.
  3. The stack memory section is allocated in FRAM and the current program counter is set to the first instruction of a function different from the main one. This condition is required for verifying *Activation Record Inconsistencies*.
  4. The heap memory section is allocated in FRAM and the variable *current_instruction* corresponds to a *CallOperation* of a function which performs heap mappings (i.e., free and realloc). This condition is required for verifying *Memory Map Inconsistencies*.

  If any of the above conditions is verified, the method returns *True*, otherwise *False*.

  Note that these conditions are the ones we identified in Chapter 4 as the optimal points for analyzing the presence of inconsistencies.

7.1.4 Test Execution Flow

The method *run_with_intermittent_execution()* that the *DataInterruptionManager* implements runs the code in an intermittent execution scenario, and analyzes the state that such execution produces.

This method runs a test for each checkpoint we require testing, and as first operation it saves the correspondent checkpoint. Then, it runs sequentially the code until we reach a point that requires a reset. When such point is reached, the method restores the checkpoint to simulate a power reset and a subsequent restore events. Then, it sequentially tests the presence of inconsistencies by running the code until it reaches the previously considered reset point. If it does not find any inconsistency, it continues the execution until it reaches a new reset point, and it repeats the described test. Otherwise, if it finds an inconsistency, it restores the dump associated with the checkpoint, and it sequentially runs the code until the next reset point is reached. In this way, the state is consistent, and we can continue testing the presence of inconsistencies.

During its work flow, the method *run_with_intermittent_execution()* exploits the following elements:

- *loaded_addresses*: it is a list containing the addresses of each accessed FRAM memory cell.

- *reset_clocks*: it is a list containing all the *global_clock* where `ScEpTIC` generated a reset. In this way, it is able to know if a reset has already been generated over the current state, and thus it can skip its re-generation. Note that we do not use the program counter for this operation, since it would lead to unexpected results in case of cycles or nested function calls.

- *_intermittent_condition()*: it is a method that the *DataInterruptionManager* implements, and it returns whether the intermittent execution analysis should continue. It takes as argument a target global clock, which is considered only if the checkpoint mechanism is dynamic. As first operation, it verifies if the program end is reached, and if so it returns *False*. Otherwise, depending on the checkpoint mechanism, it verifies the following conditions:
  - *Dynamic Checkpoint Mechanism*: it verifies if the current global clock is lower than the target one, which is passed as argument. If so, it returns *True* and the intermittent execution test will continue. Otherwise, it returns *False*, meaning that the execution depth interval has been covered for the considered checkpoint. `ScEpTIC` sets the target global clock when the intermittent execution starts, and calculates it as the sum of the global clock and the execution depth.
  - *Static Checkpoint Mechanism*: it verifies if the instruction to be executed corresponds to a checkpoint. If so, it returns *False* since a new checkpoint is reached. Otherwise, it returns *True* and the intermittent execution test will continue.

- *_test_run()*: this method has two arguments, that are the target global clock and the *loaded_addresses* list. It sequentially runs a sequence of instructions, until it reaches the specified target global clock. If during such execution the method finds an inconsistency, it restores the saved dump and clears the *load_addresses* list.

  Furthermore, *_test_run()* is called just after a reset is generated, and the target global clock that the method takes as argument corresponds to the global clock value when such reset was generated.

  This method has the effect of running sequentially the code from the tested checkpoint to the considered reset point, so to find the presence of inconsistencies in such interval.

The work flow of the *run_with_intermittent_execution()* method consists in the following operations:

1. It firstly saves a checkpoint using the *do_checkpoint()* method that the *CheckpointManager* exposes. It performs such action since we call the *run_with_intermittent_execution()* method only when we must generate and test a checkpoint.

   In fact, with a dynamic checkpoint mechanism, the method *intermittent_execution_required()* returns *True* only if a checkpoint must be tested at the current state. Instead, with a static checkpoint mechanism, it returns *True* only if a checkpoint is reached. For this reason, a checkpoint is saved.

2. It initializes the required data structures:
   - it sets *target_global_clock* to the sum of global clock and execution depth. If the checkpoint mechanism is static, the execution depth is set to 0, and this variable will be ignored.
   - It initializes *loaded_addresses* and *reset_clocks* to empty lists.
   - It sets *restore_at_the_end* to False. The method exploits such variable to establish whether it must restore a dump at the end of the current intermittent test.
   - It sets the *interrupt_global_clock* to the current global clock, and *interrupt_pc* to the current program counter. They correspond respectively to the global clock and the program counter associated to the first instruction after the checkpoint that the method is going to test.

3. The method performs the following operations, as long as the method *_intermittent_condition(target_global_clock)* returns *True*:

   3.1. It sets the *current_instruction* variable equal to the instruction to be run.

   3.2. It sets the *do_reset* variable to *False*. The method exploits this variable to establish if it should generate a reset after the execution of *current_instruction*.

   3.3. If *current_instruction* corresponds to a *LoadOperation*:
      - If *_consider_address_space()* returns *True* over the target address of the *LoadOperation*, then the method appends such address to the *loaded_addresses* variable, so to keep track of the FRAM access.
      - If the *intermittent_execution_required()* method returns *True* and *current_instruction* does not correspond to *interrupt_pc*, then the method sets the *restore_at_the_end* variable to *True*. This is required for not skipping the analysis of multiple loads

inside the considered interval. Note that if this step is omitted, we might not find some inconsistencies.

3.4. It runs the current instruction using the *run_step()* method that the *VMState* object exposes.

3.5. If the variable *current_instruction* corresponds to a *StoreOperation* and its target address is inside *loaded_addresses*. Then, the method sets the *do_reset* variable to *True*, so to reset after the execution of the current instruction. This is required to analyze the presence of a *Data Access Inconsistency*.

3.6. If the heap is allocated into FRAM, and the *current_instruction* variable corresponds to a *CallOperation* of any heap-related function such as malloc or realloc, then the method performs the following operations:

- It sets the *do_reset* variable to *True*, so to reset after the execution of the current instruction. This is required to analyze the presence of a *Memory Map Inconsistency*.
- It sets the *restore_at_the_end* variable to *True*, since after the reset the heap state might be modified, and thus the method will restore a dump at the end of the test.
- It executes the call operation sequentially.

3.7. If the stack is allocated into FRAM, and the *current_instruction* variable corresponds to a *CallOperation*, then it sets the *do_reset* variable is set to *True*. This will generate a reset that permits the method to analyze the presence of *Activation Record Inconsistencies*.

3.8. If *do_reset* is *True* and the current global clock is not present inside the *reset_clocks* list, the method performs the following operations:

3.8.1. It appends the current global clock to the *reset_clock* list, so that a reset is not re-generated at the corresponding position for the current test interval.

3.8.2. It calls the *reset()* method exposed by the *VMState* object, and then restores a checkpoint by calling the *do_restore()* method that the *CheckpointManager* object exposes.

3.8.3. If the stack is allocated into the FRAM, it calls the method *has_stack_activation_record_inconsistencies()*. If such method returns *True*, `ScEpTIC` restores the dump and it stops the current intermittent analysis. This operation is required since the activation record is inconsistent, and the following operations may lead to a crash.

3.8.4. It calls the *_test_run()* method, that manages the re-execution of instructions until the current one is reached.

4. If *restore_at_the_end* is equal to *True*, the method restores the dump.

As we discussed in Section 6.4.6, `ScEpTIC` runs this test using the *run_test()* method that the *VM* class exposes, which we show in Example 6.12. `ScEpTIC` runs the program sequentially until the *intermittent_execution_required()* method returns *True*. Then, `ScEpTIC` calls the *run_with_intermittent_execution()* method, that performs a checkpoint and tests an interval of the program for inconsistencies. `ScEpTIC` repeats this behavior until it reaches the end of the program, resulting in a complete coverage of the code.

7.1.5 Analysis Output

The result that the *DataInterruptionManager* returns consists the list of the found inconsistencies, represented with their respective class. `ScEpTIC` converts such list into a textual representation, and stores it into a *data_inconsistencies.txt* file contained in the directory that we set in the *save_dir* configuration variable.

Let us suppose that we want to test Example 7.1 for data inconsistencies, using a dynamic analysis. As first operation, we generate the LLVM IR associated with the source code, which Example 7.2 shows. Then, we configure `ScEpTIC` with a reasonable execution depth, and we set it to use a configuration similar to the one of Hibernus [11], that consists in a dynamic checkpoint mechanism with the stack and heap allocated in SRAM. Moreover, we set that the computation does not stop after a checkpoint is taken. Note that in the example only the variable *glob_fram* is allocated in FRAM.

Example 7.3 shows the result that the *DataInterruptionManager* produces after it runs the analysis. Such result tells us that there is a *Write After Read Inconsistency*, that it a *Data Access Inconsistency*, at line 6 of the *main* function. For a better interpretation of this result, we can look in the *code* directory that is present in our test result folder. In such directory we have one file for each function, that contains the version of the code of such function that `ScEpTIC` uses for running the analysis.

```
1 int glob_fram __attribute__((section(".TI.persistent"))) = 3;
2
3 int main() {
4     int i;
5     for(i = 0; i < 100; i++) {
6         glob_fram = glob_fram + 1;
7     }
8     return glob_fram;
9 }
```

Example 7.1: Example of code to be tested for data inconsistencies.

```
1  @glob_fram = global i32 3, section ".TI.persistent"
2
3  define i32 @main() {
4      %1 = alloca i32
5      %2 = alloca i32
6      store i32 0, i32* %1
7      store i32 0, i32* %2
8      br label %3
9
10     ; <label>:3:
11     %4 = load i32, i32* %2
12     %5 = icmp slt i32 %4, 100
13     br i1 %5, label %6, label %12
14
15     ; <label>:6:
16     %7 = load i32, i32* @glob_fram
17     %8 = add nsw i32 %7, 1
18     store i32 %8, i32* @glob_fram
19     br label %9
20
21     ; <label>:9:
22     %10 = load i32, i32* %2
23     %11 = add nsw i32 %10, 1
24     store i32 %11, i32* %2
25     br label %3
26
27     ; <label>:12:
28     %13 = load i32, i32* @glob_fram
29     ret i32 %13
30 }
```

Example 7.2: LLVM IR version of Example 7.1.

In our example there is only the main function, and thus we have only the *main.txt* file. Example 7.4 shows its content. From our result we can understand that the memory read happens at instruction number 8 of the main, and the memory write happens at instruction number 10. In those two lines there are respectively a *load* and a *store* operation that target the variable *glob_fram*. Such instructions are the one associated with the inconsistency. Moreover, we are also able to retrieve such information by looking at the LLVM IR, but we must consider only the actual instructions that can be executed. For example, we must ignore the blank lines and the labels. As result, our incriminated operations are at line 16 and 18 of Example 7.2, and they correspond to the operations number 8 and 10 of the *@main*. We can note that the mapping between the code that `ScEpTIC` uses and LLVM IR is trivial, and thus it is easy retrieving the corresponding LLVM IR operations.

```
1  Found inconsistencies: 1
2
3  [Write After Read Inconsistency]
4      Cell address: FGST-0x0
5      Correct content: 3
6      Read content: 4
7      Read at clock: 6
8      Written at clock: 8
9      Memory Read happens at:
10         @main -> #8 (Line: 6; Column: 12; Function name: main;
               File: ./testing_samples/test1.c)
11     Memory Write happens at:
12         @main -> #10 (Line: 6; Column: 12; Function name: main;
               File: ./testing_samples/test1.c)
```

Example 7.3: Results of the data inconsistency analysis over Example 7.1.

The obtained result describes us a possible inconsistent scenario of Example 7.2. Let us suppose that a checkpoint happens before the execution of line 16, and that the MCU shuts down after the execution of line 181, due to a low energy buffer. The execution of line 18 changed the value of *glob_fram* from 3 to 4. When there is enough energy to restart the computation, we restore the stack, the register file and the program counter. Then, we resume the execution form the instruction at line 16, which loads a wrong value of *glob_fram* from the FRAM. In fact, *glob_fram* has a value of 4, that was set by the *future* instruction at line 18 during the previous execution, and differs from the one that such variable had during the checkpoint, that was 3.

We can exploit the notions of Chapter 4 for removing the inconsistency that the analysis found. Firstly, we must choose a static checkpoint mechanism, since there is no way to solve such inconsistency using a dynamic checkpoint mechanism, since we are not able to choose where checkpoints happen. For removing the inconsistency, we should put a checkpoint between line 16 and 18. This operation is possible only if we can modify the LLVM IR, since such lines corresponds to a single source-code level instruction.

If instead we are only able to work on the source code level, we must split the increment at line 6 of Example 7.1 into two operations. We assign the value of *glob_fram*, incremented by 1, to a variable x that is not allocated into FRAM. Then, we place a checkpoint, and finally we update the value of *glob_fram* with the one stored in x. In this way, the load operation that reads the value of *glob_fram* can not access a value produced by a future instruction, and thus we fixed the inconsistency. Example 7.5 shows the resulting code. For verifying if our code actually fixes the inconsistency, we can run another analysis. In this case, `ScEpTIC` does not find any inconsistency, meaning that our fix worked.

```
1  [int 32bit] @main()
2      00: [%0]  %1 = alloca int 32bit x 1
3      01: %2 = alloca int 32bit x 1
4      02: store Value: (int 32bit) 0 in Value: (int 32bit*) %1
5      03: store Value: (int 32bit) 0 in Value: (int 32bit*) %2
6      04: branch None %3 None [Useless]
7      05: [%3]  R0 = load  Value: (int 32bit*) %2
8      06: R1 = cmp slt: Value: (int 32bit) R0 vs Value: (int
           32bit) 100
9      07: branch R1 %6 %12
10     08: [%6]  R0 = load  Value: (int 32bit*) @glob_fram
11     09: R1 = add Value: (int 32bit) R0, Value: (int 32bit) 1
12     10: store Value: (int 32bit) R1 in Value: (int 32bit*)
           @glob_fram
13     11: branch None %9 None [Useless]
14     12: [%9]  R0 = load  Value: (int 32bit*) %2
15     13: R1 = add Value: (int 32bit) R0, Value: (int 32bit) 1
16     14: store Value: (int 32bit) R1 in Value: (int 32bit*) %2
17     15: branch None %3 None
18     16: [%12]  R0 = load  Value: (int 32bit*) @glob_fram
19     17: return Value: (int 32bit) R0
```

Example 7.4: `ScEpTIC` internal representation of Example 7.1.

```
1  int glob_fram __attribute__((section(".TI.persistent"))) = 3;
2  int tmp;
3
4  int main() {
5      int i;
6      for(i = 0; i < 100; i++) {
7          tmp = glob_fram + 1;
8          checkpoint();
9          glob_fram = tmp;
10     }
11     return glob_fram;
12 }
```

Example 7.5: Possible fix to the data inconsistency that Example 7.1 has.

Finally, we can exploit the data analysis that `ScEpTIC` performs for finding the best suited checkpoint mechanism for our program, or for placing checkpoints in a way which grants data consistency. For achieving such scenario, we can set a dynamic checkpoint mechanism and run the analysis. In this way, `ScEpTIC` tests all the possible checkpoint placements, and it returns each inconsistency that our program may present. Depending on the result of such analysis, we can either place checkpoints manually or choose a specific checkpoint mechanism.

7.2 Input Interruption Manager

7.2.1 Overview

In this section we present the *InputInterruptionManager*, that implements Algorithm 7 which we described in Section 5.2. Such algorithm recognizes the access model of each input, and tells us the *Input Access Inconsistencies* that our program may present, if any.

The *InputInterruptionManager* extends the base *InterruptionManager* class that we described in Section 6.4.6. As we stated in Section 5.2, the entire analysis depends on the positions of checkpoints, and thus we can only execute it with a static checkpoint mechanism.

For executing this analysis, we must specify the access model of each input we want to verify. The *InputManager* exposes a *set_consistency_model()* method, that permits setting the access model of a specified input. It takes two arguments:

- The input name, which is the same one that we specify when we create the input.
- The access model of the input. We must express it using one of the two constants that the *InputManager* provides, which are *SAVED* and *MOST_RECENT*.

We defined the *set_consistency_model()* as a classmethod, and thus we can directly call it over the *InputManager* class.

Let us suppose we defined an input named *DHT11*, as explained in Section 6.4.4, and let us suppose that we want to verify if it has a saved access model. To achieve such scenario, we insert in the configuration file the following code:

```
InputManager.set_consistency_model('DHT11', InputManager.SAVED)
```

For finding the access model of each input, the *InputInterruptionManager* exploits different elements:

- An **Input Dependency Table**, which is used for tracking the propagation of input-dependent values inside the memory and registers.

- A **_check_input_lookup()** method, which both *VirtualMemory* and *RegisterFile* classes implement. It verifies if the access model of each input is consistent with the one we specified. Every time `ScEpTIC` accesses a register or a memory cell, they automatically call this method for verifying if an inconsistency is present.

- A **checkpoint_clock** variable, that the *VMState* class stores. It consists in a counter that is incremented every time `ScEpTIC` performs a checkpoint. Such counter permits recognizing the checkpoint interval where an input is accesses.

Furthermore, the *InputPolicyInconsistency* class represents an *Input Access Inconsistencies*. It contains the input name, the measured and required access models, and the program counter of the instruction that uses the input inconsistently with respect to the required access model.

7.2.2 Input Dependency Table

The *Input Dependency Table* maps a memory element such as a *VirtualMemoryCell* or a *Register* to a data structure called *Input Lookup Data*. It keeps track of the name of the inputs on which the value of the memory elements depends on. An *Input Lookup Data* element contains a list of input names, and for each of them it tracks the checkpoint clock value at which they were read. It consists in a python dictionary which maps a checkpoint clock value to the respective list of input names.

The implementation of the *Input Dependency Table* depends on the class of the associated memory element:

- *VirtualMemoryCell*: the *Input Lookup Data* is included in the *lookup* variable, that we already described for the data inconsistencies, and it is under the key *input*.

- *Register*: inside the *RegisterFile* base class there is a *_input_lookup* variable that contains the *Input Lookup Data* of each register. It is a python dictionary that maps a specified register to its corresponding *Input Lookup Data*.

Both the *VirtualMemoryCell* and *RegisterFile* classes expose two method for interacting with their *Input Lookup Data* transparently:

- *set_input_lookup()*: it associates to a memory element the *Input Lookup Data* that we pass as argument.
- *get_input_lookup()*: it returns the *Input Lookup Data* associated to the required memory element.

Note that the methods that the *RegisterFile* class exposes take also as argument the register name, since they are not executed directly over the memory element.

Every time a memory element is modified, `ScEpTIC` updates the *Input Lookup Data* accordingly, and keeps track of the input elements used for computing the altered value. Figure 7.8 shows how such data is updated:

1. **Invalidation of the input lookup data**. When `ScEpTIC` alters a memory element, the associated *Input Lookup Data* is not valid anymore, and thus it must invalidate such lookup data. For this reason, the *write()* methods that both *VirtualMemory* and *RegisterFile* classes expose call the *set_input_lookup()* method over the written element, passing an empty *Input Lookup Data*. As result, they update

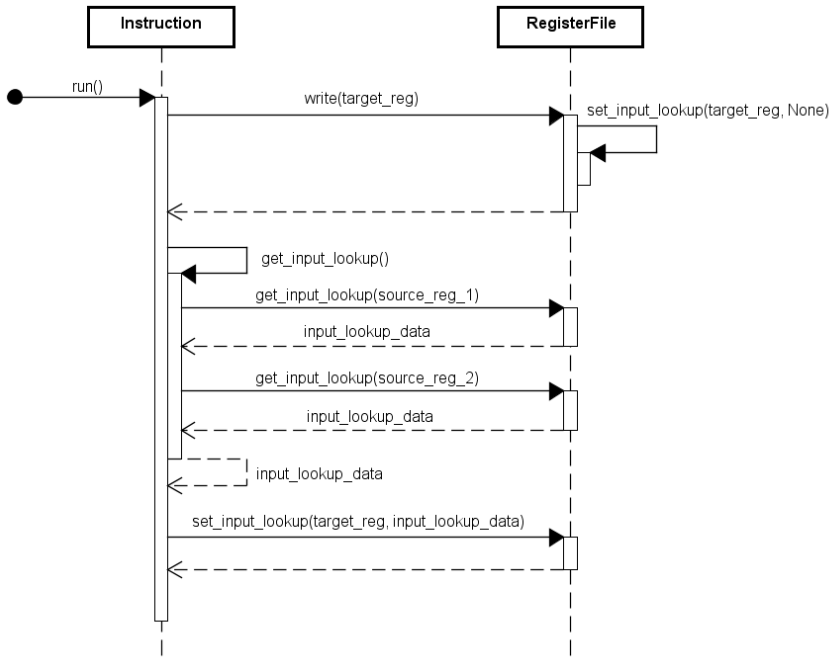

Figure 7.8: Sequence diagram representing how `ScEpTIC` updates the *Input Lookup Data* during the execution of a *BinaryOperation*.

the memory element with the new value, and the input lookup data of such element is emptied.

2. **Input lookup data update**. As described in Section 6.5, each `ScEpTIC` AST instruction extends the *Instruction* base class, and must implement a *get_val()* or *run()* method. Depending on how an instruction is executed, it must implement a way to update the input lookup data:

   - The execution of the instruction is achieved using the method that the *Instruction* base class implements. In this case, the *run()* method exposed by the *Instruction* base class updates the input lookup data. If the executed operation has a target register, the *run()* method retrieves the new input lookup data by calling the *get_input_lookup()* method. Then, it updates the input lookup data associated to the target register, using the *set_input_lookup()* method that the *RegisterFile* exposes.
     The *Instruction* base class provides an implementation of the *get_input_lookup()* method which returns an empty dictionary. It is important that each instruction writing a register implements such method, so to return the correct input lookup data. For example, the implementation available in the *BinaryOperation*

```
1 int main() {
2     int a, b, c;
3     a = input1();
4     b = input2();
5     checkpoint();
6     c = input3();
7     return a + b + c;
8 }
```

Example 7.6: Example of two input accesses

```
1 define i32 @main() {
2     %1 = call i32 @input1()
3     %2 = call i32 @input2()
4     @checkpoint()
5     %3 = call i32 @input3()
6     %4 = add i32 %1, %2
7     %5 = add i32 %3, %4
8     ret i32 %5
9 }
```

Example 7.7: LLVM IR of Example 7.7.

class gets the input lookup data of its operands and returns their combination.

- The execution of the instruction is achieved using a custom *run()* method. In this case, the implemented *run()* method must also take into consideration the input lookup data.
  In this category reside input operations. In fact, they are specified as *ScEpTIC built-ins*, and thus they are executed as function calls. When ScEpTIC executes the corresponding *ReturnOperation*, its custom *run()* method implements the creation of new input lookup data, which is then assigned to the target register.

For a better understanding on how ScEpTIC updates the *Input Dependency Table*, let us focus on the execution of Example 7.7, for which Example 7.6 shows the corresponding C version.

We set the initial *checkpoint_clock* to zero, and then we start the execution of the first instruction, that is a call to an input element. When such call returns, ScEpTIC sets the input lookup data of the virtual register *%1* equal to {*0: ['input1']*}. The same action happens for the second instruction, and the input lookup data of *%2* is set to {*0: ['input2']*}. Then, we execute the third instruction, which is a checkpoint, and thus we increment the *checkpoint_clock* to 1. We continue the execution, and ScEpTIC sets the input lookup data of *%3* equal to {*1: ['input3']*}, since the *checkpoint_clock* is now 1. As next operation, we run the *add* instruction of line 6, which uses as operands the virtual registers *%1* and *%2*. For this reason, we combine their input lookup data to obtain the one of *%4*, which is {*'0': ['input1', 'input2']*}. We apply the same behavior to the instruction at line 7, and thus we set the input lookup data of *%5* equal to {*0: ['input1', 'input2'], 1: ['input3']*}.

As we can notice in the described example, the combination of two input lookup data consists in a union of the contained elements: the dictionary

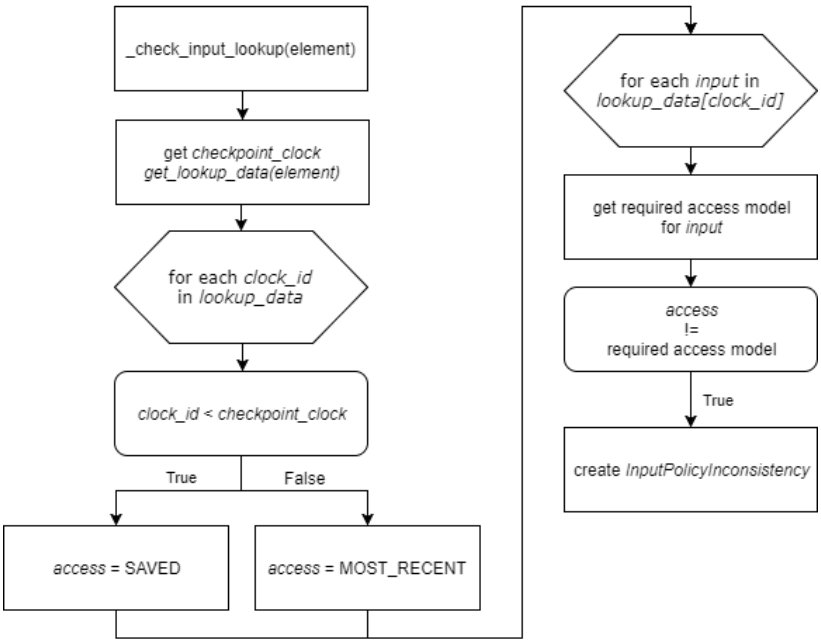

Figure 7.9: Flow chart representing the work flow of the *_check_input_lookup()* method.

keys are combined, and each list with the same key is merged into a single one.

For obtaining the access model of each input, we must verify the *Input Lookup Data* associated to a memory element every time we read it. For permitting such operation, both *VirtualMemory* and *RegisterFile* classes expose a *_check_input_lookup()* method. It takes as only argument the memory element on which we want to verify the presence of an *Input Access Inconsistency*.

The purpose of the *_check_input_lookup()* method is to verify the access model of each input used for computing the value that is stored in the specified memory element. For automatically analyze the presence of *Input Access Inconsistencies*, the *read()* methods that the *RegisterFile* and *VirtualMemory* expose automatically calls the *_check_input_lookup()*.

Figure 7.9 shows the execution flow of the *_check_input_lookup()* method, which consists in the following operations:

1. It gets the current *checkpoint_clock* value and the *input_lookup_data* that is associated to the target memory element.

2. For each key in the *input_lookup_data*, it performs the following operations:

   2.1. It measures the access model. A key corresponds to a value of the *checkpoint_clock*, and if it is lower than the current *checkpoint_clock*, all the inputs associated to the considered key have a *SAVED* access model. Otherwise, they have a *MOST_RECENT* one.

   2.2. For each input associated to the key, it gets the access model that we configured for such input. If such access model is different from the one it measured, it found an *Input Access Inconsistency*.

For better understanding the *_check_input_lookup()* method work flow, let us consider the input lookup data which we previously produced from Example 7.7. Moreover, let us suppose that we are going to execute the *add* instruction of line 7. It reads the value of both *%3* and *%4*, and thus `ScEpTIC` calls the *_check_input_lookup()* method for both of them. The current *checkpoint_clock* is 1, and for this reason the *_check_input_lookup(%3)* measures a *MOST_RECENT* access model over *input3*. Instead, *_check_input_lookup(%4)* measures a *SAVED* access model for both *input1* and *input2*.

7.2.3 Test Implementation

This analysis does not require to generate an intermittent execution, and requires only to propagate the *Input Lookup Data*, so to verify it whenever a memory element is accessed. The implementation of the *InputInterruptionManager* is the following:

- The *intermittent_execution_required()* method always returns *False*, since no intermittent execution is required. The only action it performs consists in verifying if the instruction to be executed is a *checkpoint*. If so, it increments the *checkpoint_clock* variable that the *VMState* object contains.

- The *run_with_intermittent_execution()* method is empty, since this analysis runs the code sequentially, and thus no intermittent execution is required.

7.2.4 Analysis Output

The result that the *InputInterruptionManager* returns consists the list of the found input access inconsistencies. `ScEpTIC` converts such list into a textual representation, and stores it into an *input_inconsistencies.txt* file contained in the directory that we set in the *save_dir* configuration variable.

Let us suppose that we want to test Example 7.6 for input access inconsistencies, with a *SAVED* input access model for every input element.

```
1 InputManager.create_input('input1', 'input1', 'i32')
2 InputManager.set_consistency_model('input1', InputManager.SAVED)
3
4 InputManager.create_input('input2', 'input2', 'i32')
5 InputManager.set_consistency_model('input2', InputManager.SAVED)
6
7 InputManager.create_input('input3', 'input3', 'i32')
8 InputManager.set_consistency_model('input3', InputManager.SAVED)
```

Example 7.8: Configuration of input and output functions.

```
1 [int 32bit] @main()
2     0: [%0]  %1 = call @input1([])
3     1: %2 = call @input2([])
4     2: call @checkpoint([])
5     3: %4 = call @input3([])
6     4: %5 = add Value: (int 32bit) %2, Value: (int 32bit) %1
7     5: %6 = add Value: (int 32bit) %5, Value: (int 32bit) %4
8     6: return Value: (int 32bit) %6
```

Example 7.9: Textual representation of the `ScEpTIC` AST generated from Example 7.7.

Firstly, we create the required inputs in the `ScEpTIC` configuration file, and we set their access model. Example 7.8 shows the code that we insert in the configuration file of `ScEpTIC`. Then, we generate the LLVM IR associated to the source code, which Example 7.7 shows, and we run the analysis.

Example 7.10 shows the result of this analysis, which tells us that there are two input access inconsistencies over the same input named *input3*. The program accesses this input with a *MOST_RECENT* access model, but we required a *SAVED* one.

Example 7.9 represents the code that `ScEpTIC` uses for executing the analysis. The input access inconsistencies happen at the 5^{th} and 6^{th} operations of the *main*, which are both *add* instructions. We can note that these two operation corresponds to C instruction present at line 7, and this is why `ScEpTIC` signals us two inconsistencies.

To better understand this result, let us focus on the execution of C version of the code, that Example 7.6 shows. Let us suppose that we run the code sequentially until we reach the checkpoint at line 5. Now, we retrieve the input value associated to *input3*, by running the instruction at line 6. Let us suppose that, after such operation, a shutdown happens due to a low energy buffer. When there is enough energy to restart the computation, we restore the checkpoint, and we resume the execution from the instruction at line 6, which is re-execute. Such re-execution has the effect of gathering a new input value for *input3*, effectively losing the one we have read before the shutdown. This behavior describes a *MOST_RECENT* access model,

```
1  Found inconsistencies: 2
2
3  [Input Access Inconsistency]
4      Input name: input3
5      Required consistency model: SAVED
6      Measured consistency model: MOST_RECENT
7      Checkpoint happens at:
8          @main -> #2 (Line: 5; Column: 6; Function name: main;
               File: ./input.c)
9      Input Access happens at:
10         @main -> #5 (Line: 7; Column: 19; Function name: main;
               File: ./input.c)
11
12 [Input Access Inconsistency]
13     Input name: input3
14     Required consistency model: SAVED
15     Measured consistency model: MOST_RECENT
16     Checkpoint happens at:
17         @main -> #2 (Line: 5; Column: 6; Function name: main;
               File: ./input.c)
18     Input Access happens at:
19         @main -> #6 (Line: 7; Column: 6; Function name: main;
               File: ./input.c)
```

Example 7.10: Results of the input access inconsistency analysis over Example 7.6.

but we requested a *SAVED* one for this input element.

Considering the result of our analysis, we can exploit the notions we described in Section 5.2 for solving the found inconsistency. We can proceed in three different ways:

1. We move the checkpoint after the call to the input function. This approach will make us move the checkpoint at line 5 after line 6.
2. We move the call to the input function before the checkpoint. This approach will make us move the call to the input function at line 6, before line 5.
3. We create a new checkpoint just after the call to the input function. This approach will make us create a new checkpoint just after line 6.

At a first look, the first two approaches might seem the same, but it is not always possible to move a checkpoint, and in some cases it is simpler moving the input call. For example, let us consider Example 7.11, that represents a modified version of Example 7.6, and thus presents the same input access inconsistency. If we move the checkpoint after line 9, we cause a data access inconsistency on the variable *global_fram*, which is stored in FRAM. In this case we must move the instruction at line 9 before the checkpoint at line 7, so to maintain data consistency.

```
1  int glob_fram __attribute__((section(".TI.persistent"))) = 3;
2
3  int main() {
4      int a, b, c;
5      a = input1();
6      b = input2();
7      checkpoint();
8      glob_fram = a + b;
9      c = input3();
10     b = glob_fram + c;
11     glob_fram = a - 1;
12     return a + b + c;
13 }
```

Example 7.11: Alteration of Example 7.6.

7.3 Interaction Interruption Manager

7.3.1 Overview

In this section we present the implementation of Algorithm 8, that we described in Section 5.3. It profiles different combinations of intermittent executions, and helps us to verify the behavior of intermittence-based inputs.

The *InteractionInterruptionManager* class implements the algorithm for such analysis, and it extends the base *InterruptionManager* class, that we described in Section 6.4.6. As we stated in Section 5.3, the entire analysis depends on the positions of checkpoints, and thus it can only be executed with a static checkpoint mechanism. Furthermore, such analysis exploits the following elements for profiling and tracking the execution of the program:

- An **input_table**: it is a python dictionary that the *InputManager* contains, and keeps track of the value that each input function has.
- An **output_table**: it is a python dictionary that the *OutputManager* containes, and keeps track of the value that each output function has.
- A **profiling** variable, that the *VMState* object contains. It is a python dictionary that associate to each checkpoint, the input and output functions that are executed after it.
- Two markers, named *ProfilingStart* and *ProfilingReset* that we must place in the code we want to test. They are used for producing specific intermittent execution flows.
- A marker named *ProfilingLog* that saves into the output of this analysis the value of a given variable.

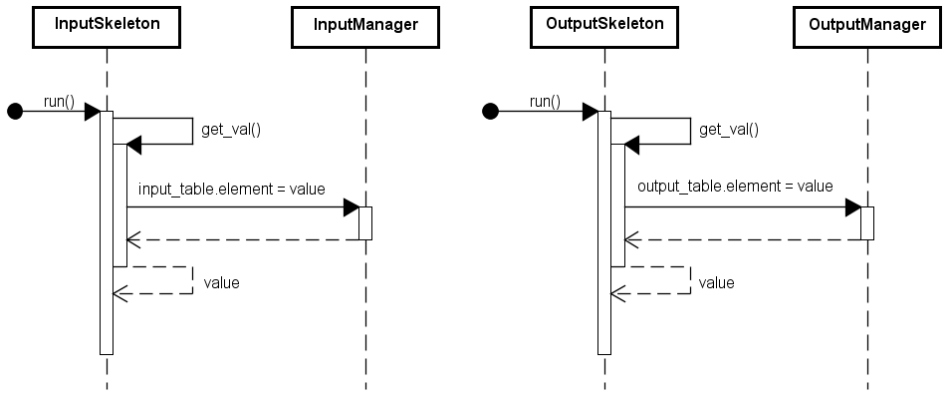

Figure 7.10: Sequence diagram representing the operations that `ScEpTIC` executes for updating the *input_table* and the *output_table*.

Figure 7.10 shows how `ScEpTIC` updates the *input_table* and *output_table* whenever it executes an input and output function. As we stated in Section 6.4.4, `ScEpTIC` treats input and output elements as extensions of the *Instruction* base class. For this reason, the update of input and output tables happen automatically when `ScEpTIC` calls the *run()* method of the *InputSkeleton* or *OutputSkeleton*, that are the classes representing input and output elements.

7.3.2 Profiling Start, Reset, and Log

This analysis requires generating specific combinations of intermittent executions, and thus we must specify the corresponding sequence of shutdowns that cause them. The combination of intermittent executions we need to analyze depends on the source code and on the effects we want to verify. For this reason, we must specify the boundaries where `ScEpTIC` profiles the execution, and where it generates power resets. We define such boundaries as *checkpoint intervals*, that consist in the subset of instructions contained between two checkpoints.

For specifying such boundaries, `ScEpTIC` makes available to us two C functions, that we can put in our source code:

- **profiling_start**(*int resets_count*): we place such function call before a checkpoint, and we use it as a marker to indicate that `ScEpTIC` must profile the following checkpoint interval. This function takes as argument an integer representing the number of power resets that `ScEpTIC` has to generate during the profiling of the marked checkpoint interval.

- **profiling_reset(***int reset_id***)**: we place such function call in the point where we want that `ScEpTIC` performs a power reset. When `ScEpTIC` starts profiling a checkpoint interval, it initializes a *reset clock*. Every time it generates a power reset, it also increments such *reset clock*.

  The *profiling_reset* function takes as argument an integer that represents the value of the reset clock at which `ScEpTIC` should generate the reset in the point marked by the *profiling_reset*.

  Note that if this argument is set to *-1*, a reset will happen whenever `ScEpTIC` encounters this function call. Instead, if `ScEpTIC` does not encounter a *profiling_reset* associated to the current value of the reset clock, it generates a power reset when it reaches the end of the checkpoint interval it is profiling.

Furthermore, we may want to analyze the value that a given variable assumes during the execution of our test. For doing so, `ScEpTIC` permits us to use the **profiling_log(***char* name[]*, *variable***)** function inside our source ode. The *name* parameter is the identifier that `ScEpTIC` uses for saving the variable inside the result. We can set it equals to the actual name of the variable, or we can set it equals to an arbitrary string. Instead, the *variable* parameter represents the variable we want to log. During the execution of the analysis, when `ScEpTIC` encounters a *profiling_log()* function call, it saves the variable inside the *profiling* dictionary that contains the execution trace of the analysis.

It is important to note that `ScEpTIC` does not treat *profiling_start()*, *profiling_reset()*, and *profiling_log()* as function calls, and thus they do not interfere with the program execution. During the creation of the `ScEpTIC` AST, `ScEpTIC` converts them to a node represented by the class *ProfilingStart*, *ProfilingReset*, or *ProfilingLog*. Such classes do not execute any operation, and they just contain a variable representing the associated parameter, which is *resets_count* for *ProfilingStart*, *reset_id* for *ProfilingReset*, and *name* and *variable* for *ProfilingLog*.

To better understand how we can use these functions for analyzing an intermittent execution, let us consider Example 7.12 and let us suppose that the *output_lock* variable is allocated into FRAM. Such variable is a *intermittence-based input*, that allows the program to execute the output function *output1* at most once.

For testing this behavior, we must place a *profiling_start()* before the checkpoint at line 9. Since we want to verify that *output1* is called at most once, we can set the reset number of the *profiling_start()* to 2. Moreover, we may be also interested in verifying the value that the variable *output_lock* assumes at each re-execution. For doing so, we put a call to the function *profiling_log('output_lock', output_lock)* after the checkpoint at line 9. Then, we can put a *profiling_reset(0)* after the *output1* function. `ScEpTIC` will generate the second reset at the end of the checkpoint interval, which is

```
1  int output_lock = 0;
2
3  int a, b, c;
4
5  int main() {
6      a = input1();
7      b = input2();
8
9      checkpoint();
10     c = a + b;
11     if(output_lock == 0) {
12         output_lock = 1;
13         output1(c);
14
15     }
16     c = c - 1;
17     checkpoint();
18     return c;
19 }
```

Example 7.12: Example of a program exploiting an intermittence based input.

```
1  int output_lock = 0;
2
3  int a, b, c;
4
5  int main() {
6      a = input1();
7      b = input2();
8      profiling_start(2);
9      checkpoint();
10     profiling_log('ol',
           output_lock);
11     c = a + b;
12     if(output_lock == 0) {
13         output_lock = 1;
14         output1(c);
15         profiling_reset(0);
16     }
17     c = c - 1;
18     checkpoint();
19     return c;
20 }
```

Example 7.13: Placement of profiling functions in Example 7.12.

before the checkpoint at line 17. Example 7.13 shows the resulting code that permits us to run such analysis.

7.3.3 Test Implementation

This analysis requires to generate an intermittent execution scenario only for the *checkpoint intervals* that we marked for profiling. For this reason, the implementation of the *intermittent_execution_required()* method returns *True* only if the next instruction that `ScEpTIC` will run is a *ProfilingStart*, otherwise it returns *False*.

When the method *intermittent_execution_required()* returns *True*, `ScEpTIC` invokes the *run_with_intermittent_execution()* method. It performs the following operations:

1. It verifies that the current instruction is a *ProfilingStart* operation, and that the next one is a checkpoint operation. If such conditions are not respected, it raises a *RuntimeException*, because the position of the *ProfilingStart* operation is not correct. In this case, we must move the *ProfilingStart* operation in the correct place, that is before a checkpoint, and then we must restart the analysis.

2. It initializes the *run_id* variable to 0, which is the reset clock. Then, it sets the variable *total_runs* equals to the *resets_count* parameter of the *ProfilingStart* operation.

3. It saves the checkpoint program counter into the *checkpoint_pc* variable, and the global clock into the *checkpoint_clock* variable. It uses this last variable for differentiating between multiple executions of the same checkpoint. Then, it performs the checkpoint by calling the *do_checkpoint()* method that the *CheckpointManager* exposes.

4. It initializes the *tracking* variable, which is a python dictionary that will contain all the information about the execution of the current checkpoint interval. Its keys correspond to the different values of the reset clock, and they are associated to a list which will be populated with the profiling information. For example, if we set *reset_count* to 2, the method initializes the *tracking* variable to `{0: [], 1:[]}`.

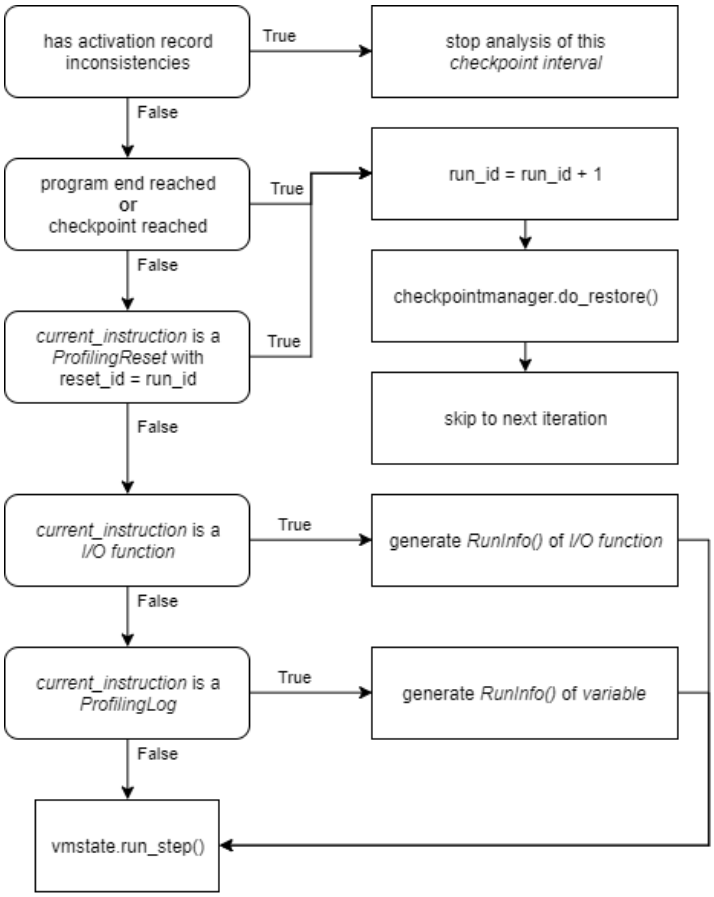

Figure 7.11: Flow chart representing how the *run_with_intermittent_execution()* method executes a single instruction.

5. It runs the operations that Figure 7.11 shows, as long as *run_id* is higher than *total_runs*:

   5.1. It verifies if an *Activation Record Inconsistency* is present, by calling the *has_stack_activation_record_inconsistencies()* method that the base *InterruptionManager* exposes. If so, it is not possible to continue the analysis in the *checkpoint interval*, since this kind of inconsistency is unrecoverable and may lead to a program crash. For this reason, it appends to `tracking[run_id]` a message representing this information. Then, it restores a dump by calling the *restore_dump()* method that the *CheckpointManager* exposes, and ends the cycle.

   5.2. It verifies if it should generate a power reset before executing the current instruction. If so, it increments the *run_id* variable by 1, it calls the *do_restore()* method that the *CheckpointManager* exposes, and then it skips to the next cycle iteration.

   Moreover, for generating a power reset `ScEpTIC` verifies the following conditions:

   - The instruction to be executed is a checkpoint. If so, we have reached the end of the *checkpoint interval*, and thus a power reset is required.
   - The instruction to be executed is a *ProfilingReset* having the *reset_id* equal to the current value of *run_id* If so, the user specified to generate a reset here, and thus `ScEpTIC` must generate it.
   - We have reached the end of the program. If so, we have also reached the end of the *checkpoint interval*, and thus a reset is required.

   5.3. It verifies if the instruction to be run is an input or output function. If so, it performs its execution until the I/O function returns. Then, this method gets the name and value associated to the I/O function, and it generates a new instance of the *RunInfo* class. It appends this instance to `tracking[run_id]`, so to keep track of it.

   5.4. It verifies if the instruction to be run is a *ProfilingLog* operation. If so, it generates a new instance of the *RunInfo* class, which contains the name and value of the variable specified in the *ProfilingLog* parameters by the programmer. It appends this instance to `tracking[run_id]`, so to keep track of it.

   5.5. It runs the current instruction.

6. It stores the information that the *tracking* variable contains into the *profiling* variable of the *VMState* object, saving it with the key *checkpoint_clock*.

To better understand how `ScEpTIC` runs this test, let us try to execute it on Example 7.13. The analysis starts when `ScEpTIC` reaches the *ProfilingStart* instruction of line 8. `ScEpTIC` initializes the *run_id* variable to 0, and *total_runs* to 2, as the *ProfilingStart* instruction specifies. As first operation, we execute the checkpoint of line 9, and then we start the profiling analysis. The first instruction `ScEpTIC` executes is the one at line 10, which is a *ProfilingLog*. It gets the current value of *output_lock* variable and then it generates the associated *RunInfo*. We continue the execution, and we reach the *if* statement of line 12. Since its condition returns *True*, as next instruction we execute line 13, which has the effect of updating *output_lock* to 1. Now, the next instruction we have to execute is an output function. For this reason, we generate the associated *RunInfo* instance, and then we run it. We continue the execution until we reach line 15, which tells us to generate a reset only if *run_id* is equal to 0. Since this condition is met, we increment *run_id* to 1, and we restore the checkpoint. The execution of our analysis restarts from line 10, which logs the value of *output_lock* with the name *ol*, and as next instruction we evaluate the condition of the *if* statement. Since the *output_lock* variable is allocated in FRAM, its value is not restored by the checkpoint, and thus it is equal to 1. Now, the condition of the *if* statement is not met, and we execute the instruction at line 17. The next instruction we are going to execute is a checkpoint, which meets the reset condition. For this reason, we increment *run_id* to 2, and we restore the checkpoint. The execution restarts from instruction at line 10, and follows the same behavior of the previous iteration. When it reaches instruction at line 17, we increment *run_id* to 3 and we restore the checkpoint. Now the analysis stops, since *run_id* is higher than *total_runs*, and we generated the required intermittent execution.

7.3.4 Test Output

The result that the *ProfilingInterruptionManager* returns consists in content of the *profiling* variable. It contains the information about I/O functions executed for each checkpoint interval tested, and the value of the logged variables. `ScEpTIC` converts such list into a textual representation, and stores it into an *interaction_profiling.txt* file contained in the directory that we set in the *save_dir* configuration variable.

Let us suppose we want to run the analysis over Example 7.13, which we already configured with the *profiling_start()* and *profiling_reset()* functions. Firstly, we configure the I/O functions that are present in the source code, as Example 7.14 shows. Then, we generate the LLVM IR of our source file, and we run the analysis.

Example 7.15 shows the result that this analysis returns. It tells us that only one checkpoint interval was tested, and that it is the one starting at the checkpoint of line 10. The content of *Run #0* represents the profiling

```
1 InputManager.create_input('input1', 'input1', 'i32')
2 InputManager.set_input_value('input1', 5)
3
4 InputManager.create_input('input2', 'input2', 'i32')
5 InputManager.set_input_value('input2', 4)
6
7 OutputManager.create_output('output1', 'output1', 'i32', 'void')
```

Example 7.14: Configuration of input and output functions for running the profiling analysis over Example 7.13.

```
1  Observed checkpoint intervals: 1
2
3  Checkpoint @main -> #7 (Line: 9; Column: 6; Function name:
       main; File: ./profile.c)
4  Global clock: 26
5
6      Run #0:
7          [OUTPUT]
8              Name: output1
9              Value: 9
10             Instruction program counter:
11                 @main -> #19 (Line: 14; Column: 7; Function
                       name: main; File: ./profile.c)
12
13         [VAR]
14             Name: ol
15             Value: 0
16
17     Run #1:
18         [VAR]
19         Name: ol
20         Value: 1
21
22
23     Run #2:
24         [VAR]
25         Name: ol
26         Value: 1
```

Example 7.15: Result of the profiling analysis over Example 7.13.

information associated to the first run of the checkpoint interval. Instead, the one that is present after *Run #1* represents the profiling information of the second run of such interval, that is after the first reset, and so on. As we can see, the *output1* function is called only on the first run, and thus our code seems to behave as expected. In fact, *output1* was executed at most once. Also, in each run interval we can see the value of the *output_lock* variable that we decided to log with the name *ol*.

When a user tests intermittence-based inputs, he must generate all the possible reset scenarios so to verify that the code behave as he expects. For this reason, this kind of analysis may require multiple runs. Furthermore, for having a valid result, we must cover all possible execution scenarios, otherwise we might lose some information.

For example, if we want a full coverage of all the possible execution flow scenarios of Example 7.13, we should also generate a reset in such a way that the *output1* function is never executed. To achieve that, we can move the *profiling_reset(0)* of line 15 up of one line in the code. In this way, we can execute the analysis for verifying that *output1* is never executed. The result of the analysis reports *No interactions* to each run of the checkpoint interval. Now, with this second execution of our analysis, we are able to establish that *output1* is executed *at most once*, independently on where a reset happens.

Finally, if the analysis returns a result which does not reflect the behavior we want on our code, it means that we did not consider some execution scenarios, and thus our code must be modified to account for that.

7.3.5 Testing Output Inconsistencies

For testing output inconsistencies we can use Algorithm 9, which we described in Section 5.4. As we stated in Chapter 5, it is a particular case of Algorithm 8, in which we generate one power reset after the execution of any output function. The *InteractionInterruptionManager* implements Algorithm 8, and we can use it to analyze output inconsistencies. For doing so, we can simply set up the analysis in this way:

- We put a *profiling_start(n)* before each checkpoint, with n equal to the number of outputs present in the following checkpoint interval. In this way, we profile every checkpoint interval twice, and we are able to verify the effects of the re-execution of the output elements.
- We put a *profiling_reset(i)* after every output, with i equal to the number of outputs preceding the one we are considering. For example, if we have two outputs, the first one has $i = 0$, and the second one has $i = 1$. In this way, we perform a reset just after the first execution of the output element.

Let us suppose we want to verify how outputs behave in Example 7.16. Example 7.17 shows the corresponding placement of the tracking functions. Once we run the analysis, its result will contain the information about the execution of the output functions, and we can inspect it to verify if the re-execution of the output functions leads to an inconsistent behavior of our program.

The more output we have, the more *profiling_reset()* we must insert in our program. Moreover, if the outputs are inside branches, it gets more

```
1  int a, b, c;
2
3  int main() {
4      a = input1();
5      b = input2();
6      checkpoint();
7      c = a + b;
8      output1(c);
9      output2(a);
10     checkpoint();
11     return c;
12 }
```

Example 7.16: Example of a program exploiting an intermittence based input.

```
1  int a, b, c;
2
3  int main() {
4      a = input1();
5      b = input2();
6      profiling_start(3);
7      checkpoint();
8      c = a + b;
9      output1(c);
10     profiling_reset(0);
11     output2(a);
12     profiling_reset(1);
13     checkpoint();
14     return c;
15 }
```

Example 7.17: Placement of profiling functions in Example 7.16.

complicated identifying the positioning parameter i of the *profiling_reset(i)* function.

This analysis can be completely automatized in a way which does not require the user to insert in its code the *profiling_start()* and *profiling_reset()* instructions. In fact, resets must be generated in fixed points (i.e., after outputs), and we have to test any checkpoint interval. For this reason, and for reducing the code modifications required by the user for performing this analysis, we created an *OutputInterruptionManager* class, which implements only the output analysis and does not require the user to put any additional code inside its program. It implements Algorithm 9, and it consists in a lighter version of the *InteractionInterruptionManager*. As consequence, the *OutputInterruptionManager* has a similar logic with respect to it.

As we stated in Section 5.4, before running our analysis, we must classify our inputs into two categories: *idempotent* and *non-idempotent*. We are interested in analyzing the behavior of *non-idempotent* inputs, since the re-execution of *idempotent* ones can not cause any *output inconsistency*. Once we classified our inputs, we must set in our configuration file the idempotency of the inputs. For doing so, we can call the *set_idempotency*() method that the *OutputManager* exposes, which takes two parameters:

- *output_name*: identifies the output to which set the idempotency category.
- *idempotency*: identifies the idempotency category to be associated with the output. It can be *OutputManager.IDEMPOTENT* or *OutputManager.NON_IDEMPOTENT*

If we do not set the idempotency category of an output, the default behavior of the *OutputManager* is to consider it to be *non idempotent*. We can alter this default behavior in the configuration file by setting *OutputManager.default_idempotent* to *True*, which will make the *OutputManager* to consider outputs *idempotent* by default.

Let us now consider the elements which *OutputInterruptionManager* uses for running the analysis. It exploits the same *output_table* and *profiling* variables we described for the *InteractionInterruptionManager* as data structures. Moreover, the work-flow is very similar, but performs fewer operations.

The *intermittent_execution_required()* method returns always *True*, since we are interested in verifying all the checkpoint intervals. In this way, the *run_with_intermittent_execution()* method runs and analyzes the entire program with the following execution flow:

1. We can enter this method in two cases: at the start of the program, and when a checkpoint is encountered. For this reason, as first operation, it verifies if we are at the start of the program. If so, it generates a checkpoint, so to initialize the first checkpoint interval. Otherwise, it verifies that the current operation is a checkpoint, and performs it.

2. It initializes two tracking dictionaries and a variable:
   - *tracking*: contains the tracking information about the execution of the output functions.
   - *reset_no*: tracks the amount of resets generated after each output routine, so to perform only one reset after each output routines.
   - *resetting_out*: tracks the output routine after which we performed a reset. It is used as index for the tracking.

3. It starts running the program, and performs a subset of the operations we described for the *InteractionInterruptionManager*, which we show in Figure 7.11:

   3.1. `ScEpTIC` verifies if an *Activation Record Inconsistency* is present. If so, it is not possible to continue the analysis for the *checkpoint interval*, since this kind of inconsistency causes the program to crash. For this reason, it appends to `tracking[resetting_out]` a message representing this information. Then, it restores a dump and ends the cycle.

   3.2. It verifies if the current instruction is a call to an output routine which is configured as non-idempotent. If so, it performs the following operations:

      3.2.1. `ScEpTIC` executes the output routine and sets the variable *resetting_out* to the name associated with it.

3.2.2. `ScEpTIC` generates a new instance of the *RunInfo* class, containing the profiling information relative to the execution of this output routine, and appends it to $tracking[resetting\_out]$.

3.2.3. `ScEpTIC` increments the $reset\_no[resetting\_out]$ by one, and verifies if it is equal to 1. If so, no reset was generated after this output routine, and thus it generates a reset and restores the saved checkpoint. Otherwise, it continues the execution.

3.3. It executes the current instruction.

4. The above cycle is executed until we reach the next checkpoint, or if the program ends.

5. When the analysis of the current checkpoint interval ends, `ScEpTIC` appends the *tracking* variable to the $profiling$ dictionary present in the $VMState$ instance.

To better understand how this test is run, let us try to execute it on Example 7.16. Moreover, let us suppose we configured both *output1* and *output2* to be *non-idempotent*. We start the execution, and as first operation we call *intermittent_execution_required*(), which returns *True*. For this reason, we call *run_with_intermittent_execution()*. It generates a checkpoint, since we are at the start of the program, and initializes its local variables. Then, it starts executing the code. In this first checkpoint interval, no output routine is present, and thus it executes the code sequentially, reaching the *checkpoint()* routine at line 6. The method returns, and we call the *intermittent_execution_required*(), which returns *True*. Now we enter *run_with_intermittent_execution()*, which performs a checkpoint, initializes its local variables, and starts executing the code. When it reaches the first output instruction at line 8, it sets *resetting_out* to *output*1. Then, it generates the *RunInfo* associated to the execution of *output*1, and appends it to $tracking[resetting\_out]$. As next operation, it increments $reset\_no[resetting\_out]$, and since it is equal to 1, it generates a power reset. The execution restarts from the instruction at line 7. When we reach the output routine at line 8, we execute the same operations we previously described, except for the reset. In fact $reset\_no[resetting\_out]$ is now 2, and thus we do not reset. As next instruction, we execute the output routine *output*2, and we generate both the associated *RunInfo* and a reset, in the same way we described. The execution restarts, and we finally reach the checkpoint at line 10. Since we reached the end of the checkpoint interval, the method returns. As next instruction, we call the method *intermittent_execution_required*(), which returns *Ture*, and thus we call *run_with_intermittent_execution()*. In this checkpoint interval there are no output routines, and thus the method executes the code sequentially until we reach the end of the program.

```
1  Checkpoint: @main -> #0
2      No output executed
3
4  Checkpoint: @main -> #6 (Line: 6; Column: 2; Function name:
       main)
5
6      Output output1
7          [OUTPUT]
8              Value: 2
9              Execution clock: 32
10             Run id: 0
11             Instruction program counter:
12                 @main -> #12 (Line: 8; Column: 2; Function
                       name: main)
13
14         [OUTPUT]
15             Value: 2
16             Execution clock: 32
17             Run id: 1
18             Instruction program counter:
19                 @main -> #12 (Line: 8; Column: 2; Function
                       name: main)
20
21
22     Output output2
23         [OUTPUT]
24             Value: 1
25             Execution clock: 47
26             Run id: 0
27             Instruction program counter:
28                 @main -> #14 (Line: 9; Column: 2; Function
                       name: main)
29
30         [OUTPUT]
31             Value: 1
32             Execution clock: 47
33             Run id: 1
34             Instruction program counter:
35                 @main -> #14 (Line: 9; Column: 2; Function
                       name: main)
36
37 Checkpoint: @main -> #15 (Line: 10; Column: 2; Function name:
       main)
38     No output executed
```

Example 7.18: Result of the output analysis over Example 7.16.

Example 7.18 shows the result of this analysis. As we can see, it is similar to the one that the *InteractionInterruptionManager* returns, but it only contains information about the execution of output elements. As we can see, *output1* may be executed twice in an intermittent execution

happening inside the checkpoint interval identified by the checkpoint at line 8. In both the executions it sends the value 2 to the environment. A similar case happens for *output2*.

With this result, we are able to identify if the environment interactions leads to an inconsistency. Let us suppose that *output1* increments the position of a servo of x degrees, with x equals to the value sent. As we can see in the result, *output1* is executed twice, and thus the final position of the servo will be 4 instead of 2. Depending on our requirements, we can state if this behavior is expected or not, and correct our code accordingly.

Chapter 8

Evaluation

8.1 Overview

This chapter describes the evaluation of `ScEpTIC` within the different testing aspects analyzed in Chapter 4 and Chapter 5, which are *Data inconsistencies testing*, *Input inconsistencies testing*, *Output inconsistencies testing*, and *Intermittence-based inputs analysis*.

We run all the tests for our evaluation over a system with an Intel i7 2600k, 16 Gb of RAM (DDR3, 1600Mhz, dual-channel configuration), Windows 10 (build 1709), and Python 3.7.2. We compiled our test cases using Clang 5.0.1-4 with LLVM 5.0.

`ScEpTIC` is a tool that focuses its analysis in finding inconsistencies and in analyzing how the program behaves in their presence. At the moment there is no other tool with the same goal, and the ones available mainly focus on providing a debugging environment with energy analysis capabilities.

Furthermore, the algorithms developed for inconsistency-free placements of checkpoints, such as the ones present in DINO [1] and Ratchet [10], are not applicable for finding inconsistencies. In fact, they are not designed for verifying the presence of inconsistencies given a checkpoint placement, but they are designed for positioning the checkpoints in a way which grants data integrity with respect to their memory configuration.

For these reasons, we will consider two different approaches for evaluating `ScEpTIC`:

- A **qualitative** approach, consisting in the analysis of the user intervention required to use and/or modify a different tool for performing the same analysis that `ScEpTIC` covers. For this aim, we want to compare our tool with respect to the available debugging environments, such as EDB [13] and SIREN [17]. They are not designed for the same goal of `ScEpTIC`, which focuses on testing intermittent executions, but they can be adapted for achieving the same objective. For this reason, we use those tools as *baselines* for our qualitative evaluation.

As *qualitative aspects*, we will consider:

- **Tool Alteration**: the minimum required alterations to the debugging tool a user needs to implement, for achieving the same testing results of `ScEpTIC`.
- **User Interactions**: the number of actions a user needs to manually perform, to achieve the same testing results of `ScEpTIC` (e.g. resets, insertion of controls, etc.). Note that `ScEpTIC` requires user manual intervention for specifying input, outputs, and insertion of specific instructions (e.g. *profiling_start()* and *profiling_reset()*) for performing inconsistency-based interaction analysis, as described in Section 6.4.4 and Section 7.3.
- **Effectiveness**: the result obtained, in terms of testing and inconsistencies found, after performing all the required actions and alterations over the debugging environment.

- A **quantitative** approach, consisting in the comparison of the results achieved using a simple and non-optimized algorithm with respect to the one implemented by `ScEpTIC`. Our objective is to evaluate the performance of the implemented algorithms (i.e., Algorithm 4 and Algorithm 7), and thus we will compare them with their equivalent simpler and non-optimized versions (i.e., Algorithm 2 and Algorithm 11), which will be used as *baselines*. To achieve that, we implemented each non-optimized algorithm as an *InterruptionManager* of `ScEpTIC`, so that the only element which differs is the actual algorithm that is applied over the input program, and not the entire architecture. In this way we are able to compare the performance of each algorithm on equal terms, so to find the possible benefits of the implemented optimizations.

  As *quantitative metrics*, we will consider:

  - **Number of executed instructions**: `ScEpTIC` executes instructions over the host machine, and the execution time may vary depending on its available resources. Using only the execution time for measuring performance may lead to inaccurate results, because the measurement will be dependent not only on available resources, but also on other host related aspects, such as the version of Python installed, and the optimizations available in the host architecture running `ScEpTIC`. For these reasons, we will use the total number of instructions executed by `ScEpTIC` over the emulated architecture. In this way, the evaluation done will grant the same results independently of the setup used.
  - **Number of support memory accesses**: as explained in Section 7.1.1 and Section 7.2.2, `ScEpTIC` uses different data struc-

tures for verifying the presence of inconsistencies. Whenever an instruction is executed, `ScEpTIC` can both modify or accesses such data structures to verify if the state is consistent. These memory accesses introduce a testing overhead which is not included within the executed instructions, and the combination of these two metrics grant a correct estimation of the overall resources invested on the host machine to run tests with `ScEpTIC`. Furthermore, we are comparing algorithms which use different approaches to find inconsistencies, and thus they use different data structures or access methods over them. The variation of such elements might not change the number of executed instructions over the emulated architecture, but do certainly change the number of operations performed by `ScEpTIC` over the host machine. Measuring these accesses permits us to have an accurate comparison of the variation in the performance of the used algorithms, as a function of the used support data structures and access methods. For these reasons, we will include the number of support memory accesses as quantitative metric.

- **Number of generated resets**: as explained in Section 4.5.1, we must recreate an intermittent execution scenario for verifying the presence of inconsistencies, by performing resets of the state and restoring a checkpoint. The number of executed instructions is directly dependent on the number of generated resets, and an optimization of where resets are performed leads to an increase of performance. By including this metric with the ones used, we are able to understand if there is a performance drop due to a high or unnecessary number of generated power resets.
- **Number of checkpoints tested**: this metric is relevant only with a dynamic checkpoint mechanism, since in this case a checkpoint could happen at any line of code. The number of executed instructions is also directly dependent on the number of checkpoints tested. By including this metric with the ones used, we are able to understand if there is a performance drop due to a high or unnecessary number of checkpoints tested.
- **Number of restored snapshots**: as explained in Section 7.1, when `ScEpTIC` finds an inconsistency, it might restore a snapshot to continue the analysis, since the runtime state is inconsistent with respect to the equivalent sequential execution of the program. Restoring a snapshot is time-consuming and causes the analysis to restart from a previous point. By looking at the number of snapshots restored we are able to understand if there is a performance drop due to a high or unnecessary number of snapshots restored.

- **Number of inconsistencies found**: the result of the analysis performed by `ScEpTIC` consists in the list of found inconsistencies, which is then analyzed by the user to solve the problems presented by the program. A high number of inconsistencies might cause multiple restorations of snapshots, and considering this metric helps us to understand their correlation. Furthermore, the lack of generalization in the representation of inconsistencies causes an algorithm to show an unnecessarily high number of them, making the work of the user which has to read the result of the analysis harder. On the other hand, if the analysis does not return all the possible inconsistencies, the result shown to the user is incomplete and will not help him to solve all the problems presented by the analyzed program.
- **Execution time**: as we previously stated, the execution time is a host-dependent metric which, if it is used alone, can lead to an inaccurate evaluation of `ScEpTIC` performance. To overcome this problem, we decided to consider also the *Instructions Executed* and the *Support Memory Accesses* metrics, which influences with different weights the execution time. For these reasons, we will not use the execution time as an individual metric for evaluating `ScEpTIC`, but instead we will use it as a *secondary* metric together with other ones. In this way, we can understand and properly weight the effects of an increased or decreased value of different metrics with respect to the amount of time required to run an analysis.

8.2 Memory Inconsistencies

For performing the evaluation of `ScEpTIC` from the point of view of testing *memory inconsistencies*, we are able to apply both a quantitative and qualitative approach.

8.2.1 Baselines

The algorithms which can be used for finding memory inconsistencies are described in Section 4.5, and `ScEpTIC` implements a variant of Algorithm 4. For the quantitative evaluation, we want to evaluate the performance of the algorithm we implemented in `ScEpTIC`, and thus we will compare it with Algorithm 1, which is the most simple algorithm able to find memory inconsistencies.

Unfortunately, this algorithm is extremely inefficient in performing a dynamic checkpoint analysis: it tests a checkpoint placement at every line, and for each of them it generates a reset at every instruction in the interval de-

fined by the execution depth. The resulting number of executed instructions can be calculated with the formula:

$$n_{executed} = \sum_{i=0}^{n_{ops}-1} \left(\sum_{j=1}^{ED(i)} j + 1 \right) - 1 \tag{8.1}$$

where:

$\mathrm{ED(i)} = min(ED, n_{ops} - i + 1)$
n_{ops} = total number of instructions of the program
ED = execution depth set by the user

For understanding this formula, let us suppose that our code has 3 instructions. The test stars by testing a checkpoint before the first instruction, and it generates a reset after all the three instructions. The resulting execution trace is $1 \rightarrow R \rightarrow 1 \rightarrow 2 \rightarrow R \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow R$, where R indicates that we perform a reset, and we verify the state. The resulting number of instructions executed is 6, and this number is calculated by the inner sum of the formula. Then, we have to test a checkpoint placed after the instruction 1, and for doing so we first have to run instruction 1, and then we are able to perform a checkpoint. This single execution is accounted by the +1 which is after the inner sum. These operations are repeated until we reach the end of the program. Once we reach the final instruction, we do not need to re-execute it for testing a checkpoint after it. For this reason, we subtract 1 from the entire sum. The resulting complexity of Equation 8.1 is $O(n_{ops}^3)$.

Independently of the algorithm we use, for verifying the presence of inconsistencies we need to compare the state that a power reset produces with the one that the equivalent sequential execution of the same code produces. Furthermore, when we find an inconsistency, we must restore a snapshot, so to continue the analysis from a consistent state. These operations introduce an overhead that limits the overall speed of the analysis we are performing: the environment we used for evaluating `ScEpTIC` comprehends an Intel i7 2600k with 16 Gb of RAM, and we were able to reach a maximum speed of $1.8 \cdot 10^4$ instructions per second.

Such overhead and the high number of instructions to be executed make not feasible run an analysis using Algorithm 1 with a dynamic execution scenario. In fact, if we consider the lowest execution depth we estimated in Section 8.2.3 (i.e., 2470), and the number of instructions of the smaller benchmark we selected in Section 8.2.4 (i.e., CRC, $5.2 \cdot 10^4$ instructions), the overall number of instructions that the algorithm tests is $1.54 \cdot 10^1 1$. Considering that our environment is able to analyze $1.8 \cdot 10^4$ instructions per second, it would take around 99 days for performing the entire dynamic analysis.

For this reason, for the evaluation of the dynamic analysis we will analytically calculate the overall number of executed instructions, and we will

also compare the performance of `ScEpTIC` with an optimized version of Algorithm 1, which we refer as *Dummy ScEpTIC*. It consists in a modified version of Algorithm 1 to which we applied the following optimizations:

- It tests a checkpoint placement before each *LOAD* operation which read data from NVM.
- It tests a checkpoint placement before the first instruction of each subroutine if the stack is allocated into NVM.
- It generates a reset after each *STORE* operation which saves data into NVM.
- It generates a reset after each *CALL* operation if the stack is allocated into NVM.

Such optimizations are a subset of the ones that `ScEpTIC` implements, and we described them in Section 4.5. They reduce the amount of the time required to perform an exhaustive dynamic analysis by reducing the number of checkpoint to be tested and resets to be generated. Furthermore, it is important to consider that we selected this subset of optimizations because they do not modify the way in which Algorithm 1 recognizes inconsistencies, and instead they only reduce the time required to run the analysis.

Moreover, for comparing `ScEpTIC` performance with the one of the two algorithms, we have created two extensions of the *InterruptionManager* base class: *BaselineDataInterruptionManager* and *DummyScEpTICDataInterruptionManager*. They respectively implement Algorithm 1 and *Dummy ScEpTIC*.

Finally, for a better analysis of our results, we must consider some implementation details shared by both *Baseline* and *Dummy ScEpTIC* algorithms:

1. **Inconsistency representation:** as we explained in Section 4.2, data inconsistencies may happen between two instruction such that the first one reads the same memory location written by the second one. If such sequence is re-executed, the first instruction is the one which uses the inconsistent value produced by the previous execution of the second instruction. One of the key elements of the data analysis performed by `ScEpTIC` is the recognition of such instruction pairs, which it is not possible to be achieved in *Baseline* and *Dummy ScEpTIC*. In fact, for how they verify the presence of inconsistencies, they are not able to properly find which is the instruction reading the inconsistent value, and they only find if an inconsistent value is present. For this reason, they consider the checkpoint generated as the *reading* instruction.

2. **Differential memory comparisons:** for finding data inconsistencies, both *Baseline* and *Dummy ScEpTIC* compare the memory state obtained after the restoration of a checkpoint with the snapshot taken

when such checkpoint was executed. This can lead to the miss classification of inconsistencies, which results in considering every reset point to be inconsistent.

To understand this problem, let us focus on Example 8.1, which consists in the LLVM IR of the loop body of the code shown in Example 8.2. Let us apply the *Baseline* algorithm to find inconsistencies. As first action, we take both a snapshot of the memory state and a checkpoint. Then, we execute the instruction at line 4, and we simulate a reset with the consequent restoration of the checkpoint. Now, as next action, we compare the memory state with the content of the snapshot we previously took. We repeat this operation until we reach the *store* instruction at line 8, which creates an inconsistent memory state. As next step of our analysis, we restore the snapshot, and we run the *load* instruction at line 9. Then, we perform a reset with the consequent restoration of the checkpoint. When we compare the memory states, we find another inconsistency. In this case, such inconsistency is not caused by the operation at which we reset the memory state, but it is caused by the previous one. The lack of recognizing where an inconsistency really happen leads to the miss-classification of this reset point.

```
1  [...]
2  call void @checkpoint()
3  [...]
4  %6 = load i32, i32* @a
5  %7 = load i32, i32* @i
6  %8 = mul nsw i32 %7, 3
7  %9 = add nsw i32 %6, %8
8  store i32 %9, i32* @a
9  %10 = load i32, i32* @a
10 %11 = add nsw i32 %10, 1
11 store i32 %11, i32* @b
12 %12 = load i32, i32* @b
13 %13 = sdiv i32 %12, 7
14 store i32 %13, i32* @c
15 [...]
```

Example 8.1: LLVM version of the loop body of Example 8.2.

```
1  [...]
2  checkpoint();
3  [...]
4  for(i = 0; i < 10; i++) {
5  [...]
6  a = a + i * 3;
7  b = a + 1;
8  c = b / 7;
9  [...]
10 }
11 [...]
```

Example 8.2: Example of a loop.

We modified the implementation of both *Baseline* and *Dummy ScEpTIC* in a way which prevents this problem:

- After we perform the memory comparisons that verifies the presence of inconsistencies, we save a second snapshot of the memory state into a data structure. In this way, we are able to track the memory states across different reset points.
- When we compare the memory state with the snapshot's content, we extract all the elements which differs. Then, we compare the value of such elements with the one they assume in the second snapshot. If they do not differ, the inconsistency is not caused by the current reset point and thus can be ignored (i.e., has been acknowledged by a previous iteration of the algorithm). Instead, if they differ, the current reset point causes an inconsistency.

If we now apply this new behavior to the previous example, when we reach the *store* instruction at line 8 we find an inconsistent state. Then, we save this second snapshot of the memory, we restore the main one, and finally we run the *load* instruction at line 9. Now, when we perform a reset in such point, we compare the memory state, which differs from the one contained in the main snapshot and thus we compare it with the secondary one. Since the value of variable b is the same in such snapshot, the inconsistency is not considered.

We must note that this behavior is not required in `ScEpTIC`, since it is designed to avoid this problem.

3. **Reset points sub-optimization:** if an analysis happens inside a loop body, it is possible that we generate a reset multiple times in the same point (i.e., once for each iteration of the loop). Given how the *Baseline* and *Dummy ScEpTIC* algorithms find inconsistencies and represents them, there is no reason for verifying the same reset point multiple times. In fact, when an inconsistency is found, a tuple (I_1, I_2) is appended to the list of inconsistencies, with I_1 the instruction after which the checkpoint is taken and I_2 the instruction after which the reset is generated. If we perform a loop iteration, and we reach the previously tested reset point I_2, we can either find an inconsistency or not. If the state is not consistent, we generate a tuple (I_1, I_2), which is already included in the list of inconsistencies, and thus we are wasting computational time.

   For this reason, we modified the implementation of *Baseline* and *Dummy ScEpTIC* algorithms so to not retry the same reset points for a given checkpoint, if an inconsistency has already been found for such reset.

   This optimization is only possible because both the *Baseline* and *Dummy ScEpTIC* algorithms are not able to find the operation which reads the inconsistent value, and they are only able to find instructions

which produces inconsistent values inside the memory. In fact, if we apply such optimization to `ScEpTIC`, we loose relevant information.

Let us now focus on how `ScEpTIC` analyzes Example 8.1, which is the LLVM IR version of the loop's body that Example 8.2 shows. When it generates a reset after the execution of the *store* operation at line 11, it finds an inconsistent state. Such inconsistency is not caused by the execution of the instruction after which we generated a reset. In fact, `ScEpTIC` finds the inconsistent pair (I_4, I_8), which corresponds to the *load* and *store* of variable *a*. If we decide to ignore further resets at line 11, we are not able to find the inconsistency (I_{12}, I_{11}), which is tracked between two different loop iterations (i.e., the *load* operation at line 12 happens during the loop iteration x and the *store* operation at line 12 happens during the loop iteration $x+1$). For this reason, `ScEpTIC` does not have the same optimization of reset points of *Baseline* and *Dummy ScEpTIC*.

Finally, one may think that we could ignore further reset points at the instruction I_{11}, which is the one generating an inconsistent state. Even if it might be true for our simple example, in a more complex code this leads to a loss of information. In fact, if we have memory accesses through pointers, we can potentially loose an inconsistency (i.e., different variables accessed through the same pointer) or we can potentially loose information about the memory cells which are inconsistent (i.e., cells of an array accessed through a pointer). For these reasons, we can not ignore reset points in `ScEpTIC`.

8.2.2 Evaluation Inputs

As our evaluation inputs, we must consider three groups of elements:

- **Execution depth**: `ScEpTIC` is able to analyze the presence of memory inconsistencies with both dynamic and static checkpoint mechanisms. Furthermore, as we stated in Section 4.5.1, we can use the dynamic scenario to test all the possible combinations of static checkpoint placements. To run tests in a dynamic scenario, we must specify the *execution depth* parameter, which indicates the maximum number of instructions executed between a checkpoint and the consequent shutdown. Conversely, with a static checkpoint mechanism, `ScEpTIC` does not require such parameter, since checkpoints are explicitly positioned inside the code.

  For this reason, evaluating the performance of the dynamic analysis of `ScEpTIC` requires us to consider in our test inputs also the execution depth, which is set by the user. As we stated in Section 4.5.1, such parameter depends on various elements, such as the energy source and the overall capacity of the energy buffer. For using a representative

value of the execution depth parameter, we will estimate it considering the energy results obtained by dynamic checkpoint mechanisms such as Hibernus [11], Hibernus++ [4], and Mementos [3].

- **Input files** and **memory configuration**: for evaluating `ScEpTIC` we can use the same benchmarks used for the evaluation of checkpoint mechanisms. It is important to note that a checkpoint mechanism works with a fixed memory configuration, and instead `ScEpTIC` can be configured to reflect different ones. In fact, as explained in Section 6.6.4, an user can configure the memory composition with respect to his testing requirements. Furthermore, from the analysis described in Section 4.2, we can infer that a program might present data inconsistencies only if a portion of its data is allocated into NVM. Since we are evaluating the performance of `ScEpTIC` for the analysis of data inconsistencies, we must recreate the conditions for which an inconsistency may happen (i.e., a portion of data in NVM). For this reason, for each input file used for the evaluation of `ScEpTIC`, we will specify a pair of memory configurations which allocate one or more memory sections into NVM. In this way, we are sure that inconsistencies are possible, and we are also able to evaluate the performance of `ScEpTIC` with respect to the variation of elements allocated into NVM.

- **Checkpoint mechanism**: to perform a test with `ScEpTIC`, we must provide not only the input file and the memory configuration, but we are also required to specify the checkpoint mechanism. For this reason, we will also consider as input to our evaluation the checkpoint mechanism configuration.

8.2.3 Execution Depth Estimation

For estimating a realistic value range of the execution depth parameter, we consider as target architecture the MSP430 with FRAM [28], which has also been used by Hibernus [11], Hibernus++ [4], and Mementos [3].

We are interested in finding how many seconds of computation remain after a checkpoint is taken, before the MSP430 shuts down. With this data, we are able to find how many instructions can be executed, which corresponds exactly to our execution depth.

The energy used by the MCU is stored in a capacitor, which as we can recall from the literature is a bipole characterized by the following differential relation:

$$i(t) = C\frac{dv(t)}{dt} \tag{8.2}$$

In the above equation, C represents the capacitance of the capacitor, $i(t)$ the current intensity, and $v(t)$ the voltage. Both current and voltage are in

function of time, but if we consider a constant current draw, we can rewrite the differential relation as:

$$I = C\frac{(V_{max} - V_{min})}{\Delta t} \tag{8.3}$$

Since we are interested in finding Δt, we can rewrite our equation as:

$$\Delta t = C\frac{(V_{trig} - V_{min})}{I} \tag{8.4}$$

The execution depth corresponds to the number of operations which are run after a checkpoint is taken, and thus we are interested in the computational time remaining after the checkpoint, which is:

$$t_{remaining} = C\frac{(V_{trig} - V_{min})}{I} - t_{checkpoint} \tag{8.5}$$

Accordingly to the MSP430 datasheet, the current draw during the active mode at 1Mhz of frequency goes from $200\mu A$ up to $420\mu A$, depending on the hit ratio of the cache. Our setup consists in having both SRAM and FRAM active, so a group of reasonable values for current consumption is $250\mu A$, $270\mu A$, and $310\mu A$, which are respectively the current consumption with 75%, 66%, and 50% of cache hit, accordingly to the datasheet.

From the analysis done in Hibernus++ [4] we know that $t_{checkpoint}$ is 1.4ms, V_{min} is 1.88V and V_{max} was found with respect to a bunch of capacitance values. Such values are shown in Table 8.1.

By applying those values to Equation 8.5, we obtain the results shown in Figure 8.1. The remaining execution time goes from $2.47ms$ up to $5.8ms$, depending on the configuration.

Since the power consumption used refers to a clock frequency of $1Mhz$, it is sufficient dividing these results by it. The obtained range of execution depth is shown in Figure 8.2 and goes from 2470 instructions up to 5800 instructions.

This range is valid for evaluating `ScEpTIC` when it is testing a dynamic checkpoint mechanism. We are also interested in evaluating `ScEpTIC` performance when we use its dynamic analysis to test all the possible static checkpoint placements. In this case, we are required to find an execution depth

C (μF)	$V_{trigger}$ (V)
10	2.03
20	1.97
30	1.93
40	1.91

Table 8.1: Capacitance and Voltage before checkpoint pairs found by the calibration of Hibernus++ [4].

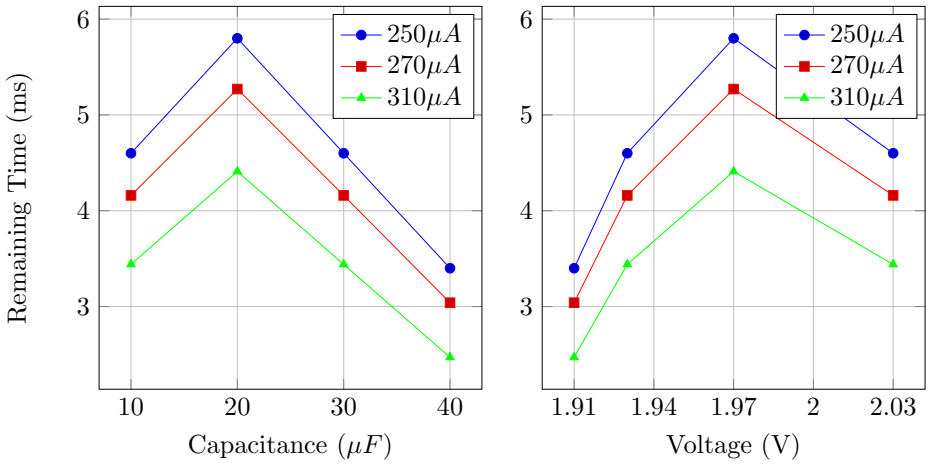

Figure 8.1: Graphs of the remaining time after a checkpoint is taken

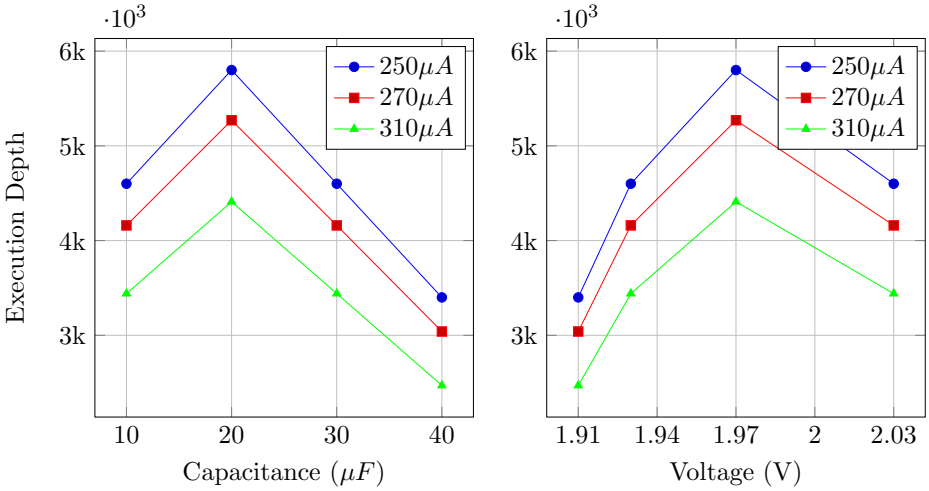

Figure 8.2: Graphs of the execution depth after a checkpoint is taken

which estimates the maximum distance between two checkpoints, since we want to perform an exhaustive test. For doing so, we can not consider static checkpoint mechanism conceived to overcome inconsistencies, otherwise we would obtain a value which does not find any of them, and our analysis would not be effective.

A valid static checkpoint mechanism for estimating the execution depth in such scenario is MementOS [3], which we described in Section 2.3. It has a dynamic behavior with a static checkpoint placement: whenever a checkpoint routine is encountered, it verifies the level of the energy buffer

and, if it is below a certain threshold, it saves a checkpoint. Once the checkpoint is saved, the execution continues until either the MCU shuts down or another checkpoint is reached.

Depending on the placement strategy of MementOS (i.e., loop-latch and function-return), the distance between two checkpoints may vary. The choice of a placement strategy is purely subjective and it is strictly related to a specific application. Furthermore, we must also consider that we can manually place checkpoints, and that a distance granting an exhaustive test should not be dependent on the strategy used. In fact, the dynamic analysis of `ScEpTIC` aims to provide us all the possible inconsistencies and it wants to help us in the selection of a checkpoint mechanism.

For this reason, we want to find the maximum possible distance between two checkpoints, which permit us to actually preserve the work done. This distance can only be such that we are able to reach the second checkpoint if we restart the computation from the first one, independently of the level of the energy buffer. Let us suppose that our MCU performs a checkpoint and then a shutdown happens due to a low energy buffer. Furthermore, let us suppose that the energy source refills the energy buffer only up to the minimum amount of energy required to restart the computation. As consequence, the computation restarts and restores a checkpoint. If the distance from the next checkpoint is too high, we do not have the certainty that we have enough energy to reach it. In such scenario, we do not only waste all the performed computation, but we also incur in the *non-terminating path bug* [18], which consists in having the program stuck between two checkpoints due to the lack of enough energy to reach the second one.

We can refer to this maximum distance considering the voltage level at which we reach two checkpoints. The minimum level the energy buffer can reach after the second checkpoint is the shutdown voltage V_{min}, otherwise we could not complete it. In this case, the maximum distance from the previous checkpoint is exactly one instruction before the voltage level of the energy buffer reaches the trigger voltage V_{trig}. In fact, if it is positioned farther, we will incur in the *non-terminating path bug*. Also, another side effect of such farther positioning would be that neither of these two checkpoints is executed: when the first one is reached, the level of the energy buffer is not low enough to trigger the checkpoint with MementOS, and due to the distance we would not be able to reach the second one. For these reasons, the maximum distance between two checkpoints is identified by the voltage pair (V_{trig}, V_{min}).

Calculating this maximum distance is exactly the same problem of finding the distance between a checkpoint and the consequent shutdown we calculated before. In fact, we want to find the distance between two checkpoints such that when we pass the first one the voltage level is V_{trig}, and when we reach the second one there is just enough energy to perform it (i.e., after the checkpoint the voltage is V_{min}). To find the associated time

interval between V_{trig} and V_{min}, we can apply Equation 8.4. The resulting value will also include the computational time required to perform the second checkpoint, and thus we must subtract it. The final result can thus be obtained by using Equation 8.5, which is the same formula we used for finding the distance between a checkpoint and the consequent shutdown. For this reason, to evaluate `ScEpTIC` performance when we use its dynamic analysis to test all the possible static checkpoint placements, we can use the same execution depth we found before.

8.2.4 Input Source Files

As we stated previously, we are interested in testing the behavior of `ScEpTIC` on code which presents inconsistencies. Most of the benchmarks used in TPC are contained in the MiBench2 suite [29], which consists in the MiBench [30] benchmark suite ported for IoT devices. Such suite has been developed for the evaluation of Ratchet [10], and the contained benchmark algorithms are used for the evaluation of other checkpoint mechanisms. For example, Hibernus [11] uses the Fast Fourier Transform, and MementOS [3] uses *CRC* and *RSA*.

For the evaluation of `ScEpTIC`, we selected three different benchmarks from the MiBench2 [29] suite. They represent a set of heterogeneous use cases which are common in the TPC domain: data integrity verification (i.e., *CRC*), signal analysis (i.e., *FFT*), and encryption (i.e., *AES*).

Moreover, we must also consider that the most common application of devices in TPC comprehend their usage as wireless sensors, and thus their work flow consists in sensing the environment, processing the sensed values, and then store the results or send them to the main node of the system. This is exactly the behavior of different benchmarks named as *sense* and used to evaluate MementOS [3], EKHO [20], and QuickRecall [12]. Also, a similar work flow is used to describe the entire behavior of Chain [16]. For this reason, we will also evaluate `ScEpTIC` with a benchmark presenting this *sense* behavior.

To perform the evaluation of `ScEpTIC`, we must also specify the memory configuration and the checkpoint mechanism we are going to use with each benchmark. Note that we will consider the checkpoint mechanism described in Section 2.3.

`ScEpTIC` is a tool designed for finding all the inconsistencies of a program. With its help, we can perform comparisons of different configurations for our application, so that we can select the one which best suits our needs. For this reason, for selecting the memory configuration and the checkpoint mechanism for our benchmarks, we will act as a developer which wants to explore different alternatives for his application.
As side effect, we will also show how `ScEpTIC` can help in these choices.

Furthermore, for evaluation purposes we will not consider the possibility

of using dynamic checkpoint mechanisms such as Hibernus [11] and QuickRecall [12]. They are designed to not present data inconsistencies by stopping the computation after a checkpoint is taken. For this reason, the analysis performed by `ScEpTIC` would be a simple sequential run of the program, both in terms of results and executed instructions, and such analysis would be useless to evaluate its performance.

For each benchmark we selected a set of configurations which allow us to act as a developer that is testing its code, and that also permits us to evaluate `ScEpTIC` under different scenarios.

CRC. Cyclic Redundancy Check codes are usually used to detect errors in data exchanged by devices. This benchmark generates the CRC code of a predefined input in two different ways: *crcSlow* and *crcFast*. The second method exploits a table called *crcTable* to keep track of intermediate results, so to reuse them instead of performing the same computation multiple times. The content of such table is not dependent on the data from which the CRC is generated.

Now that we analyzed the structure of this program, we can explore different configurations. As first action, we could select a checkpoint mechanism, and thus the memory configuration it supports. Before doing so, we must consider that when we move a portion of memory into NVM, we reduce the checkpoint overhead at the expenses of data inconsistencies. In fact, the amount of data to be saved by a checkpoint is reduced, but checkpoints must happen more frequently to avoid data inconsistencies. Performing the selection of a checkpoint mechanism without considering the amount of inconsistencies introduced by its memory configuration can cause a significant energy waste in our program.

For these reasons, as first action we should instead explore the inconsistencies introduced by a certain memory configuration. Then, if we are satisfied by the results, we can then select the checkpoint mechanisms which are designed to work with such configuration. Otherwise, we can explore another memory configuration and repeat this process over again.

The steps we follow for analyzing different memory configurations are:

1. *Dynamic analysis with global variables in NVM*
   If we do not allocate any element into NVM, we do not have any data inconsistency, and each checkpoint contains the entire main memory. To reduce the size of a checkpoint, we can move the global variable *crcTable* into NVM, and thus we can analyze the inconsistencies caused by such movement.
   The static checkpoint mechanisms which can work with this memory configuration are MementOS [3] and DINO [1]. MementOS automatically places checkpoints accordingly to a user-specified policy, and it does not take care of data inconsistencies. Instead, DINO relies on the user to place checkpoint boundaries, and then it creates versions

of variables to preserve data consistency.
Since we do not want to stick a priori with one of these two checkpoints mechanisms, we will run a dynamic analysis with `ScEpTIC`. In this way, it will try all the possible static checkpoint placements, and it will return all the inconsistencies present in the program.
Once we have this result, we know the amount of data inconsistencies present in our program. This number tells us the overhead introduced by the versioning of DINO, and it also represents the number of checkpoints required to minimize such overhead. In fact, in DINO checkpoint boundaries are directly translated into checkpoints, and thus we can exploit the methods we described in Chapter 4 to place checkpoints in a way which resolves data inconsistencies.
We must note that with DINO the burden of placing checkpoints is leaved to us. If we try to solve all the inconsistencies with a checkpoint placement, DINO would not insert any versioning of variables. However, it might be possible that the number of checkpoint we introduced is too high, resulting in an excessive overhead.
Furthermore, if the program does not present any inconsistency with this memory configuration, we are still required to find a placement which minimizes the overhead introduced by checkpoints.
As we will see in the results, this memory configuration does not present any kind of data inconsistency. For this reason, placing *crcTable* in NVM reduces the checkpoint overhead without introducing data inconsistencies, and we do not need to analyze if the checkpoint placement of MementOS solves the found inconsistencies.

2. *Static analysis with stack in NVM*
As next step, we want to explore the possibility of allocating the stack into NVM. As we explained in Section 4.3, in such scenario each memory access and call operation has the potential to produce inconsistent results. When we allocate the stack into NVM, we know a priori which magnitude of data inconsistency we can expect. In fact, it is directly proportional to the number of function calls and local variables. Furthermore, the amount of all the possible data inconsistencies makes unmanageable their resolution without the help of a checkpoint mechanism, and thus we can only use one which can support it. For these reason, running a dynamic analysis with such setup would not tell us useful information.
Ratchet [10] is the only static checkpoint mechanism designed to work with the stack allocated in NVM, and in doing so it introduces a high overhead. An alternative solution consists in exploring the possibility of using MementOS with a loop-latch strategy, and in evaluating the amount of checkpoints required to solve the found inconsistencies.
If the effort required to solve data inconsistencies is reasonable, we can use it instead of Ratchet, with the benefit of a reduced checkpoint

overhead. In fact, in our configuration, MementOS does take a checkpoint only if the level of the energy buffer is low enough, and thus the overall number of checkpoints executed during runtime is lower than the one of Ratchet.
For this reason, we will run a static analysis with `ScEpTIC` over our program, with applied the checkpoint placement produced by the loop-latch strategy of MementOS.

3. *Dynamic analysis with stack in NVM*
As we previously stated, running such analysis will not tell us useful information for our scope. However, we decided to run such analysis to demonstrate such statement and for evaluation purposes.

FFT. Fast Fourier Transform is used to convert a signal from the domain of time into the one of frequencies, so that we can analyze an input using its frequency. This benchmark computes the FFT and inverse FFT over two different random input signals. It uses a set of global variables (i.e., *realin*, *imagin*, *realout*, and *imagout*) to store intermediate results. The real work of the program is computed inside the *fft_float* function, which access through pointers such variables.

Now that we analyzed the structure of this program, we can explore different configurations. As we stated before, we do not want to stick a priory with a specific checkpoint mechanism. Instead, we want to explore different memory configurations, so to evaluate the possibility of reducing the checkpoint overhead. In doing so, we expect an increase of data inconsistencies but, depending on their frequency, it can be worth to use such memory configuration.

The steps we follow for analyzing different memory configurations are:

1. *Dynamic analysis with realin, imagin, realout, and imagout in NVM*

   As first action for our analysis, we want to evaluate the inconsistencies introduced by the movement of global variables into NVM. Since most of the work is performed over only four global variables (i.e., *realin*, *imagin*, *realout*, and *imagout*), we want to evaluate the inconsistencies introduced by their movement into NVM.

   Once we have the result of such test, we are able to understand the contribution of this set of variables to data inconsistencies. Since most of the work is performed on such set, if they introduce an unreasonable amount of inconsistencies, we can stop our analysis. In fact, by allocating all the global variables or the entire stack into NVM, we are also including in it the analyzed variable set. If we do not afford to solve the inconsistencies generated by the analyzed variable set, we will certainly not be able to afford the inconsistencies generated by the allocation of more elements into NVM.

Instead, if the effort required to solve the introduced inconsistencies is affordable for us, we can continue our analysis. As we will see, the amount of data inconsistencies generated by such variables is relatively low, and thus we can continue our analysis.

2. *Dynamic analysis with global variables in NVM*

   Since the first analysis returned us a positive result, as next step we run a dynamic analysis with all the global variables allocated into NVM. The result of such analysis will tell us the contribution to data inconsistencies of all the other global variables not included in the set we previous analyzed.

   As we will see, the amount of inconsistencies is the same of the one found by the previous analysis, and thus the other global variables does not introduce any new inconsistency. As for what we did in the CRC case, we can use this result to reduce the versioning overhead of DINO, or we can evaluate if the found inconsistencies are resolved by the *loop-latch* placement of MementOS. Note that, since the most computational-intensive part of this program consists in *for* loops, we do not consider the *function-return* strategy.

3. *Static analysis with global variables in NVM*

   As next step, we decide to run a static analysis considering the *loop-latch* placement of MementOS. In this way, we are able to compare the effort required to solve the found inconsistencies with the one required to reduce the versioning overhead of DINO.

   As we will see, this analysis returns the same results of the previous one, meaning that the *loop-latch* strategy of MementOS does not resolve any inconsistency. If we choose to stop our analysis here, we can choose either to select MementOS or DINO as our checkpoint mechanism.

   However, given the relatively low amount of data inconsistencies, we can decide to go further with the analysis, and we can evaluate the amount of data inconsistencies generated by the placement of the stack into NVM.

4. *Static analysis with stack in NVM*

   As final step, we want to explore the possibility of allocating the stack into NVM. As we previously stated in the CRC case, running a dynamic analysis with stack in NVM would not tell us useful information, and we are restricted to choose a checkpoint mechanism which support this memory configuration.

Moreover, we can directly choose to use Ratchet, but we are performing such choice without knowing the effort required to solve data inconsistencies with the *loop-latch* placement of MementOS.

For this reason, we will run a static analysis with `ScEpTIC` over our program, with applied the checkpoint placement produced by the loop-latch strategy of MementOS.

AES. Advanced Encryption Standard or AES is a cryptographic algorithm which is commonly used in TPCs for enabling the possibility of exchanging data securely between devices. This benchmark consists in running AES encryption and decryption over predefined inputs in two different modes (i.e., ECB and CBC).

The *main* function calls four subroutines, one for each pair of action and mode, which runs the requested part of the benchmark. Each subroutine setups a part of the benchmark using local variables, and then it calls the required cryptographic function.

Accordingly to the comment in the source file, the global variables *sbox*, *rsbox*, and *rcon* define data structures characterizing AES, and during the program execution no write operation changes their content. For this reason, we can directly allocate them into NVM. In this way, we are reducing the checkpoint overhead for free, since we are not increasing checkpoint frequency (i.e., no data inconsistency introduced).

Continuing our analysis of the source code, we can notice that there exists other four global variables: *state*, *Key*, *IV*, and *RoundKey*. The first three variables are pointers, and when a cryptographic function is called, it links them with the local variables passed as arguments. All the functions that compute the intermediate results works without any argument, and they directly refers to these three global variables.

Due to this particular design choice, we expect a high level of inconsistencies if the stack is allocated into NVM, since most of the computation happens on data stored in it. Furthermore, the number of function calls is substantial, and thus we also expect a substantial number of *Activation Record Inconsistencies*.

The steps we follow for analyzing different memory configurations are:

1. *Dynamic analysis with global variables in NVM*

   As first analysis, we want to evaluate the amount of inconsistencies introduced by the movement of global variables into NVM. As we previously stated, the global variables *sbox*, *rsbox*, and *rcon* are not written by any operation and thus can be safely allocated into NVM without introducing any inconsistency.

   The other group of global variables is composed by:

   - The variable *RoundKey*, which is used to derive elements from the cryptographic key when a cryptographic function is called.

- The variables *state*, *Key*, and *IV*, which are pointers used to access local variables across different function.

With a first look, we might think that the amount of work computed on the global variables will not produce a relevant number of data inconsistencies. For this reason, we choose to run a dynamic analysis with the entire set of global variables allocated into NVM.

As we will see, `ScEpTIC` will find a substantial number of data inconsistencies (i.e., about *100*) when it runs the previous analysis. By looking at this result, we are able to notice that all the inconsistencies but one happens due to the placement of the variable *state* into NVM. In fact, if we take a closer look at our source code, we will see that such variable is reassigned every time an intermediate result is produced.

This result tells us that we are able to reduce almost for free the checkpoint overhead by placing all the global variables but the *state* one into NVM. In fact, such placement does only produce one data inconsistency, which it is easy to solve.

2. *Static analysis with global variables in NVM*

   If we take a closer look at the inconsistencies produced by the placement of the variable *state* into NVM, we can notice that most of the inconsistencies has a pattern. They all happen when the checkpoint is taken in a function, and a shutdown happens during the execution of another one. For this reason, we can proceed with our analysis, and we can evaluate if using MementOS would resolve a substantial number of them.

   This program present both frequent function calls and loops. For this reason, we will explore both options by running two different static analyses: one considering the *loop-latch* strategy of MementOS, and one considering the *function-return* one.

   As we will see, both the strategies solves all the inconsistencies, and thus we can choose one of them. Furthermore, we can notice that the placement produced by the *loop-latch* strategy has a significantly lower amount (i.e., one sixth) of checkpoint executed with respect to the *function-return* one. This results in a lower checkpoint frequency, and thus a lower overhead. For this reason, we can choose this strategy.

3. *Static analysis with stack in NVM*

   The results we obtained until now are positive, and thus we can explore the possibility of allocating the stack into NVM. As we previously stated, running a dynamic analysis does not help in such scenario, and we can either choose to use Ratchet or to verify the amount of inconsistencies produced by MementOS.

Our program present a high frequency of function calls, and thus *activation record inconsistencies* will be one of our main problems. As we stated in Section 4.3, they may happen when a function returns and another one is called. Using the *function-return* strategy of MementOS will certainly resolve such kind of inconsistencies but, as we previously seen, we are increasing by 6 times the frequency of checkpoints. On the other hand, using the *loop-latch* strategy certainly does not solve any *activation record inconsistency*, and it can be a valuable option if the number of checkpoints required to solve such inconsistencies does not increase their frequency by six times.

For these reasons, as for the previous case, we will explore both options by running two different static analyses: one considering the *loop-latch* strategy of MementOS, and one considering the *function-return* one.

Once we have these results, we are able to find the most valuable option for us.

Sense. As we previously stated, the most common application for devices in TPC comprehend their usage as wireless sensors. For this reason, most of the work done in the TPC field, such as MementOS [3], EKHO [20], and QuickRecall [12], uses a benchmark called *sense*. It does not refer to a specific program, but instead it consists in the description of a particular behavior: sensing the environment, processing the sensed values, and finally send the results to the main node of the system. In fact, the *sense* benchmarks used by previous works are different from each other from the point of view of the code, but they are the same from the point of view of the behavior.

Example 8.3 shows the *sense* benchmark we will use for the evaluation of `ScEpTIC`, and it computes the mean and variance over a set of sensed samples. This benchmark does not present any function call, and its main work flow is contained into *for* loops. The global variables are used for storing sensed data, and for computing intermediate results.

As for previous cases, we are interested in finding the best memory configuration which does reduce the checkpoint overhead and does not introduce an excessive amount of data inconsistencies. In fact, allocating memory directly into NVM reduces the checkpoint overhead, but we might introduce an excessive number of data inconsistencies. This will require us to insert checkpoints into specific places, with the effect of increasing the checkpoint frequency, reducing the benefits of a smaller checkpoint overhead.

The steps we follow for analyzing different memory configurations are:

1. *Dynamic analysis with global variables in NVM*

   As it is possible to note in the code, most part of the work is performed over global variables. The only local variable is i, which is used as

```
1  int data[50];
2
3  float std_incr = 0;
4  float sum = 0, variance = 0, mean = 0;
5
6  int main() {
7  int i;
8
9  // sense 50 samples
10 for(i = 0; i < 50; i++) {
11 data[i] = input();
12 }
13
14 // compute sum and mean
15 for(i = 0; i < 50; i++) {
16 sum = sum + data[i];
17 }
18 mean = sum / 50;
19
20 // compute variance
21 for(i = 0; i < 50; i++) {
22 // pow(data[i] - mean,2) => std_incr*std_incr
23 std_incr = data[i] - mean;
24 variance = variance + std_incr*std_incr;
25 }
26 variance = variance / 50;
27
28 // send computed data to main node
29 out(mean, variance);
30 }
```

Example 8.3: Program used for evaluating `ScEpTIC` algorithm, which presents a behavior similar to *sense* benchmark of MementOS [3]

loop iterator. For this reason, we will run a dynamic analysis with the global variables in NVM, so to find the inconsistencies introduced by such configuration.

Once we have this result, we are able to establish if the amount of data inconsistencies introduced by the allocation of global variables into NVM is reasonable or not. As we will see, the number of data inconsistencies is relatively low, and thus the placement of global variables into NVM is a viable option.

Now we can either choose to exploit this result for minimizing the versioning overhead of DINO, or we can explore the possibility of using MementOS.

2. *Static analysis with global variables in NVM*

   If we look at the results of the previous analysis, we can see that all the data inconsistencies happens within different iterations of *for* loops. We can note that placing checkpoints after the end of each loop will certainly remove all the inconsistencies. For this reason, DINO seems to be a reasonable option. Since the amount of work performed inside loops is relatively low, we could just place checkpoint boundaries at their ends.

   Before considering our analysis done, we still want to evaluate other options. For this reason, we choose to apply the *loop-latch* strategy of MementOS to our program, and thus we run a static analysis.

   As we will see in the results, this configuration leaves only one data inconsistency, which can be easily resolved by placing a checkpoint in a specific position. Unfortunately, the checkpoint frequency would be higher than the one of DINO, and with this configuration our best option is thus using it.

3. *Static analysis with stack in NVM*

   Since the amount of inconsistencies introduced by the movement of global variables into NVM is manageable, we want to evaluate the possibility of allocating the stack into NVM. As we stated in the other test cases, we could either use Ratchet or MementOS.

   Since there are no function calls, we will run a static analysis considering the *loop-latch* placement of MementOS. Furthermore, we do not expect to introduce a large amount of data inconsistencies by moving the stack into NVM, since there exists only one local variable and there are no function calls.

   As we will see in our results, we introduced more inconsistencies due to the positioning of the iterator i into the NVM. Even if the number of inconsistencies is relatively low (i.e., 7), it is not beneficial to us allocating the stack into NVM. In fact, we reduce the checkpoint overhead by one single memory cell (i.e., local variable i), but we have to introduce seven more checkpoints, thus increasing their frequency. The reduction of the overhead does not compensate for this increased frequency, and thus we will obtain a loss in performance.

8.2.5 Quantitative Evaluation Results

In this section we discuss the results of the tests we previously described for evaluating `ScEpTIC` data analysis performance. For running our dynamic tests we used an *execution depth* of *3000*, which a reasonable value within the rage we found in Section 8.2.3.

CRC:

1. *Dynamic analysis with global variables allocated in NVM.*

   Figure 8.3 shows the results of this evaluation. Since it consists in a dynamic analysis, we did not measure the performance of the *Baseline* algorithm. As we explained in Section 8.2.1, it would take too much time for such algorithm to complete a dynamic analysis. Using Equation 8.1, we can analytically calculate that the *Baseline* algorithm would execute $2.25 \cdot 10^{11}$ instructions, since n_{ops} is $5.2 \cdot 10^4$ for CRC. Considering that in our configuration we were able to reach a maximum speed of $1.8 \cdot 10^4$ *instruction/s*, it would take not less than $1.25 \cdot 10^7$ seconds to complete the analysis, which are 144 days. Furthermore, in this case there is only one instruction writing the NVM, which is the assignment of *crcTable* variable inside the *crcInit()* function. For this reason, we can also analytically calculate the number of found inconsistencies, which would be 3000. In fact, the *Baseline* algorithm tries a checkpoint at every line of code, and since the execution depth is *3000*, we reach the writing instruction starting from *3000* instructions before it. As we stated in Section 8.2.1, the *Baseline* algorithm represents inconsistencies pairs of $(checkpoint, reset)$, and since we reach the inconsistent state with *3000* different checkpoints, we will have *3000* different inconsistencies.

   For these reasons, the *Baseline* algorithm performance is not comparable to the one of `ScEpTIC` and *Dummy ScEpTIC*, and it largely outperformed by both of them. Furthermore, the *Baseline* algorithm returns *3000* miss-classified inconsistencies, leading to unusable analysis result.

   Let us now continue our comparisons, considering only `ScEpTIC` and *Dummy ScEpTIC*. As first metric, we are interested in the number of inconsistencies found by each algorithm, which is shown in Figure 8.3d. In this case, *Dummy ScEpTIC* and `ScEpTIC` find the same number of inconsistencies, which is 0. As we will see in the results of the next benchmarks, *Dummy ScEpTIC* is subject to miss-classification of inconsistencies. In this case, it does not miss-classify any inconsistency thanks to knowledge of the optimizations of the reset points taken from `ScEpTIC`.

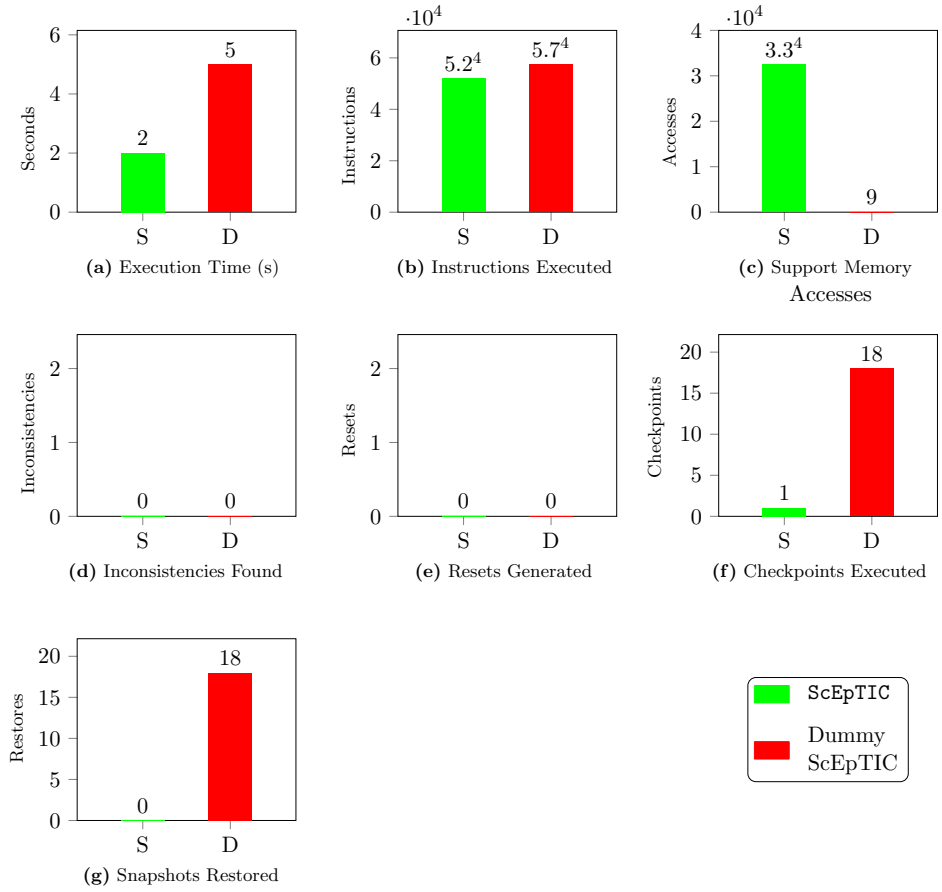

Figure 8.3: Data evaluation results: Dynamic Analysis of CRC benchmark with global variables allocated in NVM.

Let us now consider Figure 8.3b and Figure 8.3c, which show respectively the instructions executed and the support memory accesses. `ScEpTIC` executes $5.2 \cdot 10^4$ instructions with $3.3 \cdot 10^4$ support memory accesses, and instead *Dummy ScEpTIC* executes $5.7 \cdot 10^4$ instructions with only 9 support memory accesses. This increased number of executed instructions in *Dummy ScEpTIC* is caused by the number of checkpoint tested and snapshots restored, as it is possible to see in both Figure 8.3f and Figure 8.3g. In fact, `ScEpTIC` tests only one checkpoint placement and it does not restore any snapshot. Instead, *Dummy ScEpTIC* tests 18 different checkpoint placements, and it restores a snapshot at the end of each tested checkpoint, even if no inconsistency is found. This action is required for *Dummy ScEpTIC* to perform an exhaustive dynamic analysis. In fact, if it does not re-

store a snapshot at the end of each test, it would skip the test of any checkpoint placement contained in the tested interval.

Let us now consider Figure 8.3a, which shows the time required for running this benchmark, that is in the same order of magnitude for both the two algorithms. We are not interested in this metric alone, and instead we are interested in understanding its relationship with other metrics. *Dummy ScEpTIC* takes 2.5 times more than `ScEpTIC` for running this analysis. This is caused by the higher number of checkpoints tested by *Dummy ScEpTIC*. Thanks to its optimizations, `ScEpTIC` has only to test one checkpoint and it does not restore any snapshot, since the state remains consistent. The lack of such optimizations in *Dummy ScEpTIC* causes an excessive number of checkpoints tested, resulting in an overall worst performance with respect to `ScEpTIC`. Moreover, since we are performing a dynamic analysis, we are considering checkpoints to happen at any line of code. For this reason, *Dummy ScEpTIC* has to restore a snapshot after it finishes testing all the resets point associated to a checkpoint placement. This is required for granting an exhaustive test from the checkpoint placement standpoint. In fact, if it avoids this behavior, the next instruction to be executed is the one after the latest rest point tested. As consequence, the next checkpoint placement to be tested can be only after such reset point, with the effect of skipping the testing of checkpoint placements which happens between the previously tested checkpoint and the latest reset point.

Finally, given the results of such benchmark, we can also establish that the largely higher number of support memory accesses that happens in `ScEpTIC` does not influence the execution time of the test as much as it does the number of checkpoints tested and the consequent number of instructions executed which happens in *Dummy ScEpTIC*.

Given the results we just described, we can say that the performance of `ScEpTIC` is sensibly better than *Dummy ScEpTIC* and *Baseline* for this benchmark. Furthermore, *Dummy ScEpTIC* has a better performance with respect to *Baseline* thanks to the optimizations taken from `ScEpTIC`.

2. *Static analysis with stack allocated in NVM and checkpoints placed accordingly to the loop-latch strategy of MementOS [3].*

   Figure 8.4 shows the results of this benchmark. It consists in a static analysis, and thus the checkpoints to be tested are fixed inside the code. As consequence, we are also able to get the results from the analysis performed by the *Baseline* algorithm.

   Firstly, let us consider Figure 8.4a, which shows the time required by the three algorithms for running the benchmark. As we can see, the

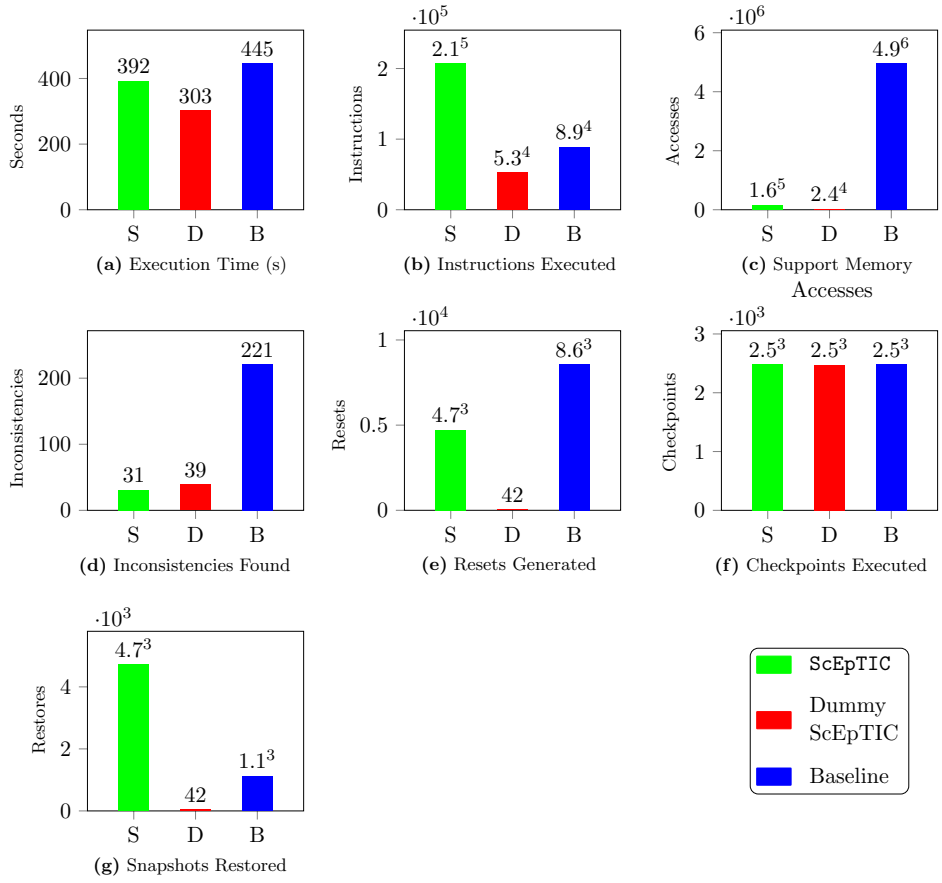

Figure 8.4: Data evaluation results: Static Analysis of CRC benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM.

execution time is in the same order of magnitude for the three algorithms. *Baseline* is the algorithm that takes more time for completing the analysis, and *Dummy ScEpTIC* is the one which takes less time.

For understanding this result, let us focus on Figure 8.4e, which shows us the resets generated during the analysis. In all the three algorithms, the generation of a power reset is followed by the restoration of a checkpoint, and thus the number of power resets also indicates the number of checkpoints restored. As consequence, we can say that resets are correlated to the amount of different combinations of intermittent execution scenarios tested. The excessive high number of resets generated by *Baseline* is caused by the lack of an optimization to the reset points, which is instead present in both `ScEpTIC` and *Dummy ScEpTIC*. *Dummy ScEpTIC* performs a number of resets which is sig-

nificantly lower than the other two algorithms, and this is the main reason for the lower execution time. The lack of such optimizations results also in a higher number of support memory accesses, as we can see in Figure 8.4c. In fact, every time a reset is generated, the support memory is accessed for verifying the presence of inconsistencies inside the obtained state.

The loop-latch placement of MementOS [3] places checkpoints at the end of the loop body, and thus the same checkpoint must be tested multiple times, since they happen inside a loop iteration and at each one of them the state is different. *Dummy ScEpTIC* performs such a low number of resets due to its representation of inconsistencies. In fact, as we explained in Section 8.2.1, it is not able to find the actual instruction causing an inconsistency, and instead it assumes that is the instruction before the reset point which caused the inconsistency. As consequence, once *Dummy ScEpTIC* finds an inconsistency, it does not retry the associated reset points for a given checkpoint, and thus it will skip the associated resets for all the subsequent analysis of the same checkpoint. Instead, `ScEpTIC` performs the analysis of the subsequent loop iterations even if an inconsistency was found, since different data might cause different inconsistencies, as we explained in Section 8.2.1. As consequence, *Dummy ScEpTIC* tests a lower number of intermittent execution scenarios, and returns us less precise information about inconsistencies compared to `ScEpTIC`.

Let us now consider Figure 8.4b, which shows the number of instructions executed. `ScEpTIC` executes a number of instructions which is higher with respect to the ones executed by *Dummy ScEpTIC* and *Baseline*. This is caused by both the resets generated and the snapshots restored, as we can see in Figure 8.4e and Figure 8.4g. In the three algorithms, a snapshot is restored whenever the state is inconsistent. The subsequent operations consist in a sequential execution of the code, which is required for reaching the latest reset point and continuing the analysis. For this reason, restoring a snapshot increases the number of instructions executed, but since they are run using a sequential execution scenario, it does not influence the execution time excessively. In fact, testing an intermittent execution is more time-consuming with respect to a sequential execution, and thus the number of snapshots restored has a significant lower impact on the execution time with respect to the number of resets generated. This is the reason for the fact that `ScEpTIC` has a significant higher number of instructions executed with a contained execution time.

Moreover, Figure 8.4c shows the support memory accesses performed by the three algorithms. The support memory is accessed whenever an algorithm requires verifying if the state is consistent.

```
[Write After Read Inconsistency]
    Cell address: S-0x24
    Correct content: 8
    Read content: 7
    Read at clock: 64
    Written at clock: 66
    Memory Read happens at:
        @main -> #6 (Line: 37; Column: 3; Function name: main)
            @crcSlow -> #52 (Line: 116; Column: 32; Function
                name: crcSlow)
    Memory Write happens at:
        @main -> #6 (Line: 37; Column: 3; Function name: main)
            @crcSlow -> #54 (Line: 116; Column: 32; Function
                name: crcSlow)
```

Example 8.4: Portion of the results of the analysis returned by `ScEpTIC`

```
Cell address: S-0x24
Checkpoint happens at: @crcSlow -> #51
Caused by: @crcSlow -> #55

Cell address: S-0x24
Checkpoint happens at: @crcSlow -> #57
Caused by: @crcSlow -> #55

Cell address: S-0x24
Checkpoint happens at: @crcInit -> #48
Caused by: @crcFast -> #42
```

Example 8.5: Portion of the results of the analysis returned by *Dummy ScEpTIC*

Dummy ScEpTIC perform such analysis every time it generates a reset, and instead `ScEpTIC` performs accesses to the support memory every time it executes an instruction. If we consider the metric alone, we might think that *Dummy ScEpTIC* performs better than `ScEpTIC`, but if we instead consider when the memory is accesses, we can notice that the usage ratio of the support memory is higher in *Dummy ScEpTIC*. In fact, it performs 42 resets with $2.4 \cdot 10^4$ memory accesses, which means 517 support memory accesses per reset. Instead, `ScEpTIC` executes $2.1 \cdot 10^5$ instructions with $1.6 \cdot 10^5$ memory accesses, which means 0.76 memory accesses per instruction.

Finally, let us now consider Figure 8.4d, which shows the inconsistencies found by the three algorithms. *Baseline* finds an excessive number of inconsistencies, and this is the direct consequence of its inability of recognizing the actual instructions causing a reset, as we explained in Section 8.2.1. As consequence, the results returned by *Baseline* are almost unusable, since the efforts required for manually verifying the

actual instructions causing the inconsistencies is not irrelevant, considering that it finds 221 inconsistencies. Instead, *Dummy ScEpTIC* and `ScEpTIC` find a comparable number of inconsistencies. For comparing their results, we must consider how the two algorithms represents inconsistencies. Example 8.5 and Example 8.4 shows a portion of the results returned by *Dummy ScEpTIC* and `ScEpTIC`. *Dummy ScEpTIC* represents inconsistencies identifying the cell address, the checkpoint and the reset point causing the inconsistency. Instead, `ScEpTIC` identifies where the memory read and write happen. `ScEpTIC` returns us a precise information about the inconsistency. For how *Dummy ScEpTIC* represents inconsistencies, it is not able to represents them as it does `ScEpTIC`. As consequence, we have imprecise and redundant information. In fact, Example 8.4 shows an inconsistency identified by `ScEpTIC` which happens at the address $0x24$ in the stack. The same inconsistency is identified as three different ones by *Dummy ScEpTIC*, as Example 8.5 shows.

Recognizing an inconsistency with the result returned by *Dummy ScEpTIC* requires us to analyze the code, for understanding where the memory read and write associated to the inconsistency happen. The results returned by `ScEpTIC` contains this information, which is essential for avoiding inconsistencies. If we inspect the results of this analysis, we can find another similar case for the cell with address $0x18$: *Dummy ScEpTIC* finds 10 inconsistencies over this same cell, and `ScEpTIC` finds that the actual cause is a single memory write. If we deepen into this analysis, we can also notice that *Dummy ScEpTIC* does not find all the inconsistencies found by `ScEpTIC`. For example, `ScEpTIC` finds an inconsistency with the memory cell at address $0x1e$, which is not even analyzed by *Dummy ScEpTIC*. This is caused by the reset points sub-optimization of *Dummy ScEpTIC*, which makes it skip some resets points. In fact, it would be useless re-test the same reset point in a future analysis, since *Dummy ScEpTIC* is not be able to analyze the memory as `ScEpTIC` does, and thus it would not find any new inconsistency.

In this benchmark, the *Baseline* algorithm is outperformed by `ScEpTIC` and *Dummy ScEpTIC*, especially because it returns us an unmanageable number of inconsistencies. As we stated in Section 8.2.1, *Dummy ScEpTIC* is the algorithm obtained by applying some optimization of `ScEpTIC` to the *Baseline* algorithm. *Dummy ScEpTIC* and *Baseline* share the same representation for inconsistencies, and thus their result is not as precise as the one of `ScEpTIC`.

Dummy ScEpTIC has a lower execution time with respect to `ScEpTIC`, but it tests a significant lower number of combinations of intermittent executions, and the quality of the information returned is significantly

lower than the one returned by `ScEpTIC`. For these reasons, we can state that for this benchmark, `ScEpTIC` performs a better analysis with a comparable execution time.

3. *Dynamic analysis with stack allocated in NVM.*

   Figure 8.5 shows the results of this evaluation. As we previously explained, we do not have the measures of the *Baseline* algorithm since we performed a dynamic analysis, and it would not be able to complete it in a reasonable amount of time.

   The difference in performance between algorithms of this analysis are similar to the one of the first benchmark setup. As we did for it, we can analytically calculate that the *Baseline* algorithm would execute $2.25 \cdot 10^{11}$ instructions, and it would take $1.25 \cdot 10^{7}$ seconds to complete the analysis, which are 144 days. Both the number of instructions executed and the execution time are way above the respective metric of both *Dummy ScEpTIC* and `ScEpTIC`, as we can see in Figure 8.5a and Figure 8.5b. Moreover, since *Dummy ScEpTIC* is obtained by applying some optimization of `ScEpTIC` to the *Baseline* algorithm, the number of inconsistencies found by the *Baseline* algorithm can not be lower than the one of *Dummy ScEpTIC*. As consequence, from an inconsistency standpoint it can only perform as equal or worse than *Dummy ScEpTIC*. For these reasons, we can say that the *Baseline* algorithm performance is significantly worse than the other two algorithms, which largely outperform it.

   Let us now focus only on *Dummy ScEpTIC* and `ScEpTIC`. As we previously explained, *Dummy ScEpTIC* restores a snapshot after it finishes testing all the resets point associated to a checkpoint placement, otherwise its analysis is not exhaustive. During an intermittent execution test, `ScEpTIC` analyzes the instructions executed so to understand if a checkpoint placement should be tested before any of them. For this reason, it is able to restore a lower number of snapshots with respect to *Dummy ScEpTIC*, as we can see in Figure 8.5g. As consequence, it is also able to skip testing useless checkpoint placements, while maintaining the test exhaustive. This results in a lower number of checkpoints test, as shown in Figure 8.5f.

   The lack of such optimization, causes *Dummy ScEpTIC* to test an excessive number of checkpoints. This results in a significantly higher amount of time required for completing the analysis with respect to `ScEpTIC`, as we can see in Figure 8.5a.

   Finally, let us consider Figure 8.5d, which shows that *Dummy ScEpTIC* finds 534 inconsistencies and `ScEpTIC` only 72. At a first look we could think that `ScEpTIC` is not able to classify some inconsistencies,

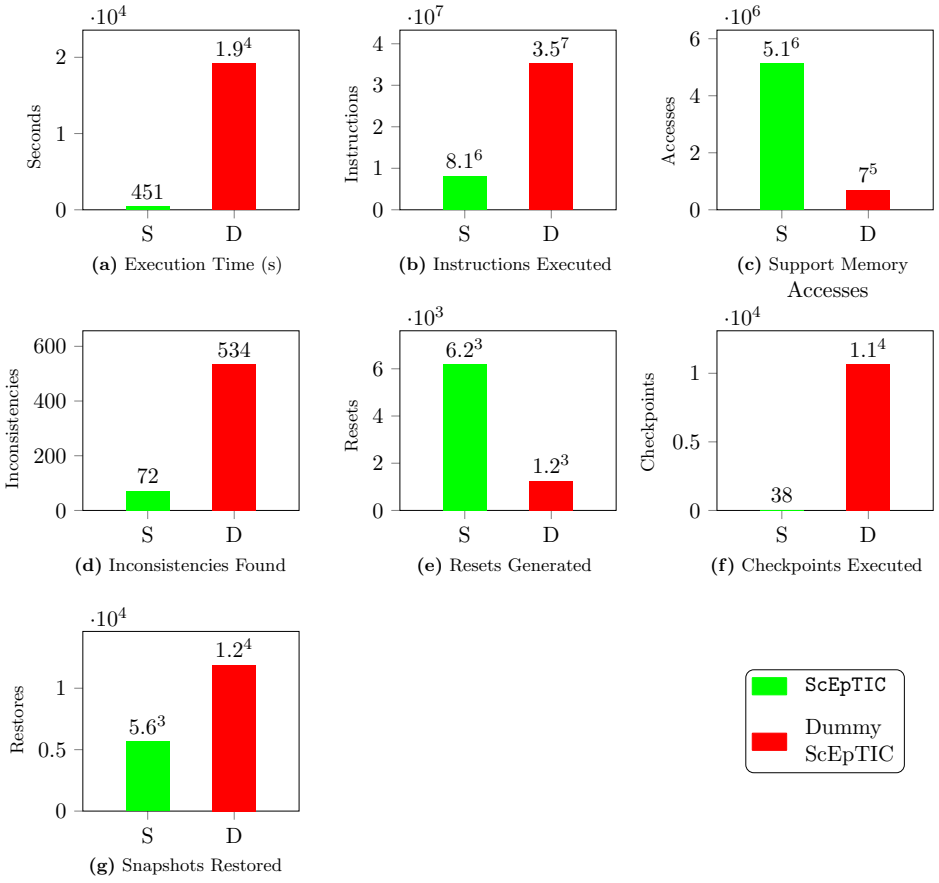

Figure 8.5: Data evaluation results: Dynamic Analysis of CRC benchmark with the stack allocated in NVM.

but it is not the case. In fact, as we previously stated, *Dummy ScEpTIC* finds all the pairs of checkpoint / reset which cause the state to be inconsistent, and instead `ScEpTIC` is able to find the actual cause of them. As consequence, all the information about inconsistencies found by *Dummy ScEpTIC* is included within the result of `ScEpTIC`. In fact, if we analyze the information returned by *Dummy ScEpTIC* for finding the actual cause of inconsistencies, we will have the same result returned by `ScEpTIC`. Moreover, *Dummy ScEpTIC* is an optimization of the *Baseline* algorithm which, as we explained in Section 4.5.1, classifies each write operation into NVM as *inconsistency*. As consequence, the information returned by *Dummy ScEpTIC* contains also miss-classified inconsistencies, which instead are not present in `ScEpTIC`.

Considering these results, we can say that `ScEpTIC` largely outperforms its two counterparts in this benchmark, both from performance and inconsistency standpoints.

Given the results of these three benchmarks, we can state that `ScEpTIC` largely outperforms *Dummy ScEpTIC* and *Baseline* in the two *Dynamic Analysis*. Instead, in the *Static Analysis* it has a comparable performance from an execution standpoint, but it returns a very precise information about inconsistencies, making its overall performance to be better than the other two algorithms. Moreover, *Dummy ScEpTIC* has a significantly better performance with respect to *Baseline*, and this is achieved thanks to the optimizations taken from `ScEpTIC`.

FFT:

1. *Dynamic analysis with variables realin, imagin, realout, and imagout allocated in NVM.* Figure 8.6 shows the results of this benchmark, which are very similar to the results of the dynamic analysis we previously analyzed for the CRC benchmark.

   We are not able to get the results for the *Baseline* algorithm since it is a dynamic analysis. As we did for the first setup of the CRC benchmark, we can use Equation 8.1 to analytically calculate that the *Baseline* algorithm would execute $1.7 \cdot 10^{12}$ instructions, since n_{ops} is $3.82 \cdot 10^5$ for FFT. Considering that in our configuration we were able to reach a maximum speed of $1.8 \cdot 10^4$ *instructions/s*, it would take $9.4 \cdot 10^7$ seconds to complete the analysis, which are 1093 days. Moreover, as we previously stated, *Baseline* finds a number of inconsistencies which can not be lower than the one found by *Dummy ScEpTIC*, since this last one is an optimization of the *Baseline* algorithm.

   For these reasons, and accordingly to the results present in Figure 8.6, we can say that the *Baseline* algorithm is largely outperformed by the other two algorithms.

   Let us now focus on the performance of `ScEpTIC` and *Dummy ScEpTIC*. As we can see in Figure 8.6a, `ScEpTIC` is significantly faster than *Dummy ScEpTIC* for performing the overall analysis. This result is similar to the one of the dynamic analysis we performed for CRC, and the reasons of such better performance are the same. In fact, the cause of the higher execution time of *Dummy ScEpTIC* is the excessive number of tested checkpoints, as we can see in Figure 8.6f. As we explained in the description of the CRC analysis, `ScEpTIC` is able to test a lower number of checkpoints thanks to its optimizations.

   A direct consequence of the higher number of tested checkpoints is a higher number of instructions executed, as we can see in Figure 8.6b.

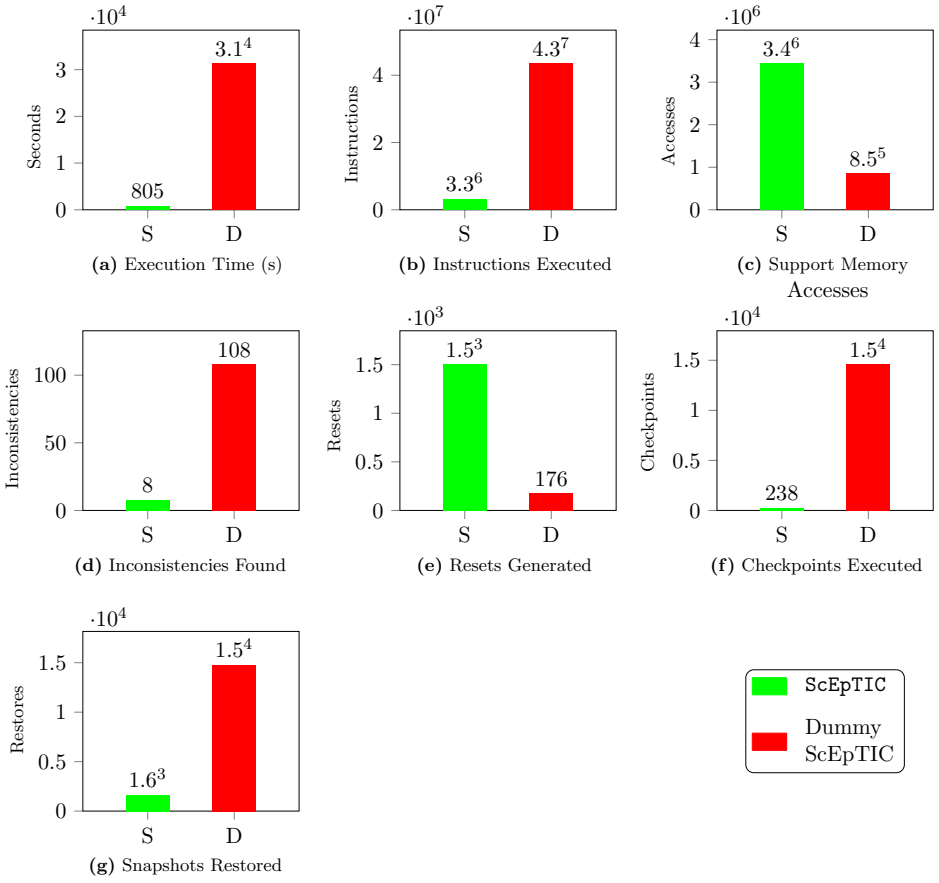

Figure 8.6: Data evaluation results: Dynamic Analysis of FFT benchmark with variables *realin*, *imagin*, *realout*, and *imagout* allocated in NVM.

Moreover, as we stated in the dynamic analysis we performed in CRC, another cause of such higher number of tested checkpoints causes also a higher number of restored snapshots, as we can see in Figure 8.6g.

Let us now consider Figure 8.6e, which shows the resets generated by the two algorithms. As we can see, *Dummy ScEpTIC* generates a lower number of power resets. As we explained for the dynamic setup of the CRC benchmark, this is caused by both its representation of inconsistencies and the number of inconsistencies it finds. As consequence, it tests a lower number of intermittent execution scenarios, as in the CRC case.

Finally, let us consider Figure 8.6d, which shows the number of inconsistencies found by the two algorithms. As we can see, *Dummy ScEpTIC* finds 108 inconsistencies. As we explained in the CRC bench-

marks, the representation of inconsistencies of *Dummy ScEpTIC* is not as precise as the one of `ScEpTIC`. As consequence, if we analyze all the 108 inconsistencies found by *Dummy ScEpTIC*, we will end up with the 8 inconsistencies found by `ScEpTIC`.

The verdict of this dynamic evaluation for the FFT benchmark is the same of the CRC one. The baseline algorithm is outperformed by both *Dummy ScEpTIC* and `ScEpTIC`. `ScEpTIC` performance is higher with respect to the one of *Dummy ScEpTIC*, since it not only takes a significant lower amount of time for running the analysis, but it also returns us a more concise information about inconsistencies. Moreover, *Dummy ScEpTIC* is able to outperform the *Baseline* algorithm thanks to the optimizations which are taken from the knowledge present in `ScEpTIC`.

2. *Dynamic analysis with global variables allocated in NVM.*

   Figure 8.7 shows the results of this benchmark. Its setup differs from the previous one only by the number of variables allocated into NVM. The additional variables allocated into NVM do not introduce any new inconsistency, as we can see in Figure 8.6d and Figure 8.7d. Moreover, the comparison of the different metrics gives us the same conclusion of the previous benchmark setup, and also the same of the dynamic analysis we performed with CRC.

   In fact, since it is a dynamic analysis, *Baseline* algorithm is largely outperformed by the other two algorithms, for the same reasons we explained in the previous dynamic analysis of FFT.

   As we can see in Figure 8.7a, *Dummy ScEpTIC* takes a substantial higher amount of time to complete the analysis with respect to `ScEpTIC`. As for the previous benchmark, it is caused by the higher number of checkpoints tested by *Dummy ScEpTIC*, which are shown in Figure 8.7f. A consequence of such higher number of tested checkpoints is the high number of snapshots restored, as we can see in Figure 8.7g.

   Finally, let us now consider Figure 8.6d and Figure 8.7d, which shows us the inconsistencies found by the two algorithms in the two different benchmarks. `ScEpTIC` finds the same number of inconsistencies, meaning that the newly allocated variables into NVM do not introduce any new inconsistency. Instead, *Dummy ScEpTIC* finds more inconsistencies with respect to the previous benchmark. This increased number of inconsistencies tells us that *Dummy ScEpTIC* miss-classifies some memory writes into NVM, and considers them to generate an inconsistency. As consequence, the result provided by the analysis of *Dummy ScEpTIC* is not only less precise with respect to the one of `ScEpTIC`, but also contains miss-leading information about inconsistencies.

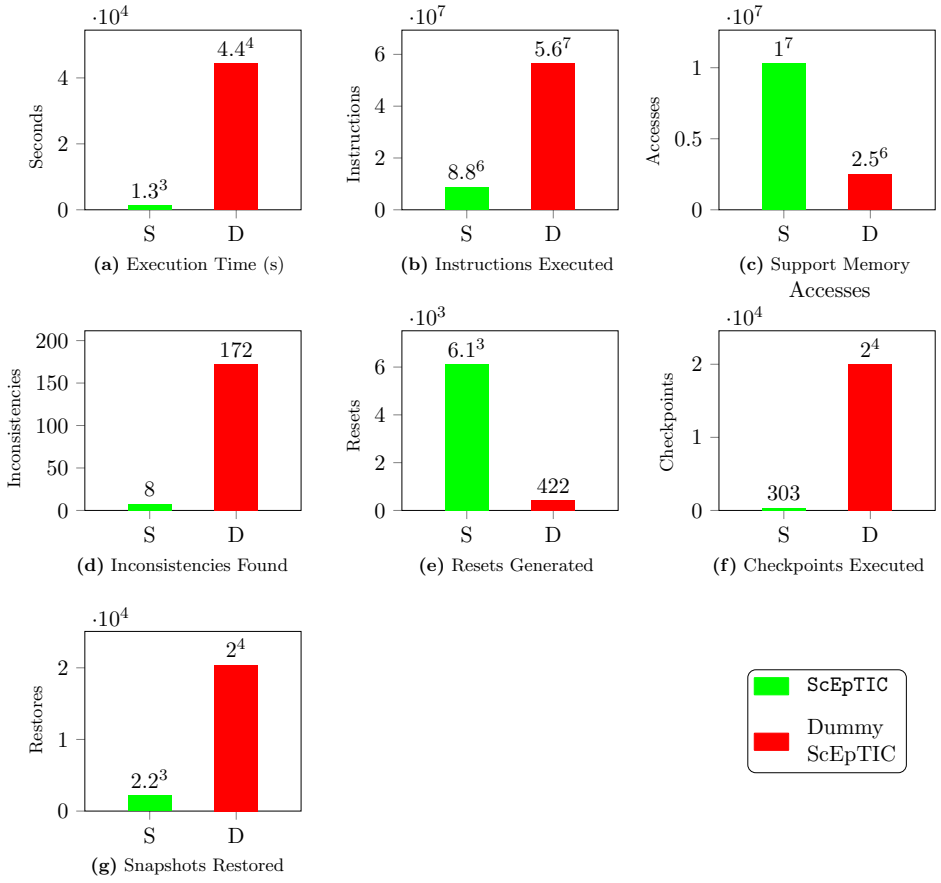

Figure 8.7: Data evaluation results: Dynamic Analysis of FFT benchmark with global variables allocated in NVM.

As for the previous dynamic analysis of CRC and FFT, `ScEpTIC` performance is higher with respect to the other two algorithms, and outperforms them both from an execution time and provided information standpoints.

3. *Static analysis with global variables allocated in NVM and checkpoints placed accordingly to the loop-latch strategy of MementOS [3].*
Figure 8.8 shows the results of this benchmark. *Baseline* is largely outperformed by the other two algorithms. In fact, it has a significantly higher execution time and number of instructions executed, as we can see in Figure 8.8a and Figure 8.8b. Moreover, as we previously explained, it represents inconsistencies in the same way as *Dummy ScEpTIC* does. For this reason, the higher number of inconsistencies *Baseline* finds consists in redundant information or miss-classified inconsistencies.

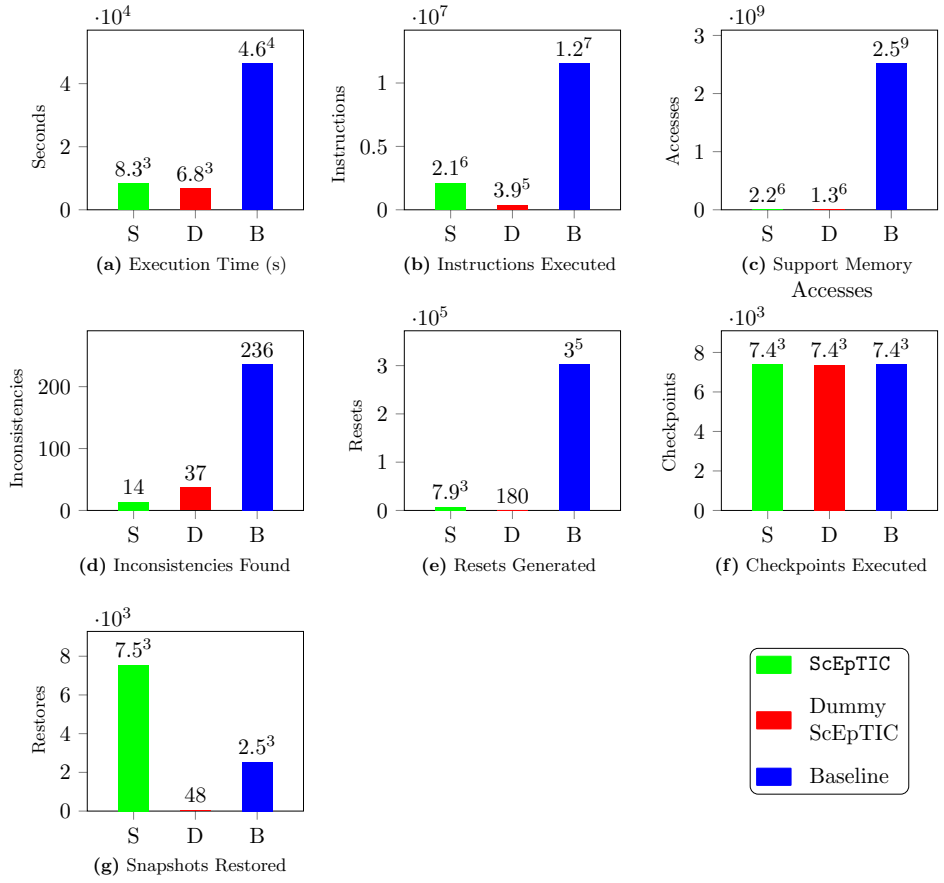

Figure 8.8: Data evaluation results: Static Analysis of FFT benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and global variables allocated in NVM.

Let us now consider Figure 8.8a, which shows the execution time required for running the analysis. *Dummy ScEpTIC* is faster than `ScEpTIC`, but their execution time has the same order of magnitude, even if the number of instruction executed is significantly higher in `ScEpTIC`, as we can see in Figure 8.8b. The higher number of instruction executed by `ScEpTIC` is caused by both the number of resets it generates and snapshots it restores, as we can see in Figure 8.8e and Figure 8.8g. Both the two metrics are largely higher in `ScEpTIC`, but the execution time is not affected significantly by such high difference. As we previously stated, restoring a snapshot does not affect significantly the execution time, since it produces a sequential execution of the code. As consequence, the significantly higher number of snapshots restored by `ScEpTIC` does not influence the execution time, since the following

instructions are executed sequentially. The high difference of resets generated between `ScEpTIC` and *Dummy ScEpTIC* does not influence the execution time as much as we might think. `ScEpTIC` generates a 43 times higher number of resets with respect to the one generated by *Dummy ScEpTIC*, but the execution time is in the same order of magnitude.

The reason for this discrepancy resides in the support memory accesses. Let us consider Figure 8.8c, which shows the support memory accesses of the three algorithms. The accesses performed by *Dummy ScEpTIC* and `ScEpTIC` are in the same order of magnitude, but the performance of *Dummy ScEpTIC* is significantly lower with respect to `ScEpTIC`. For understanding this statement, we must consider when the memory is accessed: `ScEpTIC` accesses the support memory during the execution of every instruction, and *Dummy ScEpTIC* accesses it only when it performs a reset. As consequence, we can notice that the usage ratio of the support memory is higher in *Dummy ScEpTIC*. In fact, it performs 180 resets with $1.3 \cdot 10^6$ memory accesses, which means $7.2 \cdot 10^3$ support memory accesses per reset. Instead, `ScEpTIC` executes $2.1 \cdot 10^6$ instructions with $2.2 \cdot 10^6$ memory accesses, which means 1.05 memory accesses per instruction. For this reason, the overhead introduced by *Dummy ScEpTIC* for verifying the presence of inconsistencies is much higher with respect to the one of `ScEpTIC`. As consequence, *Dummy ScEpTIC* has a performance similar to the one of `ScEpTIC`, even if the latter generates a 43 times higher number of resets. We encountered a similar condition in the second setup of the CRC benchmark, in which we run a static analysis with the stack allocated into NVM.

Let us now consider Figure 8.8d, which shows the number of inconsistencies found by the algorithms during the analysis. *Dummy ScEpTIC* finds a number of inconsistencies which is higher with respect to `ScEpTIC`. As we previously stated, the information returned by *Dummy ScEpTIC* is not as precise as the one of `ScEpTIC`, and it also contains redundant elements or miss-classified ones. If we analyze the information returned by *Dummy ScEpTIC*, we will achieve the same result returned by `ScEpTIC`.

Finally, the *Baseline* algorithm is largely outperformed by both the other two algorithms. *Dummy ScEpTIC* and `ScEpTIC` have a similar performance, but the overhead introduced by the state comparison of *Dummy ScEpTIC* slows it down significantly. `ScEpTIC` returns us a precise information, at the cost of a slight increase of the execution time. Moreover, we must always keep into consideration that *Dummy ScEpTIC* is the optimization of the *Baseline* algorithm obtained with the knowledge taken from `ScEpTIC`. For this reason, we can also say

that the optimizations of `ScEpTIC` are very effective for analyzing this problem.

4. *Static analysis with stack allocated in NVM and checkpoints placed accordingly to the loop-latch strategy of MementOS [3].*

   Figure 8.9 shows the results of this evaluation, which are very similar to the ones of the static analysis we performed with CRC.

   Let us firstly consider Figure 8.9b, which shows the number of instructions executed by the three algorithms. As we can see, `ScEpTIC` executes a number of instructions which is significantly higher with respect to the other two algorithms. If we consider the execution time shown in Figure 8.9a, we can see a case similar to the previous setup of this benchmark. The time required by `ScEpTIC` for executing the analysis is comparable to the one required by the other two algorithms, even if it executes a number of instructions which is 4 to 10 times higher.

   The reason for this behavior is the same we described in the previous benchmark setup, and it resides in the support memory accesses. For better understanding this condition, we can compare `ScEpTIC` and *Baseline* that takes the same amount of time for completing the analysis. The number of resets they perform is similar, as we can see in Figure 8.9e, but the number of instruction executed is 4 times higher in `ScEpTIC`. *Baseline* and *Dummy ScEpTIC* uses the same technique for analyzing inconsistencies, and thus they both accesses the support memory when they generate a reset. `ScEpTIC` performs 1.08 support memory accesses for any executed instruction, and instead *Baseline* performs $8.2 \cdot 10^4$ support memory accesses every time it generates a reset. As consequence, generating an intermittent execution has a higher overhead in the *Baseline* algorithm with respect to `ScEpTIC`.

   If we consider both Figure 8.9b and Figure 8.9a, we can see an interesting relation between the instructions executed and the execution time for *Dummy ScEpTIC* and *Baseline* algorithms. In fact, the execution time of *Baseline* is almost twice the one of *Dummy ScEpTIC*, and the instructions executed by *Baseline* are also almost twice with respect to *Dummy ScEpTIC*. This condition is only valid for these two algorithms, since they introduce the same type of overhead for finding inconsistencies.

   As we stated for the previous benchmarks, the high number of instructions executed by `ScEpTIC` does not affect the execution time as much as it does for *Dummy ScEpTIC* or *Baseline*. In fact, a relevant part of such instructions are executed with a sequential execution, since `ScEpTIC` restores a high number of snapshots, as we can see in Figure 8.9g.

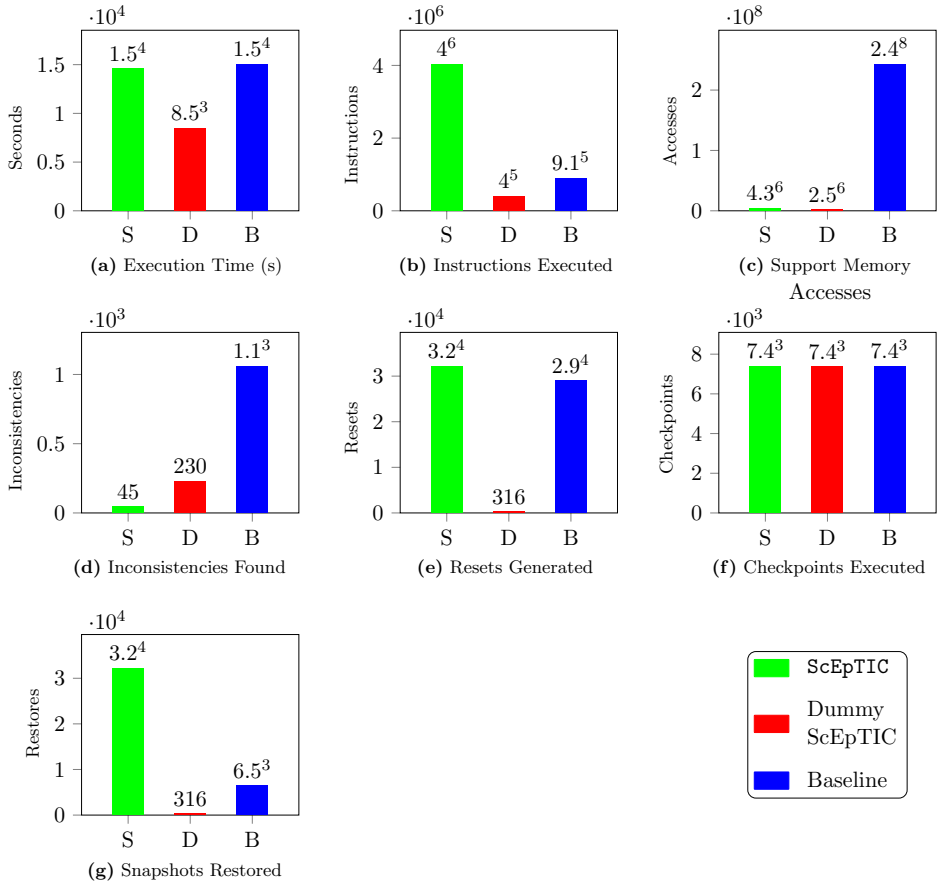

Figure 8.9: Data evaluation results: Static Analysis of FFT benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM.

Let us now consider Figure 8.9d, which shows the number of inconsistencies found by the three algorithms. *Baseline* and *Dummy ScEpTIC* have the same representation for inconsistencies, and thus the information returned by *Baseline* is full of redundant elements with respect to the one returned by *Dummy ScEpTIC*. Furthermore, the $1.1 \cdot 10^3$ inconsistencies returned by *Baseline* would be very difficult to analyze. *Dummy ScEpTIC* returns 230 inconsistencies and, as we previously stated, they can all be reduced to the one present in the results of `ScEpTIC`, which are only 45. As we stated for all the other benchmarks, `ScEpTIC` returns an information which is precise and more concise with respect to the other two algorithms.

The *Baseline* algorithm and `ScEpTIC` take the same amount of time for running this benchmark, but `ScEpTIC` returns a more valuable information which also takes less efforts to be analyzed. For this reason, we can say that `ScEpTIC` has a better performance with respect to *Baseline*, especially thanks to the way in which it analyzes and represents inconsistencies. *Dummy ScEpTIC* is an optimization of the *Baseline* algorithm, and in this benchmark outperforms it, thanks to the optimizations taken from `ScEpTIC`. Finally, *Dummy ScEpTIC* takes less time with respect to `ScEpTIC` for running the analysis, but the efforts required for analyzing the results returned by *Dummy ScEpTIC* will vanify this performance advantage. In fact, as we previously stated, `ScEpTIC` returns us a precise information which does not contain any redundant element, and directly identifies the causes of the found inconsistencies.

The overall result of this four benchmarks is similar to the one of the CRC benchmark. Considering the performance of `ScEpTIC`, we can state that it largely outperforms *Dummy ScEpTIC* and *Baseline* in the two *Dynamic Analysis*. Instead, in the *Static Analysis* it has a comparable performance from an execution standpoint, but the concise and precise information it returns regards inconsistencies makes its performance to be better than the other two algorithms. In fact, the efforts required for analyzing the results returned by *Dummy ScEpTIC* or *Baseline* vanify any execution time advantage, especially if we consider that we are performing an off-line analysis of a program. Moreover, in all the four benchmarks *Dummy ScEpTIC* has a significantly better performance with respect to *Baseline*, and this is achieved thanks to the optimizations taken from `ScEpTIC`. For this reason, we can also say that the optimizations of `ScEpTIC` are very effective for analyzing this problem.

AES:

1. *Dynamic analysis with global variables allocated in NVM.*

   Figure 8.10 shows the results of this benchmark setup. This evaluation scenario returns us a result which is similar to the dynamic analysis we performed for the CRC and FFT benchmarks.

   As we previously explained, the *Baseline* algorithm is not able to complete a dynamic analysis in a reasonable amount of time. For this reason, we analytically calculated with Equation 8.1 that the *Baseline* algorithm would execute $2.99 \cdot 10^{12}$ instructions, since n_{ops} is $6.7 \cdot 10^5$ for AES. Considering that in our configuration we were able to reach a maximum speed of $1.8 \cdot 10^4$ *instructions/s*, it would take $1.66 \cdot 10^8$ seconds to complete the analysis, which are 1922 days.

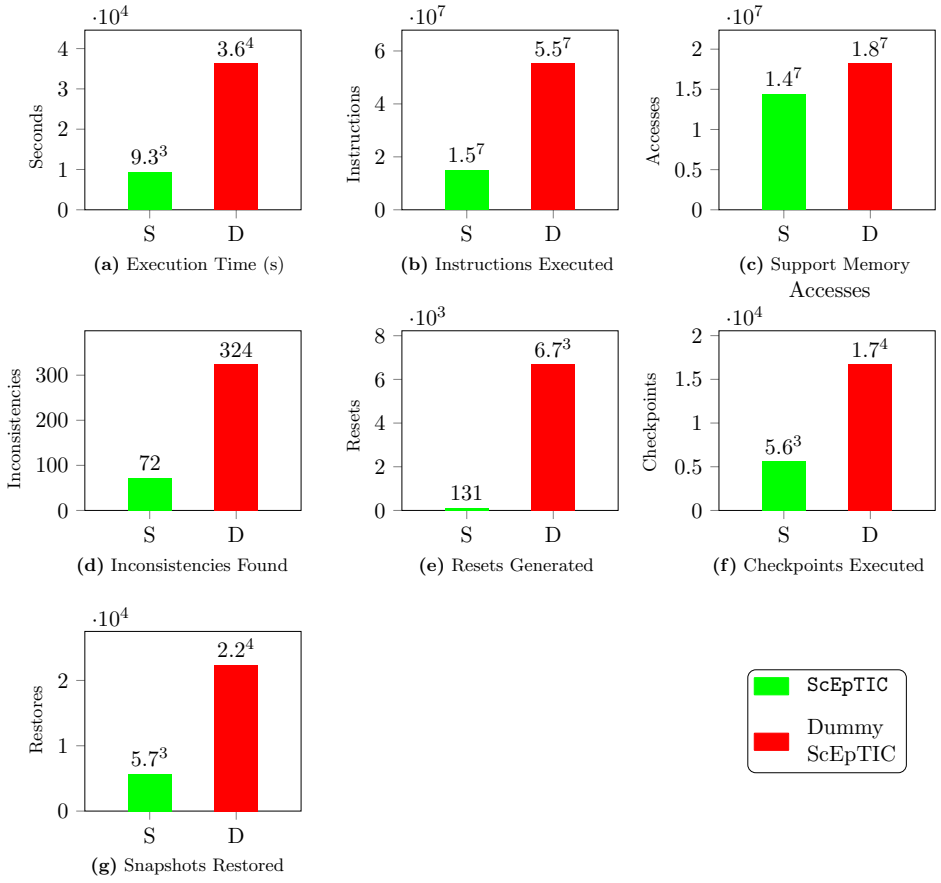

Figure 8.10: Data evaluation results: Dynamic Analysis of AES benchmark with global variables allocated in NVM.

As consequence, we can say that the *Baseline* algorithm is largely outperformed by `ScEpTIC` and *Dummy ScEpTIC*.

In this benchmark `ScEpTIC` is able to outperform *Dummy ScEpTIC* in all the metrics thanks to its optimizations and to its representation of inconsistency. This same behavior is shown in the two dynamic analysis setups of the CRC and FFT benchmarks we previously described.

In fact, as we can see in Figure 8.10a, `ScEpTIC` is significantly faster than *Dummy ScEpTIC* for performing the overall analysis. As we explained in the other dynamic analysis benchmarks, Figure 8.10f shows the cause of the higher execution time of *Dummy ScEpTIC*, and it is the excessive number of tested checkpoints. A direct consequence of this behavior is a higher number of instructions executed, as we can see in Figure 8.10b.

Let us now consider Figure 8.10d, which shows the number of inconsistencies found by the two algorithms. We can notice that *Dummy ScEpTIC* finds 324 inconsistencies, and `ScEpTIC` finds only 72 of them. As we explained in previous benchmarks, the representation of inconsistencies of *Dummy ScEpTIC* is not as precise as the one of `ScEpTIC`. As consequence, if we analyze all the 324 inconsistencies found by *Dummy ScEpTIC*, we will end up with the 72 inconsistencies found by `ScEpTIC`.

The verdict of this dynamic evaluation for the AES benchmark is the same of the CRC and FFT ones. The baseline algorithm is outperformed by both *Dummy ScEpTIC* and `ScEpTIC`. `ScEpTIC` has a better performance with respect to *Dummy ScEpTIC*, since it not only takes a significant lower amount of time for running the analysis, but it also returns us a more concise information about inconsistencies. Moreover, *Dummy ScEpTIC* is able to outperform the *Baseline* algorithm thanks to the optimizations which are taken from the knowledge present in `ScEpTIC`.

2. *Static analysis with global variables allocated in NVM and checkpoints placed accordingly to the loop-latch strategy of MementOS [3].*

   Figure 8.11 shows the results of this evaluation, and the result are similar to static analysis with the global variables allocated in NVM which we performed for the FFT benchmark.

   The *Baseline* algorithm is largely outperformed by *Dummy ScEpTIC* and `ScEpTIC` in all the metrics, except for the number of checkpoints tested, which is fixed in the code since we performed a static analysis. The main cause of such bad performance of the *Baseline* algorithm is the lack of optimizations for the resets to be generated, which instead are present in both `ScEpTIC` and *Dummy ScEpTIC*. For this reason, *Baseline* generates an unnecessary number of resets, as we can see in Figure 8.11e. Moreover, as we stated in the analysis of previous benchmarks, resets are used for generating a specific intermittent execution of the code, and thus a high number of resets leads to a high number of instructions executed, as Figure 8.11b shows. The higher number of resets generates also a higher number of support memory accesses, since *Baseline* verifies the presence of inconsistencies each time it generates a reset. If we consider Figure 8.11c, we can see that the support memory accesses executed by *Baseline* are significantly higher with respect to the other two algorithms. Those are the reasons for the higher execution time required by *Baseline* for completing the analysis.

   Let us now focus on the performance of *Dummy ScEpTIC* and `ScEpTIC`. As we can see in Figure 8.11a, they take almost the same amount

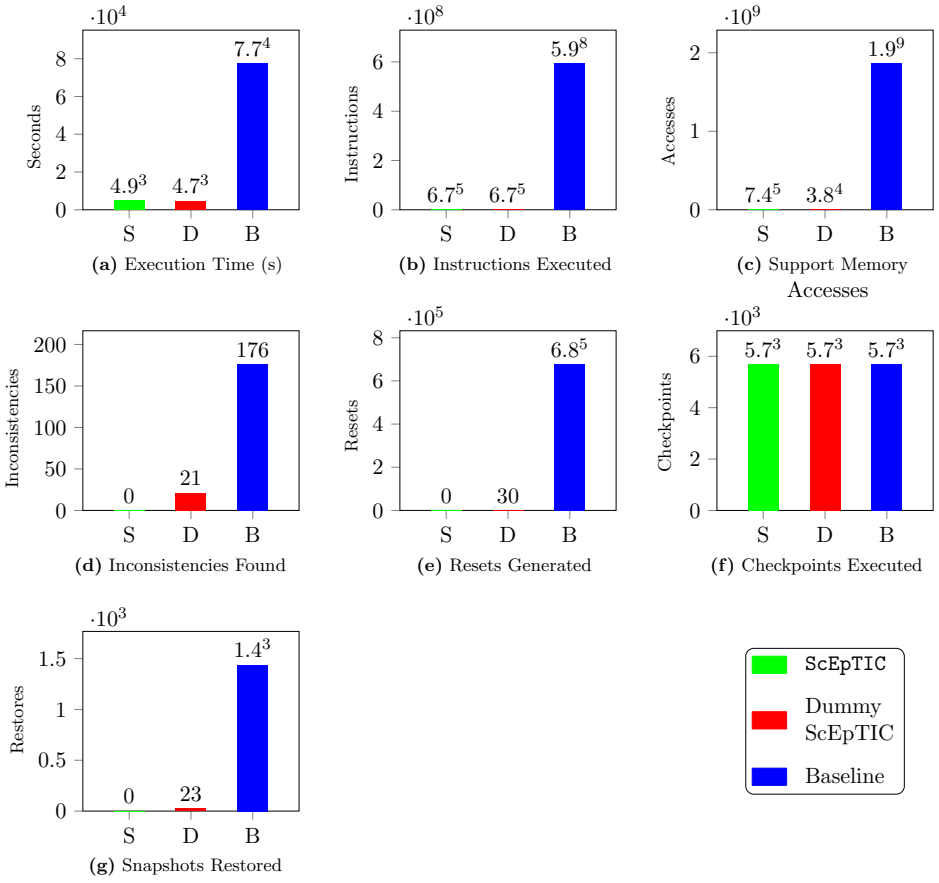

Figure 8.11: Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and global variables allocated in NVM.

of time for performing the analysis, and the number of instructions executed is also the same for both algorithms, as Figure 8.11b shows. The little difference in the execution time is caused by the support memory accesses. In fact, as we can see in Figure 8.11c, `ScEpTIC` executes almost twice the memory accesses of *Dummy ScEpTIC*. As we stated for previous benchmarks, `ScEpTIC` performs a support memory access every time it runs an instruction, leading to an access ratio of 1.1 support memory accesses per instruction. Instead, *Dummy ScEpTIC* perform a memory access every time it generates a reset, leading to an access ratio of 1227 support memory accesses per resets generated. When *Dummy ScEpTIC* generates a reset, it introduces an overhead for accessing the support memory, and it is significantly higher with respect to the one of `ScEpTIC`.

Let us now consider Figure 8.11e and Figure 8.11g, which shows respectively the resets generated and the snapshots restored. Thanks to its optimizations, `ScEpTIC` does not generate any resets and, as consequence, it does not restore any snapshot. Instead, *Dummy ScEpTIC* generates 30 resets, which leads it to restore 23 snapshots. Those numbers do not introduce a measurable overhead, and thus the performance of *Dummy ScEpTIC* is not influenced from an execution standpoint, but it is influenced from the point of view of inconsistencies. In fact, as we stated in Chapter 4, `ScEpTIC` performs a reset only after operations that write into memory cells allocated into NVM, and that are read from a previously executed instruction. Instead, *Dummy ScEpTIC* executes a reset after every instruction which writes the NVM. As consequence, the lack of such optimization leads to the miss-classification of 21 inconsistencies, as we can see in Figure 8.11d. In fact, `ScEpTIC` does not find any inconsistency, since it is able to verify if the possibly inconsistent value preset in memory is used for producing an inconsistent result. We can also notice that the lack of such optimization makes also the *Baseline* algorithm to miss-classify 176 different inconsistencies.

Finally, for the reasons we explained, we can say that `ScEpTIC` performance is comparable with the one of *Dummy ScEpTIC*, from an execution standpoint, since they perform an analysis which almost consists in a sequential execution of the code. If we consider an inconsistency standpoint, *Dummy ScEpTIC* returns an incorrect result, since it miss-classifies all the inconsistencies. Fixing the inconsistencies that *Dummy ScEpTIC* returns using the techniques we explained in Chapter 4 does not produce any benefit, since the code does not have any inconsistency. In fact, we may increase the overhead due to the introduction of a new checkpoint, or we may waste our time moving checkpoints that already grant consistency, with the risk of introducing an inconsistency that the previous placement avoided.

For these reasons, we can say that `ScEpTIC` overall performance is better than the one of *Dummy ScEpTIC*. Moreover, *Dummy ScEpTIC* is able to outperform *Baseline* thanks to the optimizations taken from the knowledge present in `ScEpTIC`. For these reasons, we can say that `ScEpTIC` is very effective in analyzing the problem of inconsistencies in this benchmark.

3. *Static analysis with global variables allocated in NVM and checkpoints placed accordingly to the function-return strategy of MementOS [3].*

   Figure 8.12 shows the results of this benchmark, and the differences between the metrics is exactly the same of the previous benchmark. In fact, *Baseline* is outperformed by `ScEpTIC` and *Dummy ScEpTIC*

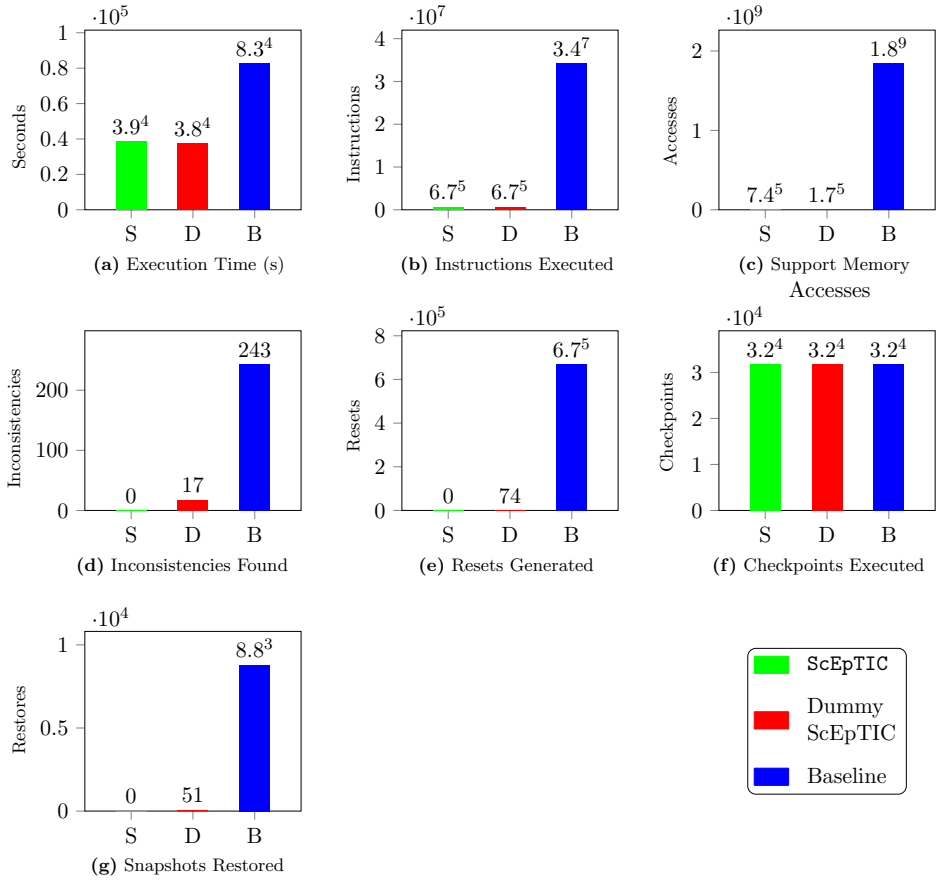

Figure 8.12: Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *function-return* strategy of MementOS [3] and global variables allocated in NVM.

in any metric. `ScEpTIC` and *Dummy ScEpTIC* have almost the same performance from an execution standpoint but, as for the previous benchmark, *Dummy ScEpTIC* produces a result that contains only miss-classified inconsistencies. For this reason, we can not use such result and thus we can say that the performance of `ScEpTIC` is better than the one of *Dummy ScEpTIC*, as for the previous benchmark.

4. *Static analysis with stack allocated in NVM and checkpoints placed accordingly to the loop-latch strategy of MementOS [3].*

   Figure 8.13 shows the results of this benchmark, which are similar to the static analysis of FFT with the stack allocated into NVM.

   The first thing we can notice is that `ScEpTIC` takes more time for running this benchmark with respect to the other two algorithms, as

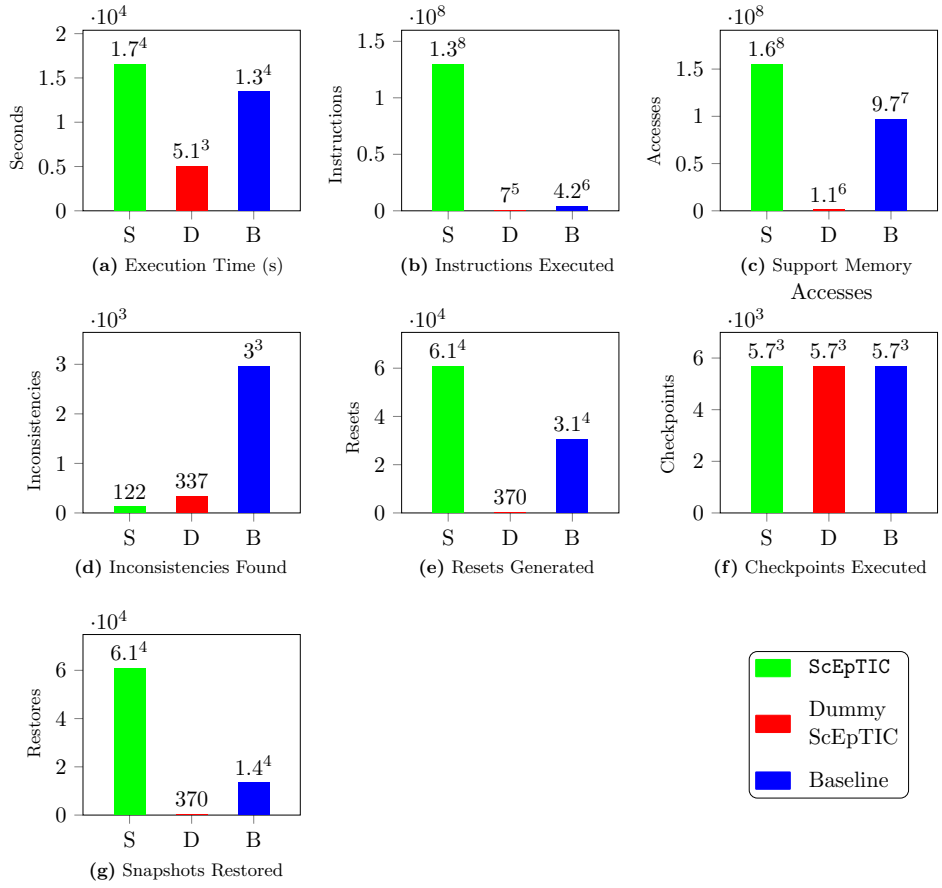

Figure 8.13: Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM.

we can see in Figure 8.13a. The main reason of the higher execution time is the number of resets generated, as Figure 8.13e shows. In fact, `ScEpTIC` generates twice the number of resets generated by *Baseline* and 165 times the number of resets generated by *Dummy ScEpTIC*. The reason of the lower number of resets generated resides in the way *Baseline* and *Dummy ScEpTIC* finds and represents inconsistencies. As we stated in the previous benchmarks, they are not able to understand the actual cause of the inconsistency, and they represent inconsistencies as pairs of checkpoints and reset points causing the found inconsistency. Once they find an inconsistency, they will skip the generation of the same reset point in future analysis of the same checkpoint. The *loop-latch* placement of *MementOS* [3] which is used in this benchmark places checkpoints at the end of the loop

body. As consequence, a relevant number of resets is ignored, since the algorithms would find the same inconsistency over again. As we explained in Section 8.2.1, since `ScEpTIC` correctly identifies the actual cause of inconsistencies, skipping a reset point would lead to an inaccurate analysis.

The higher number of resets generated by `ScEpTIC` makes it test a higher number of different intermittent executions. As consequence, the number of instruction executed by `ScEpTIC` is higher with respect to the other two algorithms, as we can see in Figure 8.13b. The number of instructions executed by `ScEpTIC` is more than 30 times higher than the other two algorithms, but the execution time is comparable to the one of the *Baseline* algorithm and only 3 times higher with respect to *Dummy ScEpTIC*. The reason of this fact resides both in the snapshots restored, and in the support memory accesses. In fact, `ScEpTIC` tests a higher number of resets, and as consequence it also restores a higher number of snapshots, as we can see in Figure 8.13g. As we stated in the analysis of the other benchmarks, restoring a snapshot causes a subsequent sequential execution of the code, which is run at the maximum possible speed, since it is not interrupted. As consequence, such higher number of snapshots restored has a high impact on the instruction executed, which on the other hand does not impact significantly on the execution time.

For understanding the contribution of support memory accesses to the execution time, we have to consider the way in which such memory is accessed, as we did for the other static setups of the previous benchmarks. `ScEpTIC` accesses the support memory every time it executes an instruction. Instead, *Dummy ScEpTIC* and *Baseline* access the support memory every time they perform a reset. Let us consider Figure 8.13c, which shows the support memory accesses of the three algorithms. `ScEpTIC` performs $1.6 \cdot 10^8$ accesses with $1.3 \cdot 10^8$ instructions executed, and thus its access rate is 1.23 support memory accesses per instruction. Instead, *Dummy ScEpTIC* performs $1.1 \cdot 10^6$ accesses with 370 resets, and thus its access rate is 2973 support memory accesses per reset. Similarly, *Baseline* access rate is 3129 support memory accesses per reset. The overhead paid by `ScEpTIC` for accessing the support memory is several times lower with respect to the other two algorithms. Moreover, every time *Dummy ScEpTIC* and *Baseline* generate a reset, they pay such overhead, leading to a much slower intermittent execution. As consequence, the execution time of `ScEpTIC` is within the same order of magnitude of the other two algorithms, even if `ScEpTIC` executes and tests a higher number of instructions, as we can see in Figure 8.13a.

Let us now consider Figure 8.13d, which shows the number of inconsistencies identified by each algorithm. As we stated for the other benchmarks, `ScEpTIC` finds the actual cause of inconsistencies, and thus it does not produce redundant information in its results. Instead, *Dummy ScEpTIC* and *Baseline* have the same representation of inconsistencies, which makes them unable to identify the actual cause of inconsistencies. They only verify if the state is different with respect to when the checkpoint was taken, but they do not analyze if there is an operation which uses such inconsistent value. As consequence, they only find the instructions which modify the NVM, but they are unable to identify if the alteration produces wrong results, nor the instruction using such altered value. As consequence, the results returned by *Dummy ScEpTIC* and *Baseline* is full of redundant elements and is also subjected to miss-classification of inconsistencies. *Dummy ScEpTIC* is an optimization of *Baseline*, and they find the same kind of inconsistencies. *Baseline* finds a higher number of inconsistencies, since it does not have the same optimizations to reset points that *Dummy ScEpTIC* has. For this reason, it finds a higher number of resets causing an inconsistent state, but if we analyze such results, we will end up with the same one present in *Dummy ScEpTIC*. Moreover, if we analyze the results returned by *Dummy ScEpTIC* along with the benchmark code, we will be able to identify the actual cause of the inconsistencies, and we will end up with the same information returned by `ScEpTIC`.

The result returned by `ScEpTIC` is more manageable with respect to *Dummy ScEpTIC* and *Baseline*, and it is also precise about the actual cause of inconsistencies. The amount of time required by `ScEpTIC` for performing the analysis is in the same order of magnitude with respect to *Baseline*. The efforts required for analyzing the result produced by *Baseline* are higher with respect to the execution time difference with `ScEpTIC`. For this reason, we can say that `ScEpTIC` has a better overall performance with respect to *Baseline*. `ScEpTIC` takes 3 times more the amount of time required for *Dummy ScEpTIC* for completing the analysis, but it also returns 3 times less the number of inconsistencies. The effort required for analyzing the result returned by *Dummy ScEpTIC* are higher with respect to the time difference with `ScEpTIC`, especially if we consider that we do not have the information about which are the actual cause of inconsistencies. Moreover, we are performing an offline analysis to the code, and since the information returned by `ScEpTIC` is more precise and concise, waiting for its completion is worth, especially if we consider the effort required for analyzing the 337 inconsistencies returned by *Dummy ScEpTIC*.

We can notice that *Dummy ScEpTIC* outperforms *Baseline* in all the metrics. This is achieved thanks to the optimizations taken from the knowledge of `ScEpTIC`. As result, we can say that the guidelines we developed alongside with `ScEpTIC` for analyzing inconsistencies are very effective with respect to the ones present in *Baseline*.

Finally, `ScEpTIC` overall performance is higher with respect to the other two algorithms, thanks to the quality of the information it returns. In fact, the time difference between `ScEpTIC` and the other two algorithms is not comparable to the effort required for extracting the actual cause of inconsistencies from their results.

5. *Static analysis with stack allocated in NVM and checkpoints placed accordingly to the function-return strategy of MementOS [3].*

   Figure 8.14 shows the result of this evaluation. It is similar to the results of the second and third setup of this same benchmark, in which the global variables were allocated in the NVM.

   In Figure 8.14a we can notice that `ScEpTIC` and *Dummy ScEpTIC* takes almost the same amount of time for running the analysis, and instead *Baseline* is slower. If we now consider Figure 8.14b, we can see that the difference between the instructions executed by the three different algorithms is not reflected over the execution time in the same way. In fact, as we observed for all the other benchmarks, the execution time is highly influenced by the number of resets generated, snapshots restored, and the ratio of support memory accesses. Moreover, the resets generated and snapshots restored both influences the instructions executed, but their effects over the execution time is different. In fact, the former produces an intermittent execution of the code, which is more time-consuming, and instead the latter produces a sequential execution, which is less time-consuming.

   Baseline and `ScEpTIC` executes almost the same number of instructions, but the former takes almost twice for executing the analysis. As we previously stated, this is caused by the higher number of resets generated by *Baseline*, as we can see in Figure 8.14e. Each time *Baseline* performs a reset, it also verifies the state for finding inconsistencies. As consequence, the higher number of resets generated also produces a higher number of support memory accesses, as we can see in Figure 8.14c.

   Dummy ScEpTIC and `ScEpTIC` runs the benchmark in almost the same amount of time, but `ScEpTIC` executes 4 time the number of instructions executed by *Dummy ScEpTIC*. Moreover, *Dummy ScEpTIC* generates a lower number of resets and restores also a lower number of snapshots. As we stated for the other benchmarks, the reason of the comparable execution time resides in the support memory accesses

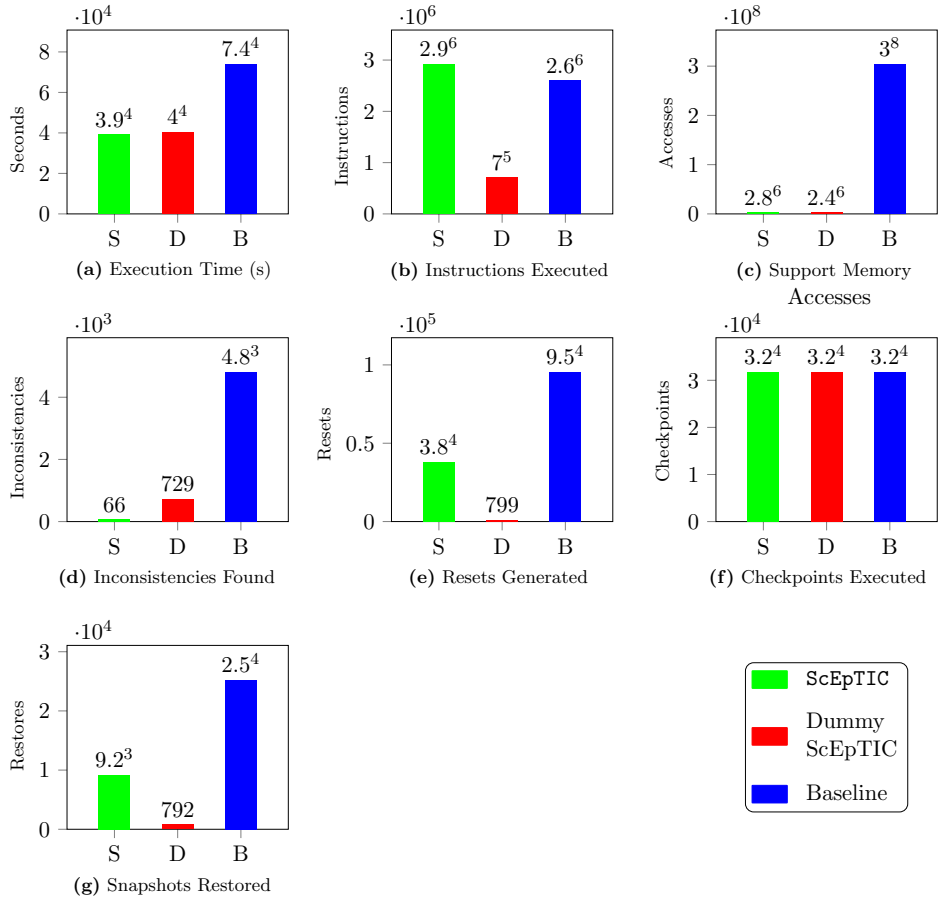

Figure 8.14: Data evaluation results: Static Analysis of AES benchmark, with checkpoints placed accordingly to the *function-return* strategy of MementOS [3] and the stack allocated in NVM.

and the consequent overhead introduces. In fact, `ScEpTIC` and *Dummy ScEpTIC* perform almost the same number of support memory accesses, but the overhead is significantly higher in *Dummy ScEpTIC*, since they slow down its performance. In fact, *Dummy ScEpTIC* accesses the support memory each time it performs a reset, leading to an overhead of 3004 support memory accesses per reset generated. Instead, `ScEpTIC` accesses the support memory during the normal execution, and the overhead is 0.97 support memory accesses per instruction executed. The access ratio is significantly lower in `ScEpTIC`, and the introduced overhead has almost no impact over the performance. Instead, *Dummy ScEpTIC* pays a significantly higher cost for accessing the support memory, and thus it has a high impact over the perfor-

mance. Moreover, we saw this behavior in all the other benchmarks, and it is caused by the way in which inconsistencies are identified by `ScEpTIC` and *Baseline*/*Dummy ScEpTIC*.

Let us now consider the number of inconsistencies returned by the three algorithms, which are shown in Figure 8.14d. As we explained, *Baseline* and *Dummy ScEpTIC* have the same representation of inconsistencies, and they do not find the actual cause of them, but only the resets points which produce a possibly inconsistent state. The excessive higher number of inconsistencies returned by *Baseline* is caused by the higher number of resets generated. If we analyze such inconsistencies, we can reduce them to the ones returned by *Dummy ScEpTIC*. Moreover, if we analyze the result returned by *Dummy ScEpTIC* we will end up with the same result of `ScEpTIC`, but the effort required for doing so is considerable. Considering that `ScEpTIC` and *Dummy ScEpTIC* takes the same time for running the analysis, this extra effort for analyzing the result returned by *Dummy ScEpTIC* can not be justified.

The final verdict of this benchmark is very similar to the one of the second and third benchmark of AES, which we previously described. *Baseline* is outperformed by the other two algorithms in almost all the metrics, and the result it returns is completely unmanageable. `ScEpTIC` and *Dummy ScEpTIC* takes the same amount of time for running the analysis, but the overall performance of `ScEpTIC` is considerably better, since it returns us more concise and precise information about inconsistencies. Finally, *Dummy ScEpTIC* has a better performance with respect to *Baseline* thanks to the optimizations taken from the knowledge present in `ScEpTIC`. In fact, it is thanks to such optimizations that *Dummy ScEpTIC* is able to perform a considerably lower number of resets with respect to *Baseline*, resulting in a higher overall performance.

The overall result of this five benchmarks is similar to the one of the CRC and FFT benchmarks. Considering the performance of `ScEpTIC`, we can state that it largely outperforms *Dummy ScEpTIC* and *Baseline* in the *Dynamic Analysis*. In fact, its ability in identifying the checkpoints to be tested permits it to perform a significant lower of checkpoints, resulting in a better performance.

Instead, in the *Static Analysis* it has a comparable performance from an execution standpoint. Since we are performing an offline analysis, we are interested in the quality of the information returned alongside the time efforts required for obtaining it. As consequence, the concise and precise information `ScEpTIC` returns regards inconsistencies makes its overall performance to be better than the other two algorithms. In fact, the efforts required for

analyzing the results returned by *Dummy ScEpTIC* or *Baseline* vanify any execution time advantage.

Moreover, in all the five benchmark setups *Dummy ScEpTIC* has a significantly better performance with respect to *Baseline*, as it did in all the benchmarks of CRC and FFT. As we described in Section 8.2.1, *Dummy ScEpTIC* algorithm consists in the *Baseline* algorithm optimized with a part of the knowledge we developed alongside `ScEpTIC`. Given the performance improvement of *Dummy ScEpTIC* over *Baseline* shown also in these benchmarks, we can also say that the optimizations of `ScEpTIC` are very effective for analyzing this problem.

Sense:

1. *Dynamic analysis with global variables allocated in NVM.*

   Figure 8.15 shows the results of this benchmark. We previously described the code of this benchmark in Example 8.3, and for this evaluation we used a constant value for the input. This evaluation scenario returns us a result which is similar to the dynamic analysis we performed for the CRC, FFT, and AES benchmarks.

   As we stated in the other dynamic analysis, we did not run this benchmark using the *Baseline* algorithm, since it would take an unreasonable amount of time for completing the analysis in such scenario. As for what we did for the other dynamic analysis, we can use Equation 8.1 to analytically calculate that the *Baseline* algorithm would execute $3.05 \cdot 10^9$ instructions, since n_{ops} is 2635 for Sense. Considering that in our configuration we were able to reach a maximum speed of $1.8 \cdot 10^4$ *instructions/s*, it would take $1.69 \cdot 10^5$ seconds to complete the analysis, which are almost 2 days. Moreover, as we stated previously, *Baseline* represents inconsistencies in the same way *Dummy ScEpTIC* does, and it has the same method of analysis. For this reason, it would not find a lower number of inconsistencies with respect to *Dummy ScEpTIC*, and we can also see this behavior in the previous benchmarks we run. For these reasons, and accordingly to the results present in Figure 8.15, we can say that the *Baseline* algorithm is largely outperformed in all the metrics by the other two algorithms.

   Let us now compare the performance of `ScEpTIC` and *Dummy ScEpTIC*. As we can see in Figure 8.15a, `ScEpTIC` takes less time to complete the analysis. This lower execution time is achieved thanks to the ability of `ScEpTIC` of recognizing where checkpoints must be tested. In fact, in Figure 8.15f we can see that `ScEpTIC` verifies a lower number of checkpoints with respect to *Dummy ScEpTIC*. In Figure 8.15e and Figure 8.15g we can see the benefits of this behavior. In fact, testing a lower number of checkpoints permits `ScEpTIC` to generate a lower

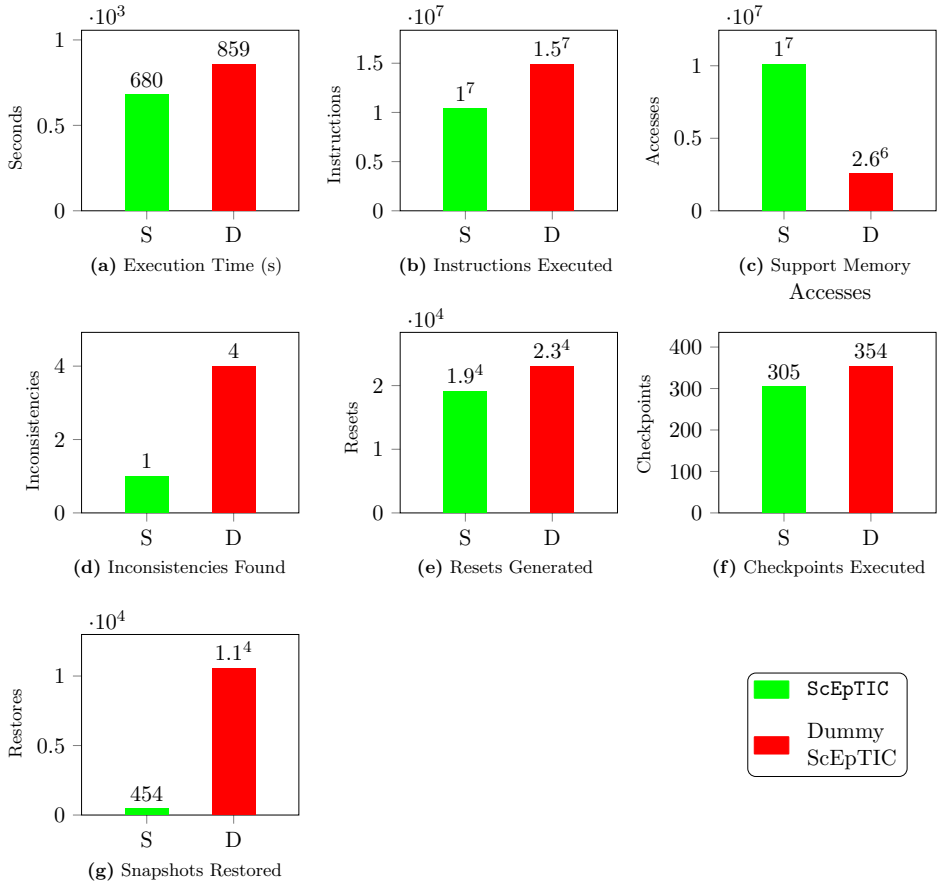

Figure 8.15: Data evaluation results: Dynamic Analysis of Sense benchmark with global variables allocated in NVM.

number of resets, since less intermittent executions are required, and as consequence it also generates a lower number of snapshots. Moreover, this also affects positively the instructions executed during the analysis, as we can see in Figure 8.15b.

The only metric in which *Dummy ScEpTIC* seems to perform better than `ScEpTIC` is the one measuring support memory accesses. As we did for the other benchmarks, for comparing the effects of this metric over the performance, we need to consider where support memory accesses happens. *Dummy ScEpTIC* verifies the presence of inconsistencies every time it generates a reset, and thus the access ratio is 113 support memory accesses per reset. Instead, `ScEpTIC` verifies the presence of inconsistencies while it executes the code, resulting in an access ratio of 1 support memory access per instruction executed. The

overhead introduced by support memory accesses is higher in *Dummy ScEpTIC*, and it is paid every time it generates a reset. As consequence, generating an intermittent execution is more expensive for *Dummy ScEpTIC*.

Let us consider Figure 8.15d, which shows the inconsistencies found by the two algorithms. `ScEpTIC` returns a single inconsistency, and identifies as its cause the re-execution of the instruction at line 16 of the code that Example 8.3 shows. Such instruction increments the global variable *sum*, and its re-execution will produce an inconsistent result. Instead, as we previously stated, *Dummy ScEpTIC* is not able to provide us the exact cause of the inconsistency, and it returns 4 possible combinations of checkpoint and reset points that lead to an inconsistency. If we analyze such combinations, we can see that *Dummy ScEpTIC* identifies the same inconsistency of `ScEpTIC`, but it also finds that the re-execution of the instruction at line 23 causes an inconsistency. Such inconsistency represents a miss-classification, since its re-execution produces always the same result and does not modify the runtime state. In fact, we used a constant value for the input, and thus the variance that the program computes is zero. The instruction of line 23 calculates the standard deviation of the data, which is always zero for our input, since the variance is zero. For this reason, the re-execution of the instruction of line 23 can not produce an inconsistent result, since it does not alter the runtime state, and thus *Dummy ScEpTIC* miss-classifies this inconsistency. As we demonstrated, the result returned by `ScEpTIC` has more value and provides us a precise information about the inconsistency. Moreover, it is able to condense all the information and it is not subjected to the problem of miss-classification that both *Baseline* and *Dummy ScEpTIC* have.

Finally, the verdict of this benchmark is the same of the other dynamic analysis we performed with CRC, FFT, and AES. In fact, we can say that the performance of `ScEpTIC` is better than *Dummy ScEpTIC*, especially thanks to both the execution time advantage and the concise information it returns. As we stated for all the other benchmarks, we must also consider that *Dummy ScEpTIC* is able to outperform *Baseline* thanks to the optimizations taken from the knowledge we developed alongside `ScEpTIC`. This result tells us one more time that the techniques we developed are very effective for analyzing this kind of problems.

2. *Static analysis with global variables allocated in NVM and checkpoints placed accordingly to the loop-latch strategy of MementOS [3].*

   Figure 8.16 shows the result of this benchmark. The differences between the metrics are almost identical with respect to the results of

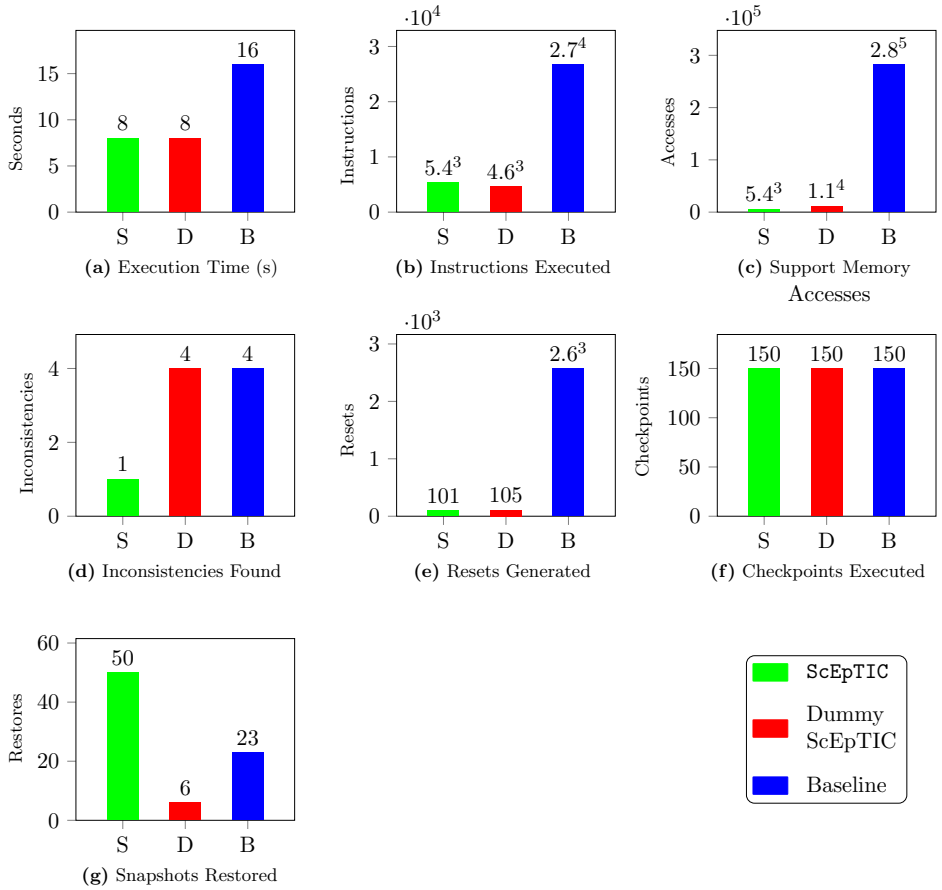

Figure 8.16: Data evaluation results: Static Analysis of Sense benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and global variables allocated in NVM.

the third benchmark setup we analyzed for FFT, which had the global variables allocated into NVM.

As we can see in Figure 8.16a, the time required by the three algorithms for running the analysis is in the same order of magnitude. *Baseline* is the algorithm with the highest execution time, and instead *Dummy ScEpTIC* and `ScEpTIC` takes almost the same amount of time for running the analysis. The higher execution time of *Baseline* is caused by the higher number of resets generated, as we can see in Figure 8.16e. In fact, as we stated for the other benchmarks, resets are used for generating an intermittent execution, and thus the execution time is affected significantly by them. Moreover, *Baseline* verifies the presence of inconsistencies every time it generates a reset. For this

reason, this higher number of resets also causes the higher number of support memory accesses shown in Figure 8.16c, with the consequent overhead. For these reasons, the performance of *Baseline* is lower with respect to the other two algorithms.

Dummy ScEpTIC and `ScEpTIC` have a comparable performance in all the metrics. Figure 8.16g shows the snapshots restored by the three algorithms. `ScEpTIC` restores a higher number of snapshots, but this does not cause a significant performance loss. In fact, as we stated in the analysis of the other benchmarks, restoring a snapshot causes a subsequent sequential execution of a portion of code, that we perform to pass the reset point causing the inconsistency, and thus for continuing the test from a consistent state. We run such portion of code at the maximum possible speed, since it is not interrupted and we do not require any comparison of states. As consequence, the higher number of snapshots restored has an impact on the instruction executed, but it does not influence significantly the execution time. Moreover, the number of instructions executed by the two algorithms are almost the same, as we can see in Figure 8.16b. The higher number of instructions executed by `ScEpTIC` are caused by the higher number of snapshots it restores but, as we previously stated, this does not influence significantly the execution time.

In this benchmark, the support memory accesses are higher in *Dummy ScEpTIC*, as we can see in Figure 8.16c. The overhead introduced by this metric is higher in *Dummy ScEpTIC*. In fact, as we stated in the other benchmarks, it accesses the support memory every time it generates a reset, and thus its access overhead is 104 support memory accesses per reset. Instead, `ScEpTIC` accesses the support memory every while it executes the instructions, leading to an overhead of 1 support memory access per instruction. As consequence, in *Dummy ScEpTIC* verifying the presence of inconsistencies has a higher impact over the performance with respect to `ScEpTIC`.

The execution time of `ScEpTIC` and *Dummy ScEpTIC* is the same, even if the former executes a higher number of instructions. As we previously stated, the higher number of instructions executed by `ScEpTIC` are run in a sequential execution scenario, and thus they do not influence the execution time significantly. Moreover, the higher overhead of *Dummy ScEpTIC* for verifying the state compensates for the higher number of instructions executed by *Dummy ScEpTIC*, making the execution performance of the two algorithms to be almost the same.

Let us now consider Figure 8.16d, which shows the number of inconsistencies returned by the analysis performed by the three algorithms. As we previously stated, `ScEpTIC` is able to identify the actual cause

of an inconsistency, and instead *Baseline* and *Dummy ScEpTIC* returns us pairs of checkpoints and reset which can lead to an inconsistent state. *Baseline* returns the same list of inconsistencies found by *Dummy ScEpTIC*, and the inconsistencies returned by `ScEpTIC` and *Dummy ScEpTIC* are the same of the previous benchmark. In fact, the static analysis we performed is a particular case of the previous dynamic analysis, which finds all the inconsistencies introduced by the allocation of the global variables into NVM, independently of the checkpoint placement. As we stated for the previous benchmark, the result returned by `ScEpTIC` has more value and it does not contain miss-classified information.

For these reasons, we can say that the overall performance of `ScEpTIC` is better with respect to the other two algorithms. From an execution standpoint, it performs better with respect *Baseline*, and has a similar performance with respect *Dummy ScEpTIC*. Instead, from the point of view of the returned information, `ScEpTIC` has a better performance with respect to the other two algorithms.

Finally, as we stated in the previous benchmarks, *Dummy ScEpTIC* is obtained by applying to *Baseline* a set of optimizations taken from the knowledge we developed alongside `ScEpTIC`. Given the increased performance of *Dummy ScEpTIC* over *Baseline*, we can say that the techniques we developed are very effective in analyzing this problem.

3. *Static analysis with stack allocated in NVM and checkpoints placed accordingly to the loop-latch strategy of MementOS [3].*

   Figure 8.17 shows the results of this evaluation, which is similar to the static analysis we performed for the FFT benchmark with the stack allocated into NVM.

   As we can see in Figure 8.17a, the execution time of the three algorithms is within the same order of magnitude. The differences in the execution time are reflected in the resets generated, as we can see in Figure 8.17e. In fact, as we previously stated, resets are used for generating an intermittent execution of the code, with a significant influence over the execution time. *Dummy ScEpTIC* is able to perform such a lower number of resets thanks to the optimizations taken from `ScEpTIC`. Such optimizations are not present in *Baseline*, and thus it generates a higher number of them. Moreover, *Dummy ScEpTIC* performs such a lower number of resets due to its representation of inconsistencies. In fact, as we stated in the other benchmarks, *Dummy ScEpTIC* represents inconsistencies as pairs of checkpoint and reset points causing an inconsistency, and it skips generating resets in points which are already present in such pairs. This is possible since re-generating such resets does not introduce new information in *Dummy ScEpTIC*.

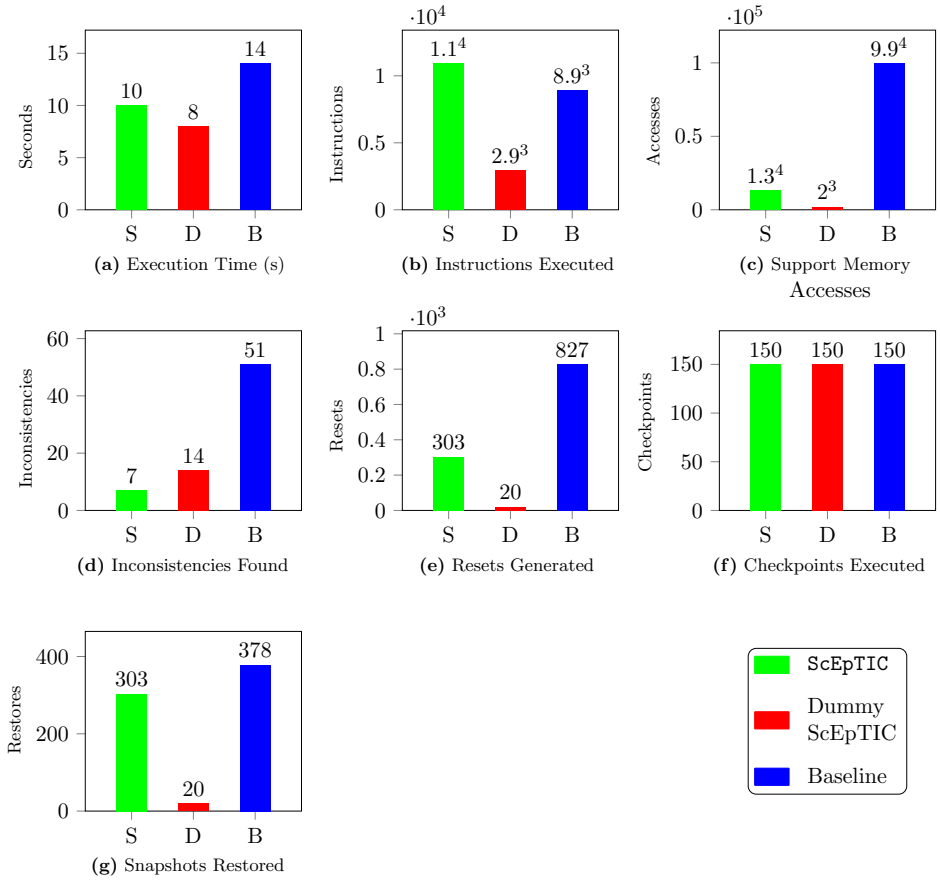

Figure 8.17: Data evaluation results: Static Analysis of Sense benchmark, with checkpoints placed accordingly to the *loop-latch* strategy of MementOS [3] and the stack allocated in NVM.

Instead, `ScEpTIC` has a more detailed representation of inconsistencies, and it is able to find the actual cause of them. For this reason, skipping the generation of a reset would not grant an exhaustive test, as we explained in Section 8.2.1.

Generating more resets also leads to restoring a higher number of snapshots, as we can see in Figure 8.17g. In fact, if the state is inconsistent after a reset, it must be restored for performing an accurate analysis, otherwise we would not be able to properly recognize inconsistencies.

Let us now consider Figure 8.17b, which shows the instructions executed by the three algorithms. `ScEpTIC` executes a higher number of instructions with respect to *Baseline*, even if it restores a lower number of snapshots and generates less resets. This strange fact happens

because *Baseline* verifies the presence of inconsistencies only after it performs a reset. If the state is inconsistent, it restores a snapshots and executes the instructions until it reaches the next reset point. Instead, `ScEpTIC` verifies the presence of inconsistencies while it runs the instructions. When it generates a reset, it restores the latest checkpoint and continues the execution until it reaches the next reset point. When it reaches such instruction, if an inconsistency was found, it restores a snapshots and then it re-executes the instructions until it reaches the point at which it stopped the execution. As consequence, if `ScEpTIC` finds an inconsistency, it re-executes a sequence of instructions three times: the first one for generating the state changes, the second one for verifying the effects of the power reset, and the third one for restoring the snapshot and preparing the analysis of the next reset point. For this reason, `ScEpTIC` executes a higher number of instruction with respect to *Baseline*, even if it does perform a lower number of resets and restores a lower number of snapshots.

The described behavior enables `ScEpTIC` not only to gather the actual cause of inconsistencies, but also to perform a lower number of support memory accesses with respect to *Baseline*, as we can see in Figure 8.17c. Moreover, as we described in the previous benchmarks, the overhead introduced for finding inconsistencies is significantly lower in `ScEpTIC` with respect to the other two algorithms. In fact, `ScEpTIC` has an access ratio to the support memory of 1.18 accesses per instruction executed. Instead, *Dummy ScEpTIC* has an access ratio of 100 accesses per reset generated, and *Baseline* has an access ratio of 120 accesses per reset generated.

Finally, let us consider Figure 8.17d, which shows the inconsistencies found by the three algorithms. The number of inconsistencies found by *Baseline* is significantly higher with respect to `ScEpTIC` and *Dummy ScEpTIC*. As we stated in the previous benchmarks, *Baseline* and *Dummy ScEpTIC* represents inconsistencies in the same way. The lack of optimizations to the reset points in *Baseline* makes it classify a higher number of inconsistencies, but the actual information contained in the result it returns is the same of the one of *Dummy ScEpTIC*. As consequence, we can say that both *Dummy ScEpTIC* and `ScEpTIC` outperform *Baseline* both from an information and execution standpoint. The elements returned by *Dummy ScEpTIC* does not identify the actual cause of the inconsistency. If we analyze such elements alongside with the code of the benchmark, we are able to find such information, and we will end up with the same result returned by `ScEpTIC`. The efforts required for analyzing the result of *Dummy ScEpTIC* does not justify the little performance advantage it has over `ScEpTIC`, which is only 2 seconds. For this reason, we can

say that the overall performance of `ScEpTIC` is better with respect to *Dummy ScEpTIC*, especially thanks to the quality of the information it returns.

The overall result of this three benchmarks confirm the same conclusions we stated for the other evaluations. `ScEpTIC` largely outperforms *Dummy ScEpTIC* and *Baseline* in the *Dynamic Analysis*, thanks to its optimizations. Instead, in the *Static Analysis* it has a comparable performance with the other two algorithms from an execution standpoint. The efforts required for analyzing the results produced by *Dummy ScEpTIC* does not compensate for its little performance advantage over `ScEpTIC`. The concise and precise information `ScEpTIC` returns regards inconsistencies makes its overall performance to be better than the other two algorithms, since we do not require any additional effort for analyzing its result.

Moreover, in all the three benchmark setups *Dummy ScEpTIC* has a better performance with respect to *Baseline*, as it did in all the other benchmarks. As we described in Section 8.2.1, *Dummy ScEpTIC* algorithm consists in the *Baseline* algorithm optimized with a part of the knowledge we developed alongside `ScEpTIC`. Given the performance improvement of *Dummy ScEpTIC* over *Baseline* shown also in these benchmarks, we can also say that the optimizations of `ScEpTIC` are very effective for analyzing this problem.

Conclusion. The final verdict of the data quantitative evaluation is similar with respect to the conclusions of each evaluation benchmarks. When it comes to dynamic analysis, we demonstrated that `ScEpTIC` is significantly faster with respect to both *Baseline* and *Dummy ScEpTIC*. This is achieved thanks to its optimizations and analysis mechanisms, which enables it to verify a lower number of checkpoints, without compromising the exhaustiveness of the test.

Instead, in static analysis `ScEpTIC` execution performance is comparable to the one of *Dummy ScEpTIC* and it is usually faster than *Baseline*. In this case, checkpoints are statically fixed inside the code, and thus all the three algorithms tests the same number of them. The performance boost over the *Baseline* algorithm is achieved thanks to the optimizations to the resets point. Moreover, these optimizations are also the one which enable *Dummy ScEpTIC* to outperform *Baseline*, as we also stated in the benchmarks. *Dummy ScEpTIC* have a slightly better performance than `ScEpTIC`, from an execution point of view. This is achieved thanks to the same optimizations shared with `ScEpTIC`, and due to the way in which *Dummy ScEpTIC* represents inconsistencies. The way in which the algorithms represent inconsistencies defines also how they can analyze the state for finding them. As we stated in Section 8.2.1, *Dummy ScEpTIC* does not find the actual cause of inconsistencies, and instead it finds the possible pairs of checkpoints and resets that can lead to an inconsistent state.

As consequence, it is able to skip already tested reset points without compromising the exhaustiveness of the test it conducts. Instead, `ScEpTIC` can not skip resets already tested, since it could miss some relevant information about inconsistencies. The way in which `ScEpTIC` represents inconsistencies seems to slow it down in static analysis, but the effort required for analyzing the results produced by *Dummy ScEpTIC* or *Baseline* totally vanifies the performance advantage. Moreover, verifying the consistency of the state introduces an overhead caused by support memory accesses, and in our benchmarks results `ScEpTIC` had the lowest overhead for accessing the support memory. For these reasons, and accordingly to the results of the benchmarks, we can establish that `ScEpTIC` overall performance is better than the other two algorithms. In fact, it performs a more accurate analysis and returns more precise information.

Finally, in all the executed benchmarks we saw *Dummy ScEpTIC* significantly outperform *Baseline*. As we stated in page 194, *Dummy ScEpTIC* is obtained by optimizing *Baseline* with some optimizations taken from the knowledge we developed alongside `ScEpTIC`. This performance boost confirm us that the analysis technique we developed in this thesis are very effective against the problem of exhaustively finding data inconsistencies.

8.2.6 Qualitative Evaluation

In this section we compare the analysis of memory inconsistencies `ScEpTIC` performs with the ones we can obtain with *EDB* [13] and *Siren* [17], that are two tools conceived for debugging intermittent executions.

They both expose similar functionalities, and we must manually perform the actions required for analyzing memory inconsistencies, which are:

1. Generating a power reset in precise points during the execution.

   Both *EDB* and *Siren* do not directly provide such functionality, but we can achieve it by placing a *breakpoint* where we want to reset, and then we can use the *reset* command which performs the MCU reset.

   Moreover, we also need to ensure that the code between checkpoints and breakpoints is executed without any unwanted shutdown due to a low energy buffer. *EDB* provides such possibility through energy guards, and instead *Siren* requires us to generate an energy profile that grants no unwanted power reset.

2. Taking and restoring snapshots, to re-establish a consistent state in presence of inconsistencies.

   Both *EDB* and *Siren* do not directly provide such possibility. We can create and restore a snapshot using the exposed functionalities which enables direct accesses to the main memory, but we require a way and a location to store such snapshots.

3. Generating a checkpoint in precise points during the execution.

   This operation is required for running a dynamic analysis, and both *EDB* and *Siren* do not provide such possibility. We can obtain such functionality by forcing the execution to call the checkpoint function. This requires us to pause the execution using a *breakpoint*, manually prepare the stack for such call, and then force the address of the checkpoint function into the program counter.

4. Analyzing the runtime state for verifying the presence of inconsistent results.

   EDB and *Siren* are not conceived for analyzing the presence of memory inconsistencies in a program, and they do not provide any analysis technique for testing the presence of inconsistencies in intermittent executions.

   As we demonstrated in the quantitative evaluation, trying all the possible combinations of checkpoints and resets is not practical. For this reason, we must use the same techniques `ScEpTIC` implements, or a subset of them.

   For verifying the presence of inconsistencies with *EDB* and *Siren*, we must manually access the entire memory. We also require a technique for analyzing the presence of inconsistencies.

   In the quantitative evaluation of the analysis `ScEpTIC` implements, we demonstrated that its performance advantage resides in both the reduction of checkpoints and power resets analyzed, and in the way it analyzes the state for identifying inconsistencies.

   Without performing any significant alteration to the tools, we can adopt the same techniques `ScEpTIC` implements for reducing the number of checkpoints and power resets analyzed, obtaining the same settings of *Dummy ScEpTIC*. Such techniques require pausing the execution at every clock cycle, and verifying the type and targets of the instruction to be executed as next operation.

   If we want to apply the same approach of `ScEpTIC` for analyzing the state, we require performing significant alterations to the tools. In fact, we have to introduce a lookup table and update it whenever an instruction is executed.

The analysis of a program with *EDB* or *Siren* requires us to manually perform a significant number of actions that introduce a considerable time overhead, making the entire process impractical. In fact, we require a breakpoint at any line of code, and we must decide where to perform power resets. Moreover, if we also want to analyze checkpoint placements dynamically, we require to manually execute them.

The large number of manual actions we need to perform also makes the analysis susceptible to human errors, leading to unusable results. Without altering the two tools, we can reach the same performance and result effectiveness of *Dummy ScEpTIC*, which is outperformed by `ScEpTIC`, as we demonstrated in the quantitative evaluation of our tool. For these reasons, the manual analysis using *EDB* or *Siren* is not feasible. Moreover, `ScEpTIC` automatically performs this analysis, without requiring the user to interact with the program during runtime, resulting in a lower effort for obtaining a qualitatively better result.

Automatizing all the manual interactions requires a significant intervention on *EDB* and *Siren*. We need to alter how they execute instructions and implement the analysis logic for performing snapshots, dynamic checkpoints, and power resets. With these alterations, the performance and results we can obtain would be almost similar to the ones of `ScEpTIC`. *EDB* performs the analysis over the target device, resulting in a lower performance with respect to *Siren* and `ScEpTIC`, which emulate the execution of the code. Instead, *Siren* only supports the MSP430 [2] architecture, that is a limitation not present in both *EDB* and `ScEpTIC`.

EDB and *Siren* are very effective for debugging intermittent executions, but they are not conceived for analyzing memory inconsistencies. Adapting these two tools for performing this kind of analysis in practical time is only possible thanks to the testing techniques `ScEpTIC` provides, but the overall result might be limited in terms of supported architectures or performance.

8.3 Input Inconsistencies

8.3.1 Evaluation Baseline

The analysis of input inconsistencies is completely user-dependent, since an input access is considered to be consistent based on the user requirements. The algorithm implemented in `ScEpTIC` for analyzing input inconsistencies has a linear complexity with respect to the total number of executed instructions in a sequential run of the program. Furthermore, for finding input value propagation, `ScEpTIC` exploits a lookup table.

As baseline for both qualitative and quantitative evaluation, we need to find another algorithm for which we can also recreate its behavior in a debugging environment.

Let us focus on Example 8.6, which represents an input accessed with a *saved* access model, and let us try to execute the code as we are in the debugging environment. We set input I_1 to produce a value of 1, and we start executing the entire code until we reach the checkpoint at line 3. Now, for verifying the access model of our input, we perform the checkpoint and continue the execution of the code in two parallel instances. In the first instance we will run the code sequentially, as normal. In the second instance,

```
1 [...]
2 R0 = input(I1)
3 CHECKPOINT
4 ADD R1, R0, 3
5 [...]
6 CHECKPOINT
```

Example 8.6: Access to input I_1 with a *saved* input access model

```
1 [...]
2 CHECKPOINT
3 R0 = input(I1)
4 ADD R1, R0, 3
5 [...]
```

Example 8.7: Access to input I_1 with a *most recent* input access model

```
1 CHECKPOINT
2 R0 = input(I1)
3 BRANCH (R0 != 1), END
4 ADD R1, R0, 3
5 END:
6 [...]
```

Example 8.8: Example of code which cannot be properly analyzed by Algorithm 11.

```
1 [...]
2 R0 = input(I1)
3 CHECKPOINT
4 R1 = input(I2)
5 ADD R2, R1, 3
6 [...]
```

Example 8.9: Example of two inputs.

we change the value of input I_1 to another arbitrarily one (e.g. 3), and then continue the execution. When we reach in both instances the checkpoint at line 6, we stop the execution, and we compare the resulting memory states. In this case there is no discrepancy between the two parallel executions, and so the save access model is granted, because changing of the input value does not modify the produced data. If, instead, a discrepancy is found, the input access model is *most recent*.

Let us now apply the same procedure on Example 8.7, and let us suppose that input I_1 is initialized with the value 1. We reach the checkpoint at line 2, and thus we create two instances of our execution. The first instance continues the execution considering the initialized value of input I_1, and when it reaches instruction at line 4, it sets R_1 to 4. To run the second instance, we first need to change the value of I_1 to an arbitrary value, let us say 3. When the instruction at line 4 is executed, R_1 is set to 6. Now, let us suppose we stop our instances because we reach another checkpoint. We must compare the obtained memory states, and we obtain two different values of R_1, which are 4 and 6. This discrepancy shows a most recent access model.

These examples describe the most simple algorithm we can think of, which is able to find input access inconsistencies. For this reason, we will use it as baseline for the evaluation of `ScEpTIC`, and we have summarized it in Algorithm 11. A limitation of this algorithm is not being able to analyze programs which includes input-dependent data inside branch conditions.

In fact, if we change the input data, we might also change the evaluation of the branch, thus resulting in a different execution flow.

To show this case, let us consider Example 8.8, and suppose that the register R_1 has a value of 4 before the execution of line 4. Furthermore, suppose we impose a *most recent* access model to input I_1. We run the first instance with the value of input I_1 set to 1, and we obtain a value of 4 inside register R_1. For running the second instance, we must change the value of input I_1, with the effect of skipping the branch. In this case R_1 is still set to 4, since it has not changed. When we verify the memory, we expect some discrepancy, since we required a *most recent* access model. Unfortunately, in this case we obtain no discrepancy, and thus the algorithm miss classifies an inconsistency on input I_1.

Another limitation of is Algorithm 11 not being able to analyze multiple inputs at the same time. In fact, since it requires to change an input value for analyzing its access model, the algorithm must re-execute $n + 1$ times the same portion of code, with n equal to the total number of inputs. Let us focus on Example 8.9, which has 2 inputs: I_1 and I_2. To analyze access models, we should run 3 times the code below line 3: the first run does not modify any input value and it is used for comparison, and the other two runs change respectively the value of I_1 and I_2. Then, at the end of the second and third run we are able to compare the memory states to verify the access model of the considered input.

Furthermore, Algorithm 11 does not propagate input usages, and thus it is not able to verify if an input is actually used or not in the portion of code analyzed. This prevents the automatic recognition of inconsistencies for inputs with a *MOST_RECENT* access model, because we can not be sure if such input is used or not. In fact, let us consider the *checkpoint* at line 3 of Example 8.9, and let us suppose we want a *MOST_RECENT* access model over input I_1. For analyzing I_1 the algorithm changes its value and compares the obtained state with the one of the unmodified execution of the same portion of code (i.e., below line 3). Since both states are equivalent, it measures a *SAVED* access model for input I_1, and should display an inconsistency. Unfortunately, input I_1 is not used within the analyzed portion of code, and the algorithm is not able to recognize that for the lack of input usages tracking.

For this reason, the algorithm returns access model of each input within the analyzed portions of code, even if its value is not used in it. It is up to the user analyzing the code for finding where its input data propagates, so to consider only the relevant input access models measured by the algorithm in each function of code. The only inconsistencies automatically found and displayed are the ones of inputs with a *SAVED* access model which behave with a *MOST_RECENT* one, which are the only ones the algorithm is able to verify.

As *baseline* for the quantitative evaluation, we implemented the described algorithm as an extension of the *InterruptManager* module.

Algorithm 11 Alternative input access model analysis algorithm, used as baseline

Require: *access_models*, an array specifying the access model of each input.

```
1:  while program end not reached do
2:      // Save snapshot from which instances will run
3:      base_snapshot ← snapshot of memory and registers
4:      skip checkpoint
5:
6:      // Run until next checkpoint
7:      while current instruction ≠ CHECKPOINT do
8:          run current instruction
9:
10:     // Save snapshot from which memory comparisons will be made
11:     target_snapshot ← snapshot of memory and registers
12:
13:     for each input i ∈ inputs do
14:         restore base_snapshot
15:         change value of input i
16:
17:         // Run until next checkpoint
18:         while current instruction ≠ CHECKPOINT do
19:             run current instruction
20:
21:         // Compare current state with target_snapshot
22:         // for finding access model
23:         if current_state ≠ target_snapshot then
24:             access model of input i in this section of code is most recent
25:
26:             if access_models[i] == saved then
27:                 return input access inconsistency for input i
28:         else
29:             access model of input i in this section of code is saved
30:
31:     // restore snapshot to continue analysis from next checkpoint interval
32:     restore target_snapshot
```

8.3.2 Input Source File

As stated in Section 5.2, input-based inconsistencies can be analyzed only with a static checkpoint mechanism. For this reason, we will consider MementOS [3] as our checkpoint mechanism. Furthermore, since we are not verifying data access inconsistencies, we will not allocate any memory element to the NVM.

The only benchmarks present in the literature which consider inputs are the ones which presents a behavior called *sense*. Such benchmark does not refer to a specific program, but instead it consists in the description of a particular behavior: sensing the environment, processing the sensed values, and finally send the results to the main node of the system. Most of the work done in the TPC field, such as MementOS [3], EKHO [20], and QuickRecall [12], uses such kind of benchmark for the evaluation of their performances.

On the contrary of data inconsistencies, input inconsistencies are not considered from previous works, and thus there are no established benchmarks. To recreate a real-life case scenario, we can use as source file a program accessing inputs and processing their values, thus presenting a behavior similar to *sense*. Furthermore, we must perform our evaluation considering different numbers of inputs, so to verify if the variation of such number produces changes in the performance.

Our *sense* benchmark is shown in Example 8.10, which we will use for the evaluation of the input inconsistency analysis of `ScEpTIC`. It consists in a program which senses values from the environment, performs their sum, and sends it to the main node. Before sending data, it also verifies if the sum exceeds a critical value, and if so it sends an error message to the main node. To tune the number of inputs, input functions will be added in a way which grants an equilibrium in the number of inputs before and after the checkpoint. For example, Example 8.11 will be used for testing 4 inputs.

The considered code does not have a high computational complexity, since we are not interested in evaluating computation performance, and instead we are interested in the analysis one. For this reason, the benchmark program mainly focuses on input accesses. The only computation is represented by the sum of input values, and this is required because it represents a sort of value processing, but more importantly it grants incremental changes in the *sum* variable, making us able to verify the support memory accesses. If no processing was present, the algorithm will still work without any problem, but no input propagation would happen, and thus `ScEpTIC` will not access any support memory for tracking such propagation.

To evaluate the levels of performance in different conditions, we will consider four scenarios, in which the number of input is respectively one, two, three and four. For each scenario we will execute the evaluation considering every possible combination of access models. For example, in the

```
1  int i;
2  int val;
3  int sum = 0;
4  int critical = 10000;
5
6  int main() {
7      // Read data and update sum
8      for(i = 0; i < 50; i++) {
9          val = input();
10         checkpoint();
11         sum = sum + val;
12
13         if(sum > critical) {
14             out('Error: critical value exceeded!');
15         }
16     }
17
18     // Send sum to main node
19     out(sum);
20 }
```

Example 8.10: Program used for evaluating `ScEpTIC` input analysis algorithm, inspired by real-world scenarios and the *sense* benchmark of MementOS [3].

scenario with two inputs we will run the program four times: one time with *MOST_RECENT* for both inputs, one time with *SAVED* for both inputs, and two times by alternating *MOST_RECENT* and *SAVED* for the two inputs. In this way, we can not bias our evaluation by focusing only on one case, and we are able to consider all the possible combination of inconsistencies over input accesses.

Finally, the memory configuration is not important for this analysis since we are not testing data inconsistencies. The algorithm implemented by `ScEpTIC` does not perform any reset, and thus no data inconsistency can interfere during the analysis of input access inconsistencies. Instead, the algorithm used as baseline is not able to analyze input access inconsistencies if the state is inconsistent. In fact, it changes input values to verify the access model of each input, and thus if a data inconsistency is present, it might classify an input access inconsistent due to the presence of a data inconsistency. Since we are not interested in evaluating `ScEpTIC` with respect to the worst case scenario of the baseline algorithm, we will use as memory configuration the same one of MementOS [3], which does not allocate any memory element into NVM, except for the checkpoint.

```
1  int i;
2  int val1, val2, val3, val4;
3  int sum = 0;
4  int critical = 10000;
5
6  int main() {
7      // Read data and update sum
8      for(i = 0; i < 50; i++) {
9          val1 = input1();
10         val2 = input2()
11         checkpoint();
12         val3 = input3();
13         val4 = input4();
14         sum = sum + val1 + val2 + val3 + val4;
15
16         if(sum > critical) {
17             out('Error: critical value exceeded!');
18         }
19     }
20
21     // Send sum to main node
22     out(sum);
23 }
```

Example 8.11: Version of Example 8.10 with 4 input elements.

8.3.3 Quantitative Evaluation Results

We run the previously described *sense* benchmark multiple times, configuring it with one, two, three, and four inputs. For each one of these four configurations, we evaluated every possible combination of input access models. For example, for the configuration with one input, we run the benchmark two times: one with a *most recent* access model set for the input, and one with a *saved* one. As result, we run the benchmark 30 times: 2 times with one input, 4 times with two inputs, 8 times with thee inputs, and 16 times with 4 inputs. The different execution of each input configuration gave the same results from all the metrics, except for the inconsistencies found. In fact, both *Baseline* and `ScEpTIC` do not vary the execution depending on the input access models selected.

Figure 8.18 shows the metrics relative to the execution of the four input configurations, and Figure 8.19 show the inconsistencies found by each run of the four input configurations.

Execution. Firstly, let us consider the performance from an execution point on view. Figure 8.18a shows the amount of time required by the two algorithms for running the analysis. As we can see, `ScEpTIC` is faster with respect to *Baseline* in every benchmark configuration. The higher execution time of *Baseline* is caused by the higher number of instructions it executes

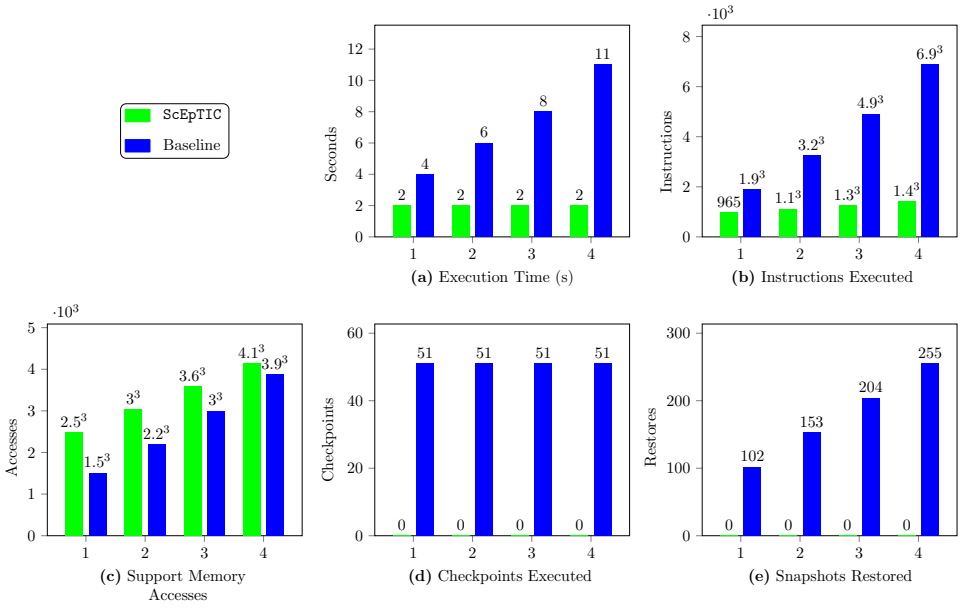

Figure 8.18: Input evaluation results: metrics of the input access analysis with one, two, three, and four different inputs.

for running the analysis, as we can see in Figure 8.18b. In fact, `ScEpTIC` performs a sequential execution for running the analysis, and instead *Baseline* generates multiple intermittent executions. As consequence, `ScEpTIC` executes a significant lower number of instructions with respect to *Baseline* in all the four different configurations of the benchmark, and thus it requires less time for completing the analysis.

Moreover, as we can see in Figure 8.18b, when the number of inputs present in the program increases, the instructions executed by the two algorithms also increase. `ScEpTIC` performs a sequential execution of the code, and this increase is a consequence of the introduction of new instructions. Instead, *Baseline* performs a number of intermittent executions which is dependent on the number of inputs. As consequence, increasing the number of inputs makes *Baseline* to generate a higher number of intermittent executions, which leads to a substantial increase of the instructions executed.

Figure 8.18d shows the number of checkpoints executed by each algorithm in the four benchmark configurations. `ScEpTIC` performs this analysis as a sequential execution, and thus it does not execute any checkpoint. Instead, *Baseline* runs this analysis as an intermittent execution, and thus it performs a checkpoint every time it is encountered. Moreover, we can also notice that *Baseline* performs the same number of checkpoints, independently of the number of inputs present in the code. The reason of this behavior is that checkpoints are statically fixed inside the code, and thus their execution does not depend on the input configuration.

As consequence, adding more inputs does not change the number of checkpoints executed during the analysis.

Let us now consider Figure 8.18e, which shows the number of snapshots restored during the analysis. As we can see, `ScEpTIC` does not restore any snapshot. In fact, it performs a sequential execution, and analyzes the correctness of the required access models by propagating the information regarding input accesses across the support memory. Instead, *Baseline* produces different intermittent executions, and tests the access models by changing the value returned by an input so to measure the changes in the state. As consequence, it restores multiple snapshots for analyzing input access models. In fact, as we stated in Section 8.3.1, every time it has to verify the access model of an input, it firstly run the code until it reaches a checkpoint, and then it saves the runtime state into the support memory. As next operation, it restores the snapshot taken when the previous checkpoint was executed, and sets a new value for an input. Finally, it executes the program until it reaches the next checkpoint, and compares the runtime state with the one saved in the support memory, so to find the access model of the input. For this reason, a higher number of inputs results in a higher number of snapshots restored, as we can see in Figure 8.18e.

As we previously stated, the higher number of instructions executed by *Baseline* is a consequence of the intermittent execution it performs, which is characterized by the execution of checkpoints, generation of resets, and restoration of snapshots. These operations are the ones used for creating an intermittent execution, and thus are the ones causing *Baseline* to have such higher number of instructions executed with respect to `ScEpTIC`. Moreover, we did not report the graph representing the power resets generated, since both the two algorithms did not generate any reset. In fact, `ScEpTIC` performs a sequential execution, and thus it does not perform any power reset. Instead, *Baseline* directly restores a snapshot for generating intermittent executions, since it is interested in the generation of multiple intermittent executions which starts from a consistent state. Generating a power resets might introduce *Data Inconsistencies*, as we shown in Chapter 4, which would lead to an inaccurate analysis. In fact, in such a scenario, *Baseline* would not be able to understand if the difference in the state was caused by a data inconsistency or the changed input value.

Let us now consider Figure 8.18c, which shows the support memory accesses performed by the two algorithms. As we can see, `ScEpTIC` perform a higher number of support memory accesses with respect to *Baseline*. We can also see that increasing the inputs of the program, decreases the gap between the support memory accesses of the two algorithms. As we previously explained, the sequential execution of `ScEpTIC` permits it to run a lower number of instructions, and to both avoid the execution of checkpoints and restoring snapshots. All these operations influences the execution time significantly, and instead support memory accesses seems to not affect it.

Algorithm	One Input	Two Inputs	Three Inputs	Four Inputs
`ScEpTIC` (accesses/instruction)	2.59	2.73	2.77	2.93
Baseline (accesses/snapshot)	14.71	14.38	14.71	15.29

Table 8.2: Access ratio to the support memory of `ScEpTIC` and *Baseline*.

In fact, the support memory accesses `ScEpTIC` performs with 4 inputs are 64% higher with respect to the ones it performs with 1 input, but the execution time is not affected.

Moreover, for understanding the relation of this metric with the others, we must consider when and how the support memory is accesses, as we did for the evaluation of data inconsistencies.

`ScEpTIC` uses the support memory for keeping track of input-dependent elements in the runtime state. During the execution of the program, whenever `ScEpTIC` reads the value of an element such as a register or a memory cell, it also verifies if such value is input-dependent, and thus it is able to measure the access model. Instead, *Baseline* saves into the support memory a snapshot of the runtime state. Then, it executes the multiple times the same portion of code, each time modifying the value returned by an input access. It uses the support memory for comparing the runtime state present at the end of these multiple executions, so to verify if a variation to the input value changes the state. For these reasons, we can say that `ScEpTIC` accesses the support memory every time it executes an instruction, and instead *Baseline* accesses the support memory every time it generates a snapshot. Table 8.2 shows the support memory access ratios of the two algorithms in the different configurations of the benchmark. As we can see, `ScEpTIC` has a significantly lower access ratio with respect to *Baseline*, and thus the overhead introduced by accessing the support memory is higher in *Baseline*. Moreover, we can also see that increasing the number of inputs in the program produces a higher access ratio in both the two algorithms, but *Baseline* is subjected to a higher increase with respect to `ScEpTIC`.

In Table 8.2 we can observe that when we increase the number of inputs from one to two, we see a decreased access ratio in *Baseline*. For understanding the reason of this strange behavior, let us focus on how *Baseline* works. It analyzes the program considering multiple checkpoint intervals, which consist in sequences of instructions such that the first and last ones are checkpoints. *Baseline* runs each checkpoint interval $n + 1$ times, with n equal to the number of inputs. The first run is used for observing the runtime state achieved without modifying the value returned by the input, which is saved into the support memory. The other runs are used for verifying if the state changes when the value returned by each input is modified,

and for doing so it compares the runtime state with the snapshot saved into the support memory. Between each run, a snapshot representing the initial runtime state of the checkpoint interval is restored. The number of support memory writes does not depend on the inputs, and instead depends on the number of checkpoint intervals tested, which are measured as checkpoints executed in Figure 8.18d. As consequence, if we increase the number of inputs in our program, the support memory writes remain unchanged. Instead, we increase the number of support memory reads and snapshots restored, since we introduce a new execution for each checkpoint interval. For each checkpoint interval, the overall number of support memory accesses should be $n + 1$ times the dimension of the saved runtime state, since we perform 1 write and n reads of the same memory elements. Instead, when *Baseline* compares the current runtime state with the one saved into the support memory, it stops as soon as it finds a discrepancy, and thus it performs a lower number of support memory reads with respect to the ones we estimated. This is the reason of the decreased access ratio in *Baseline* when we increase the number of inputs to two: the overall number of support memory accesses introduces are not proportional to the increase of snapshots restored. In fact, the support memory writes are almost not incremented, since the variation to the number of elements in the runtime state is only of one memory cell and one register. Instead, the support memory reads introduced increase more than 95%, but they are not doubled. As consequence, the overall increase of support memory accesses is 46%, and the snapshots restored increments by 50%. As consequence, we see a decreasing access ratio. Ideally, the access ratio of *Baseline* should be constant, since the support memory accesses and the snapshots restored should both increment by $1/n$. The way in which *Baseline* performs state comparisons makes it able to increment the support memory accesses by less than $1/n$, and thus we see a decrease of the access ratio in some cases.

The support memory accesses of `ScEpTIC` are higher with respect to *Baseline*, but `ScEpTIC` has a lower overhead for performing support memory accesses, as we saw in the access ratios. In fact, *Baseline* accesses the support memory every time it restores a snapshot, and instead `ScEpTIC` accesses it during the execution of the instructions.

From an execution standpoint, we demonstrated that the overall performance of `ScEpTIC` is significantly better than the one of *Baseline*. The more input we have in the program, the better `ScEpTIC` performs when it is compared to *Baseline*. We could argue that the access ratio of `ScEpTIC` increases alongside the inputs present in the program, and instead *Baseline* has the tendency of a constant access ratio with respect to the increase of inputs. Clearly, with a significantly high number of inputs, the access ratio of `ScEpTIC` will be higher with respect to the one of *Baseline*. In such case, the execution time of `ScEpTIC` will be exponentially lower with respect to *Baseline*, since the former performs a sequential execution, and instead the

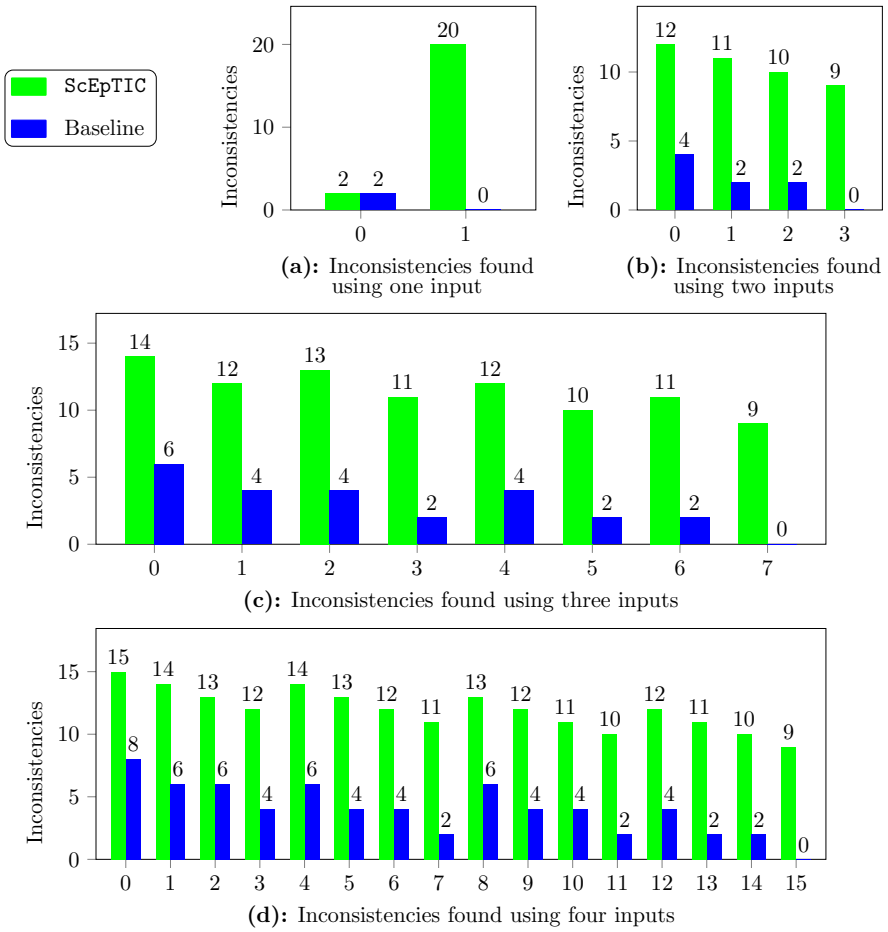

Figure 8.19: Input evaluation results: inconsistencies found in the four configurations of our benchmark. Each configuration was run once for every possible combination of input access methods of its inputs.

latter will perform multiple intermittent executions. For this reason, the advantage on the support memory access ratio of *Baseline* would not grant a performance advantage over `ScEpTIC`.

Returned information. Let us now focus on the quality of the information returned by the two algorithms about input access inconsistencies.

Let us consider Figure 8.19, which shows the inconsistencies found by the algorithms in the different configurations of this benchmark. As we previously stated, we run each configuration multiple times, one for every possible combination of the inputs access models.

We represented input access models as binary values with length equal to the number of inputs in the program. The $i-th$ bit represent the access

model of the $i-th$ input. We associated a 0 to a *SAVED* access model, and a 1 to a *MOST_RECENT*. We encoded this binary values as integers, and we placed them under the corresponding group in the x axis of the graphs. For example, 1 corresponds to 01, and indicates that we run the test with a *SAVED* access model for the first input, and with a *MOST_RECENT* one for the second input.

Figure 8.19a shows the number of inconsistencies found by the two algorithms during the analysis of the benchmark configured with one input. We see two different groups of results: in the first one we configured a *SAVED* access model for the input, and in the second one we configured a *MOST_RECENT* access model. In the first analysis, the two algorithms return the same number of inconsistencies. Instead, in the second analysis, `ScEpTIC` identifies 20 inconsistent input accesses, and *Baseline* is not able to identify any inconsistency since it can only identify inconsistencies over inputs configured with a *SAVED* access model, as we stated in Section 8.3.1. To overcome this problem, *Baseline* returns also a list containing the access model measured for each input in every tested checkpoint interval, but the actual recognition of inconsistencies is leaved to us.

Let us now consider Figure 8.19b, which shows the number of inconsistencies found by the two algorithms during the analysis of the benchmark configured with two inputs. The first group refers to the analysis configured with a *SAVED* access model for both the two inputs. `ScEpTIC` finds 12 inconsistencies, and instead *Baseline* finds 4 of them. This difference does not imply that *Baseline* is not able to find all the inconsistencies. In fact, with a deeper inspection of the results returned by both the two algorithms, we can state that they are able to identify all the inconsistencies, and the different number found is a consequence of the different representation of inconsistencies in the results of the two algorithms. `ScEpTIC` tracks the propagation of input-dependent values across the memory elements, and thus it is able to provide us information about the instructions accessing an input with an access model different with respect to the one we specified. As consequence, it returns us an inconsistency for every instruction accessing an input inconsistently. Instead, *Baseline* is only able to identify the access model of an input, but not the instructions accessing its value. As consequence, it returns a list of checkpoint intervals in which the input is accesses inconsistently. For this reason, we can say that both the two algorithms are able to correctly identify the inputs which are accessed inconsistently, but `ScEpTIC` is able to provide us a more detailed information about where such accesses happens.

If we consider the other three groups of Figure 8.19b, we can see also in this case that *Baseline* finds a lower number of inconsistencies with respect to `ScEpTIC`. In this three cases the reason of such lower number of inconsistencies recognized by *Baseline* resides also in the fact that it is not able to identify if an input with a *MOST_RECENT* access models is accessed

inconsistently. *Baseline* provides us the list of the measured access models, but we have to analyze it alongside the code for verifying if inputs are accessed inconsistently. Moreover, in the fourth configuration group, *Baseline* is not able to identify any inconsistency, since the two inputs are configures both with a *MOST_RECENT* access model.

We can find the same condition we described in both Figure 8.19c and Figure 8.19d, which shows the inconsistencies found with three and four inputs. The first group is the only one in which *Baseline* is able to identify all the inconsistencies, since all the inputs are configured with a *SAVED* access model. Instead, in the latest groups of the two benchmarks, *Baseline* is not able to automatically identify any inconsistency, since all the inputs are configured with a *MOST_RECENT* access model. In the other groups, *Baseline* is only able to identify the inconsistencies regarding the inputs configured with a *SAVED* access mode. For verifying the inconsistencies of the inputs configures with a *MOST_RECENT* access model, we must rely on the returned list which provides the measured access models of each input for every checkpoint interval.

The information returned by `ScEpTIC` is more complete with respect to the one of *Baseline*, and it is always able to provide us the actual list of inconsistencies. Instead, *Baseline* is not able to identify if inputs with a *MOST_RECENT* access model are accessed inconsistently, and thus we have to analyze the code alongside with the list containing the measured access models. Moreover, *Baseline* does not provide us any information about the instructions using an input inconsistently, and thus it is harder to solve such inconsistency.

Considering our analysis of the results that Figure 8.18 and Figure 8.19 show, we can say that `ScEpTIC` performs better than *Baseline*, not only from the execution standpoint, but also in the quality of the information returned. The key of its better performance resides in the tracking of the propagation of input values, which is not performed by *Baseline*. In fact, such tracking permits `ScEpTIC` both to perform a sequential execution and to identify the instructions using input-dependent values. As consequence, it is able to gather a more precise information about inconsistencies in less time with respect to *Baseline*.

8.3.4 Qualitative Evaluation

In this section we compare the analysis of input access inconsistencies `ScEpTIC` performs with the ones we can obtain with *EDB* [13] and *Siren* [17], that are two tools conceived for debugging intermittent executions. They do not address *input inconsistencies*, and thus we require adopting a technique for analyzing the presence of this inconsistency type.

We can choose to use Algorithm 7, that is the one `ScEpTIC` implements for analyzing input access inconsistencies. This technique does not require

the analysis of intermittent executions, and it simply runs the program sequentially. For implementing such mechanism, we require a lookup table that keeps track of input accesses, and we must propagate this information every time an instruction is executed.

EDB directly runs on the device we want to test, and thus we are not able to modify how instructions are executed. As consequence, we must execute interactively the program, so to manually propagate the elements of the lookup table. This process is not practical, since it requires us to perform a significant number of actions.

Instead, *Siren* emulates the execution of the program, and thus we are able to extend its code for keeping track of input value propagation. The alteration required for such implementation are significant, since we must propagate input values every time both registers and memory cells are accessed. The performance and result effectiveness we can obtain with *Siren* will be comparable to the ones of `ScEpTIC`, especially thanks to the analysis technique the latter provides.

Both *EDB* and *Siren* are not conceived to track the propagation of input values, and the efforts required for implementing or performing such operation are significant. Another approach we can follow for analyzing input access inconsistencies is to use Algorithm 11, that is the one we used as baseline for our quantitative evaluation. It equires us to analyze various intermittent executions of the code, and we are not required to modify the tools.

For running this kind of analysis, we must perform the following actions:

1. Generating a power reset in precise points during the execution.
   Both *EDB* and *Siren* do not directly provide such functionality, but we can achieve it by placing a *breakpoint* where we want to reset, and then we can use the *reset* command which performs the cpu reset.
   Moreover, we also need to ensure that the code between checkpoints and breakpoints is executed without any unwanted shutdown due to a low energy buffer. *EDB* provides such possibility through energy guards, and instead *Siren* requires us to generate an energy profile that grants no unwanted power reset.
2. Taking and restoring snapshots, for analyzing the obtained access model and for restoring a consistent runtime state.
   Both *EDB* and *Siren* do not directly provide such possibility. We can create and restore a snapshot using the exposed functionalities which enables direct accesses to the main memory, but we require a way and a location to store such snapshots.
3. Reproducing specific input values, for analyzing their effects over the execution.
   EDB is a hardware-based solution, and thus we require reproducing a physical event for generating a specific value of an input element.

Instead, *Siren* is a software-based solution, and it makes simpler generating a specific value for input elements such as sensors.

4. Analyzing the runtime state for verifying the access model of input elements.
   For verifying the access model, we must compare the snapshot with the current memory state. Both *EDB* and *Siren* expose functionalities for manually accessing and reading the content of single memory cells.

Manually performing this analysis permits us to overcome the problem of not being able of recognizing all the input inconsistencies, since we analyze their presence while we run the program. With this approach, we require putting *breakpoints* before and after each checkpoint, since we require restoring snapshots and producing power resets in such positions. The number of power resets we need to generate depends on the number of input elements present in the program, as we demonstrated in the quantitative evaluation of our tool. The number of actions we need to perform introduces both a considerable time overhead and the possibility of human errors, making the entire process impractical, especially with a high number of inputs.

Automatizing this approach requires us to perform a significant alteration to the tools, and would lead us to reduce the effectiveness of the analysis. In fact, we would not be able to recognize automatically all the inputs inconsistencies, as we demonstrated in the quantitative evaluation. For this reason, the automation of this analysis techniques does not introduce any benefit, and we must discard this option. Moreover, *EDB* still requires the user to manually produce different input values, and thus we are not entirely able to automatize this analysis using such tool.

With a low number of inputs, we can use this approach for identifying input inconsistencies using *EDB* and *Siren*, obtaining a result effectiveness similar to the one of `ScEpTIC`.

We can exploit *EDB* and *Siren* for analyzing input access inconsistencies, but they are not conceived for this kind of analysis. Independently of the technique we consider, we can not automatize the analysis with *EDB*, and thus we are not able to reach the same performance of `ScEpTIC`. Instead, *Siren* can be extended to support the analysis performed by `ScEpTIC`, but the analysis will be limited to the MSP430 [2] architecture.

8.4 Intermittent Execution Analysis

8.4.1 Evaluation Setup

As we described in Section 5.3, *Intermittence-based inputs* are user-dependent, since their definition is subjective and depends on the application requirements. Testing their correctness consists in debugging the effects of a defined set of power resets, and thus we do not require the generation of all

the possible intermittent execution combinations. Moreover, as we stated in Section 5.4, *Output Inconsistencies* can be tested in the same way we test *intermittence-based inputs*. In fact, we only have to generate a power reset after the execution of an output routine, so to measure the effects of its possible re-execution over the environment.

The analysis performed by `ScEpTIC` for *Data Inconsistencies* and *Input Inconsistencies* optimizes both where checkpoints need to be tested and where resets needs to be generated. Instead, for analyzing both *intermittence-based inputs* and *output inconsistencies*, the algorithm implemented by `ScEpTIC` does not optimize the resets to be generated, since they are user-defined. Moreover, as we stated in Section 5.3, this kind of analysis requires a static checkpoint mechanism, and thus we must test all the checkpoints. The actual optimization `ScEpTIC` aims to provide is the one regarding the actions required to the user for debugging intermittent executions.

In fact, once the user configure the reset points, the checkpoints to be analyzed, and the variables to be tracked, `ScEpTIC` produces and analyzes the resulting intermittent executions.

For these reasons, we will evaluate this kind of analysis qualitatively, and we will especially focus on the user intervention. In fact, profiling operation like the one we described does not require a particular algorithm, and can be easily performed in the available debugging environment, like EDB [13] and Siren [17]. The real difference between `ScEpTIC` and the other debugging environments consists in the user-dependent actions required for analyzing an inconsistency-based execution flow. For example, in `ScEpTIC` the user is required to modify the source code by inserting where resets should happen, and then the entire analysis is automatically performed. Instead, in other debugging environments the user is required to put breakpoints for both performing a reset and analyzing the state. These actions are comparable, and thus we are interested in evaluating the user experience with the different approaches available in `ScEpTIC`, EDB [13], and Siren [17].

As input for our evaluation, we will consider only a static checkpoint mechanism since, as described in Section 5.3, this analysis can be only performed in such scenario. Moreover, for the purpose of this evaluation we will not differentiate between inputs and outputs routines, since the actions taken by `ScEpTIC` and the debugging environments consist in tracking the executed function, independently of their type. For this reason, we will threat them as general I/O routines.

Unfortunately, no previous work considers intermittence as input of the program, nor the possibility of output inconsistencies, and thus no benchmark exists in the literature for this scenario. To perform our qualitative evaluation, we need to base our reasoning over a source file representing a real-world scenario for this domain, like the *sense* benchmark previously described. We will not perform the execution of the benchmark in `ScEpTIC`, but instead we will describe and compare the required actions the user needs

```
1  int i;
2  int restarts; // in NVM
3  float orientation;
4
5  int main() {
6      [...]
7      for(i = 0; i < 10; i++) {
8          restarts = 0;
9          checkpoint();
10         executions = executions + 1;
11
12         if(executions > 8) {
13             send('Recalibration failed');
14             send(0);
15         }
16         else if(executions > 3) {
17             orientation = get_servo() + 0.2;
18             send('Recalibrating unstable source');
19             set_servo(orientation);
20         }
21         else if(executions > 1) {
22             send('code re-executed');
23         }
24         [...]
25         checkpoint();
26     }
27     [...]
28 }
```

Example 8.12: Benchmarks used for the evaluation of intermittence-based inputs.

to perform for testing the execution flow.

Furthermore, to test this scenario, we require these conditions:

1. Inconsistencies over one or more variables, which may influence the execution flow.
2. A bunch of I/O function which are executed depending on the value of inconsistent variables.

We will base our reasoning on Example 8.12, which represents a real case scenario of the TPC domain. Before analyzing the code, we must consider that we are not interested in computational intensive tasks for the evaluation of this analysis, because we do not need to focus on computational power or on checkpoint placements, and instead we are interested in the user perspective. Moreover, we will use the source file only for reasoning purposes, so that the actions taken by the user are more clear.

Our benchmark consists in a simple calibration mechanism for an energy source. Let us suppose that we have a solar panel attached to a servo

which controls its orientation, and that we can move such servo through the output function *set_servo()*. The input function *get_servo()* returns the current orientation of our servo, and the output function *send()* is used for communications purposes. Our intermittence-based input is the variable *executions*, which tracks the number of times the code between lines 10 and 24 is executed. Thanks to this input, we are able to perform different actions accordingly to the number of re-executions.

8.4.2 Qualitative Evaluation

In this section we compare `ScEpTIC` with *EDB* [13] and *Siren* [17], that are two tools conceived for debugging intermittent executions.

For analyzing environment interactions and the behavior of a program running in an intermittent execution scenario, we require:

1. Tracking environment interactions, so to analyze their execution.
   - `ScEpTIC` abstracts the elements capable of environment interactions, such as sensors and actuators, and threat them as program functions. It automatically tracks the execution of input and output elements, and their values.
   - *EDB* has a direct connection to the bus managing I/O peripherals, and it supports monitoring I/O events.
   - *Siren* requires emulating each external components, and it already provides an implementation of the most commonly used ones. It permits logging events happening on I/O ports.
2. Reproducing specific input values, for analyzing their effects over the execution. *EDB* is a hardware-based solution, and thus we require reproducing a physical event for generating a specific value of an input element. Instead, both *Siren* and `ScEpTIC` are software-based solutions, and they make simpler generating a specific value for input elements such as sensors.
3. Retrieving the value of variables, and signaling the execution of certain code regions.
   - `ScEpTIC` provides a $printf()$ function that can be used inside the program for printing the value of any variable. It also provides a $profiling\_log()$ function that can be used for tracking events and variable values.
   - *EDB* provides both a $printf()$ function and commands for directly accessing the memory, that can be used for retrieving variable values. Instead, for tracking events it provides *watchpoints*.
   - *Siren* provides a $siren\_command()$ function that can be used for both printing the value of variables and for tracking events. It also provides commands for directly accessing the memory, and has a built-in support for tracking the executed code and for monitoring events.

4. Generating the combination of intermittent executions that reproduce the conditions we want to test. This can be achieved either by generating a shutdown or performing a reset arbitrarily during the execution.
    - `ScEpTIC` permits us to generate a set of precise intermittent executions by placing *profiling_start*() and *profiling_reset*() functions directly inside the code. It automatically handles the generation of shutdowns and the re-execution of the code.
    - *EDB* does not provide a direct way for generating a power reset in a precise point. For achieving such functionality, we must set a *breakpoint* where we want to reset, and then we can use the *reset* command which performs the MCU reset. We also need to use an energy guard or a constant power source to ensure that the code between checkpoints and breakpoints is executed without any shutdown due to a low energy buffer.
    - *Siren* does not provide a direct way for generating a power reset, and requires us to perform the same actions we described for *EDB*. *Siren* does not have energy guards, and for ensuring that power resets happens only where we require, we must generate an energy profile that grants such behavior.

As we can see, for this analysis the main difference between these tools resides in how we generate a specific intermittent execution.

Let us now focus on how we can analyze the behavior of Example 8.12, which exploits the intermittence-based variable *restarts*. We can use the guidelines we provide in Section 5.3.2.

A simple way to test the behavior of this code, is to generate 7 power resets after the execution of line 24, since the code block executed changes after the first, third, and eight resets. Then, we track the value of *executions* variable, and the execution of *send* and *set_servo* I/O functions. The result of this analysis should tell us which portion of code is executed after each power reset, and the I/O events that are generated within such execution.

The action we require performing with `ScEpTIC` for testing these combinations of intermittent executions are:

1. We create the configuration for *send*(), *set_servo*(), and *get_servo*() I/O functions.
2. We place in our code a *profiling_start*(7); before line 8.
3. We place in our code a *profiling_reset*(−1); after line 24.
4. We place in our code a *profiling_log*(*executions*); after line 10, so to track the value of the *executions* variable alongside I/O events.
5. We run the analysis.

Performing the same analysis with *EDB* requires us to:

1. Attach the hardware component of *EDB* to the board we want to analyze.
2. Attach to the board the servo and the hardware component enabling the communication through the *send*() function.

3. Place a $printf("\%d", executions)$ after line 10.
4. Set a *code breakpoint* at line 24.
5. Set an *energy guard* between lines 8 and 24, so to ensure no unwanted power reset happens.
6. Activate the logging for I/O events.
7. Start the execution, and when we reach the *code breakpoint*, we execute the *reset* command. In this way, the MCU resets its volatile state and restarts the execution. We require repeating this step for each power reset we want to generate.

Finally, executing this same analysis with *Siren* requires us to:

1. Create the module implementing the servo and the communication system.
2. Place a $siren\_command("PRINTF", executions)$ after line 10.
3. Set a *breakpoint* at line 24.
4. Generate an energy profile that ensures no unwanted power reset happens.
5. Activate the logging for I/O events.
6. Start the execution, and when we reach the *breakpoint*, we execute the *reset* command. In this way, the MCU resets its volatile state and restarts the execution. We require repeating this step for each power reset we want to generate.

For analyzing a defined set of intermittent executions we do not need to modify any tool, since they are all conceived for this kind of analysis.

EDB and *Siren* have a similar workflow for performing this analysis. The only difference is that *EDB* is a hardware-based solution that requires a direct connection with the device we want to test, and the MCU runs the code we want to analyze. Instead, *Siren* has the advantage of being a software-based solution and thus does not require such user intervention. Both *EDB* and *Siren* require us to generate intermittent executions during runtime. Instead, `ScEpTIC` permits us to define at compile time the set of intermittent executions to be analyzed. In our example, we generated all the power reset in the same point, and thus interacting with the program during runtime for producing intermittent execution is trivial. If we want to test power resets in different points, we either have to remove a breakpoint once we reach it and insert a new one, or place multiple breakpoints and choose the correct one for executing the *reset* command. The higher is the number of different power rests, the higher gets the number of breakpoints we must place. Choosing the correct position of a power reset in the presence of a high number of breakpoints can be confusing, and we can make an error that causes the analysis to produce wrong results. For this reason, we can state that `ScEpTIC` has an easier setup than the other two tools, and permits us to generate a high number of power resets with a lower effort than *EDB* and *Siren*.

Let us now consider the results the three tools return. *EDB* and *Siren* both provide us the tracked information during runtime, and thus we analyze the behavior of the program during its execution. With a high number of breakpoints it can be difficult and confusing analyzing their results, for the same reasons we previously described. `ScEpTIC` provides the result at the end of the analysis, as we already described in Section 7.3. For each intermittent execution it generates, it provides the list of encountered I/O functions and the value of the logged variables. As consequence, the analysis of the result `ScEpTIC` returns is easier than the one of *EDB* and *Siren*, especially with a high number of intermittent executions generated.

Both *EDB* and *Siren* are conceived for debugging intermittent executions, and they are effective in doing so. They have more debugging capabilities than `ScEpTIC`, but for this kind of analysis we only require tracking the execution, logging variables and generating power resets. Even if the result returned by the three tools are comparable, in our opinion `ScEpTIC` makes the analysis of multiple intermittent executions easier.

Chapter 9

Conclusion and Future Works

Devices powered with the energy harvested from the environment have lower maintenance costs, and enables applications and services considered to be impractical due to battery limitations. They experience an *intermittent execution* characterized by frequent shutdowns that may lead to an unpredictable behavior. Different techniques exists for preserving the device state across shutdowns, but very little work addresses the problems of testing and analyzing the effects that an intermittent execution produces. In this thesis we addressed such problems, and our contribution consists in two different parts: an analysis of the effects that an intermittent execution causes, and `ScEpTIC`, an intermittence-based debugging tool.

In Chapter 4 we studied and analyzed the effects that an intermittent execution causes in the device memory. We found three different taxonomies of unwanted behaviors, that we referred as *inconsistencies*: **Data Access Inconsistencies**, **Activation Record Inconsistencies**, and **Memory Map Inconsistencies**. For each taxonomy, we analyzed its causes from the machine-code level, and we provided guidelines on how we can remove such unwanted behaviors. Moreover, in Section 4.5 we provided a set of techniques for finding the presence of these kinds of inconsistencies in a program, and for analyzing the effects that they may cause over the device state and behavior.

In Chapter 5 we studied and analyzed the effects that an intermittent execution causes to environment interactions, and we discussed the possibility of using intermittence as input for our programs. In the first part of this chapter, we analyzed how an intermittent execution affects both *input* and *output* interactions with the environment. We found and analyzed two different taxonomies of unwanted behaviors: **Input Access Inconsistencies** and **Ouput Inconsistencies**. For each taxonomy, we analyzed its causes from the machine-code level, and we provided guidelines on how we can

remove such unwanted behaviors. Moreover, we also provided a set of techniques for verifying and analyzing their presence. In the second part of this chapter, we analyzed the possibility of exploiting *Memory Inconsistencies* as **Intermittence-Based Inputs** for our programs, and we discussed the possibilities that this new kind of input opens. We examined real-case scenario examples, and we provided a technique for analyzing the correctness and effects of *intermittence-based inputs*.

In Chapter 6 and Chapter 7 we presented `ScEpTIC`, an intermittence-oriented debugging tool that we developed. We implemented in it the techniques we provided in the first part of this thesis. Inside these chapters, we described the main architectural components of `ScEpTIC`, and we described how we can use it for analyzing the correctness of a program in presence of an intermittent power source. `ScEpTIC` was able to analyze in practical time the effects that all the possible combinations of intermittent executions produce.

We evaluated the performance of the different techniques we implemented in `ScEpTIC`, comparing them with analysis approaches that anyone can think of. `ScEpTIC` demonstrated the effectiveness of our techniques in all the different analysis. For example, the analysis of *Memory Inconsistencies* had a speedup of more than *six order of magnitude* with respect to a naive brute-force approach. Furthermore, we also evaluated the performance of our techniques with a lighter version of them, that did not include some analysis elements. We demonstrated that all the components of our techniques are essential for reducing the time that our analysis requires.

We developed `ScEpTIC` with extensibility in mind, and we would like to extend the analysis it supports and increase its performance. In particular, we would like to:

- Implement the analysis that only finds the presence of inconsistencies.

  In `ScEpTIC` we did not implement such analysis, since our purpose was to also analyze how inconsistencies affects the program behavior.

- Parallelize the analysis that `ScEpTIC` performs for identifying the effects of inconsistencies.

  Our techniques execute one test for each checkpoint that we must analyze. Such tests are independent of each other, and thus we can exploit parallelism for improving performance and reducing the analysis execution time.

- Make `ScEpTIC` able to perform an input-independent analysis.

  Currently, `ScEpTIC` requires us to provide the test inputs. For example, if the program senses the temperature from the environment, we must configure the value of this parameter for running the analysis. Moreover, if the program changes its behavior depending on such input, we

must run one test for each value that produces a different execution flow. By making `ScEpTIC` input-independent, we would automatize the entire analysis.

- Implement in `ScEpTIC` an active mode of debugging, that permit us to pause the execution for analyzing the runtime state or changing it.

In conclusion, the literature overlooked most of the inconsistency taxonomies that we analyzed in this thesis. It also did not provide any technique able to analyze in practical time all the possible effects that intermittent executions may cause over the behavior of a program. The analysis and techniques we developed in this thesis aims to increase the reliability of intermittent programs. We provided both a detailed study of the effects that intermittence causes over the device behavior, and a set of techniques enabling the analysis of intermittent executions in practical time. The contributions we made in this thesis enable the programmer understanding, analyzing, and avoiding the problems that frequent shutdowns may cause over the device behavior.

Bibliography

[1] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *SIGPLAN Not.*, vol. 50, pp. 575–585, June 2015.

[2] "MSP430$^{\text{TM}}$ ultra-low-power sensing & measurement MCUs." `http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html`.

[3] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," *SIGARCH Comput. Archit. News*, vol. 39, pp. 159–170, Mar. 2011.

[4] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[5] "Proteus digital health." `https://www.proteus.com/`, 2015.

[6] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. Blaauw, "A modular 1mm3die-stacked sensing platform with optical communication and multi-modal energy harvesting," in *2012 IEEE International Solid-State Circuits Conference*, pp. 402–404, Feb 2012.

[7] E. Sazonov, H. Li, D. Curry, and P. Pillay, "Self-powered sensors for monitoring of highway bridges," *IEEE Sensors Journal*, vol. 9, pp. 1422–1429, Nov 2009.

[8] K. Vijayaraghavan and R. Rajamani, "Novel batteryless wireless sensor for traffic-flow measurement," *IEEE Transactions on Vehicular Technology*, vol. 59, pp. 3249–3260, Sept 2010.

[9] E. Sardini and M. Serpelloni, "Self-powered wireless sensor for air temperature and velocity measurements with energy harvesting capability," *IEEE Transactions on Instrumentation and Measurement*, vol. 60, pp. 1838–1844, May 2011.

[10] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (Berkeley, CA, USA), pp. 17–32, USENIX Association, 2016.

[11] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Systems Letters*, vol. 7, pp. 15–18, March 2015.

[12] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan, "Quickrecall: A hw/sw approach for computing across power cycles in transiently powered computers," *J. Emerg. Technol. Comput. Syst.*, vol. 12, pp. 8:1–8:19, Aug. 2015.

[13] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," *SIGOPS Oper. Syst. Rev.*, vol. 50, pp. 577–589, Mar. 2016.

[14] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, (New York, NY, USA), pp. 5:1–5:3, ACM, 2014.

[15] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proc. ACM Program. Lang.*, vol. 1, pp. 96:1–96:30, Oct. 2017.

[16] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," *SIGPLAN Not.*, vol. 51, pp. 514–530, Oct. 2016.

[17] M. Furlong, J. Hester, K. Storer, and J. Sorber, "Realistic simulation for tiny batteryless sensors," in *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'16, (New York, NY, USA), pp. 23–26, ACM, 2016.

[18] A. Colin and B. Lucia, "Termination checking and task decomposition for task-based intermittent programs," in *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, (New York, NY, USA), pp. 116–127, ACM, 2018.

[19] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, (New York, NY, USA), pp. 21:1–21:6, ACM, 2017.

[20] J. Hester, T. Scott, and J. Sorber, "Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, (New York, NY, USA), pp. 1–15, ACM, 2014.

[21] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt, "Mspsim - an extensible simulator for msp430-equipped sensor boards," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, 2007.

[22] "LLVM language reference manual." `https://llvm.org/docs/LangRef.html`.

[23] "The LLVM compiler infrastructure." `https://llvm.org/`.

[24] N. Shasha and S. Toledo, "Storing a persistent transactional object heap on flash memory," in *IEEE International Conference on Software-Science, Technology Engineering (SwSTE'07)*, pp. 66–76, Oct 2007.

[25] "clang: a C language family frontend for LLVM." `http://clang.llvm.org/`.

[26] "LLVM language instruction reference." `https://llvm.org/docs/LangRef.html#instruction-reference`.

[27] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, pp. 895–913, Sept. 1999.

[28] "MSP430$^{\text{TM}}$ datasheet." `http://www.ti.com/lit/ds/symlink/msp430fr5737.pdf`.

[29] "Mibench2 porting to IoT devices." `https://github.com/impedimentToProgress/MiBench2`.

[30] 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 the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.