Fast and Energy-efficient Intermittent Computing

by

Saad Ahmed

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

LAHORE UNIVERSITY OF MANAGEMENT SCIENCES

Author .....
Department of Computer Science

Certified by.....
Dr. Muhammad Hamad Alizai
Assistant Professor
Thesis Supervisor

Abstract

Transiently-powered computers (TPCs) form the foundation of the battery-less Internet of Things, using energy harvesting and small capacitors to power their operation. This kind of energy supply breaks a fundamental assumption in existing computing systems, i.e., uninterrupted operation. The resulting computing pattern thus becomes *intermittent*: periods of normal computation and periods of energy harvesting come to be unpredictably interleaved. In this thesis, we identify key challenges and propose solutions to support such an intermittent computing pattern.

First, TPCs ensure forward progress of programs through state checkpointing in non-volatile memory. Checkpointing is, however, expensive in energy and adds to the execution times. To minimize this overhead, we present **DICE**, a system that renders differential checkpointing profitable on these devices. DICE is unique because it is software-only and efficient because it only operates in volatile main memory to evaluate the differential. Any arbitrary code can be enabled with DICE using automatic code-instrumentation requiring no additional programmer effort. The impact on final performance is striking: a DICE-enabled system requires one order of magnitude fewer checkpoints and one order of magnitude shorter time to complete a workload in real world settings.

Second, energy harvesting based supply is characterized by extreme variations in supply voltage, as capacitors charge when harvesting energy and discharge when computing. We experimentally find that these variations cause marked fluctuations in clock speed and power consumption, which determine energy efficiency. We demonstrate that it is possible to accurately model and concretely capitalize on these fluctuations. We derive an energy model as a function of supply voltage and develop **EPIC**, a compile-time energy analysis tool. When using EPIC with existing TPC system support, run-time energy efficiency drastically improves, eventually leading up to a 350% speedup in the time to complete a fixed workload.

Third, we present **D^{2}VFS**, a run-time technique to intelligently regulate supply voltage and accordingly reconfigure clock frequency of intermittently-computing de-

vices. Statically setting clock frequency of TPC fails to achieve energy efficiency, as the setting remains oblivious of fluctuations in capacitor voltage and of their impact on a microcontroller operating range. In contrast, D^{2}VFS captures these dynamics and places the microcontroller in the most efficient configuration by regulating the microcontroller supply voltage and changing its clock frequency. Our evaluation shows that D^{2}VFS markedly increases energy efficiency; for example, ultimately enabling a 30-300% reduction of workload completion times.

Together, these three contributions make intermittent computing *fast* and *energy-efficient*, bridging the gap between the available energy and power demands of a TPC, which goes a long way in securing a battery less future for IoT.

Acknowledgments

First of all, who else to thank but Almighty Allah; the Beneficent, the Merciful.

I owe my profound gratitude to Dr. Hamad Alizai whose mentorship and guidance did wonders to raise the confidence of a shy and introvert student like me. I can vividly remember the day when I was presenting at ComSys: the amount of confidence he showed in me, to do good in front of some very good researchers (when I had no confidence in myself), helped me feel as if I am the best researcher in the room. His confidence inculcated self-belief that helped me achieve my goal and to be what I am today. His enthusiasm, work ethic, and energy was always hard to match. I tried my level best but, as I am writing this, I still feel that I could have done better. He left no stone unturned to provide opportunities for me to excel as a researcher, develop my vision, and keep achieving my research milestones in time. I would like to thank Dr. Hamad for *everything* that he did for me; both on and off the record. I am highly indebted to his undaunting support and encouragement. I would also like to thank my collaborator Dr. Junaid Haroon and Dr. Luca Mottola for their wise advices, valuable feedback, and critique which helped me improve my work.

Many thanks to SysNet colleagues (Samar, Hassan, Abubakar, Farhan, Tehreem) for the endless-discussions on almost every topic, fun, and laughters, we had in the lab. I would like to thank my co-authors (Dr. Naveed, Quratulain) as well as thesis student (Junaid) for his input as this dissertation would have been incomplete without his help. I always had the support of our administrative staff Ishtiaq Bhatti and Noor-e-Sehar for managing my conference travels, and formal Ph.D. requirements. They made sure my needs were met while I was focusing on my research work.

And finally, the never-ending support, teachings, prayers of my father, and late mother are the reason I was able to persevere and concentrate on my goals, during my tenure. I thank them for pushing me and helping me in becoming a better person. Last but not the least, I would like to thank my wife for her love, motivational talks, and prayers. Thank you for being patient with me and always showing me the bright of side of things. Now onto the next journey together.

Contents

1 Introduction **23**

1.1 Problem Statement 25

1.2 Observations 26

1.3 Thesis Statement and Major Contributions 28

1.3.1 Efficient Differential Checkpointing 28

1.3.2 Modeling Dynamic Energy Consumption 29

1.3.3 Fast Program Execution 29

1.4 TPC Device Architecture 30

1.5 Limitations 31

1.6 Structure 31

2 Intermittent program execution: Challenges and Taxonomy **33**

2.1 TPC Dynamics: A Bigger Picture 34

2.1.1 TPC Fundamentals 35

2.1.2 Program-state retention 37

2.2 Intermittent program's state retention:

Taxonomy of solutions 38

2.3 Checkpointing Strategy 40

2.3.1 Copy-all 40

2.3.2 Copy-used 41

2.3.3 Copy-if-change 42

2.3.4 Key Properties 43

2.3.5 Summary 44

2.3.6 Qualitative Comparison With DICE 45
2.4 Checkpoint Triggering Mechanisms 46
2.4.1 Proactive 46
2.4.2 Reactive 50
2.4.3 Key Properties 52
2.4.4 Summary 52
2.4.5 Qualitative Comparison with EPIC 54
2.5 Non-volatile Hardware Support 54
2.5.1 Mixed Volatility Main Memory 55
2.5.2 Fully Non-volatile Main Memory 56
2.5.3 Non-volatile processors 58
2.5.4 Key Properties 59
2.5.5 Summary 60
2.5.6 Observation 62
2.6 Conclusion 63

3 Differential Checkpointing **65**
3.1 Motivation 66
3.1.1 Requirements 67
3.1.2 Major Contributions 68
3.2 Related Work 69
3.2.1 Mainstream Computing 69
3.2.2 Intermittent Computing 70
3.3 Scrutinizing Differential Checkpointing Techniques 72
3.3.1 Hashing 73
3.3.2 Tracking 73
3.3.3 Static analysis 74
3.3.4 Comparative Analysis 75
3.4 DICE In A Nutshell 78

3.5 Recording Differentials . . . 80
3.5.1 Global Context . . . 80
3.5.2 Call Stack . . . 83
3.5.3 Pointer Dereferencing . . . 85
3.6 Implementation . . . 85
3.7 Benchmark Evaluation . . . 87
3.7.1 Settings . . . 88
3.7.2 Results → Update Size . . . 90
3.7.3 Results → Smallest Energy Buffer . . . 91
3.7.4 Results → Checkpoints . . . 92
3.7.5 Results → Completion Time . . . 96
3.8 Application Evaluation . . . 97
3.9 Summary . . . 102

4 Modeling Dynamic Energy Consumption 105
4.1 Motivation . . . 106
4.1.1 Observation . . . 107
4.1.2 Major Contributions . . . 109
4.2 Background . . . 110
4.3 Energy Model . . . 112
4.3.1 Modeling Clock Drift . . . 113
4.3.2 Modeling Dynamic Power Consumption . . . 116
4.3.3 Model Representation . . . 117
4.3.4 Summary of Findings . . . 119
4.4 EPIC . . . 120
4.4.1 Workflow . . . 120
4.4.2 Finding MCU Cycles per Basic Block . . . 122
4.4.3 Microbenchmarks . . . 124
4.4.4 EPIC with HarvOS . . . 125
4.4.5 EPIC with CleanCut . . . 130

4.5 MSPsim++ . . . 133
4.5.1 Design . . . 133
4.5.2 MSPsim++ with MementOS . . . 134
4.6 Discussion . . . 137
4.7 Related Work . . . 138
4.8 Summary . . . 140

5 Fast Program Execution **141**
5.1 Motivation . . . 143
5.1.1 Challenge and Opportunity . . . 143
5.1.2 Major Contributions . . . 145
5.2 Overview . . . 146
5.3 D^{2}VFS . . . 148
5.3.1 Architecture . . . 148
5.3.2 Implementation . . . 152
5.4 Benchmark Evaluation . . . 154
5.4.1 Settings . . . 155
5.4.2 Power profile and measurements . . . 156
5.4.3 Results → Computation per Active Epoch . . . 156
5.4.4 Results → Checkpoints and Energy Buffers . . . 157
5.4.5 Results → Completion Time . . . 159
5.5 Real World Evaluation . . . 161
5.6 Related Work . . . 163
5.7 Summary . . . 165

6 Discussion and Conclusion **167**
6.1 Summary . . . 168
6.1.1 Differential Checkpointing . . . 168
6.1.2 Modeling Dynamic Energy Consumption . . . 170
6.1.3 D^{2}VFS . . . 171
6.2 Limitations . . . 172

6.2.1 Differential Checkpointing 172
6.2.2 Modeling Dynamic Energy Consumption 173
6.2.3 Fast Program Execution 173
6.3 Future Work 173
6.3.1 Approximate Computing 173
6.3.2 Secure and Reliable checkpointing 174

List of Figures

2-1 TPC's Energy Cycle:*A basic illustration of how energy is consumed during TPC's operation. A major chunk of it is consumed in checkpointing and restoring the state; leaving very little for program execution.* 34

2-2 TPC's State Retention. *Retaining TPC's state across periods of energy unavailability involves state-retention of three sub-systems of TPC i.e., Peripherals, Timers and Application's Program, to ensure correct resumption in the next active epoch.* 37

2-3 Program-state Retention is energy hungry. *With energy being a scarce resource, TPC spend energy on devising efficient checkpointing strategy, its precise triggering mechanism and employing additional hardware support for making the program-state persistent across reboots, to avoid checkpointing in the first place.* 39

2-4 Checkpointing Strategies. *Based on the approach to determine change in program-state, existing checkpointing strategies can be divided into three main categories; Copy-all, Copy-used and Copy-if-change.* . . . 40

2-5 Triggering Mechanisms.*TPCs proactively track energy buffer state or employ additional hardware to react to trigger checkpointing, when the energy buffer is low.* 47

2-6 Hardware Support. *Three different architectures of transiently-powered computers based on extent of non-volatility of program-state supported.* 55

3-1 Intermittent computing solutions. *DICE targets traditional device architectures with volatile main memory and seamlessly integrates with all the corresponding solutions.* 70

3-2 Computational overhead of differential checkpointing techniques in FFT application. 75

3-3 DICE fundamental operation. *DICE updates checkpoint data based on differentials at variable level in the global context, or with modified stack frames.* 78

3-4 Qualitative comparison of the checkpoint techniques. Copy-all *has highest energy costs due to maximum writes on NVM.* Copy-used *avoids copying unused memory areas, reducing the energy cost.* Copy-if-change *further reduces energy costs, using NVM reads to compute differentials. DICE records changes in the stack at frame granularity, but only operates in main memory.* 78

3-5 Example instrumented code. 81

3-6 Example instrumentation for reactive or proactive checkpoints. *With reactive checkpoints, each statement possibly changing global context data is preceded by a call to* **`record()`**. *With proactive checkpoints, code locations where a checkpoint may take place are known, so calls to* **`record()`** *may be aggregated to reduce overhead.* 82

3-7 Identifying possibly modified stack frames. *The stack tracker (ST) is reset to the base pointer (BP) when the system resumes or at startup. ST does not follow BP as the stack grows, but it does so as the stack shrinks. The dark grey region between ST and the stack pointer (SP) is possibly modified.* 84

3-8 Example instrumented code to record local variables that are passed by reference. 85

3-9 Update size. *The size of NVM updates is significantly smaller when using DICE compared to the original systems. Compared to* copy-if-change*, it remains the same or marginally larger.* 90

3-10 Smallest capacitor. *A DICE-equipped system completes the workload with smaller energy buffers. This is due to a reduction in the energy cost of checkpoints, enabled by the reduction in update size.* 92

3-11 Number of checkpoints necessary against varying capacitor sizes. *With smaller capacitors, highly-intermittent executions greatly benefit from DICE.* 93

3-12 Improvement at different capacitor sizes enabled by DICE. *The improvements are larger in highly-intermittent executions.* 94

3-13 Completion time without concrete checkpoints. *The additional run-time overhead due to code instrumentation is limited.* 95

3-14 Completion time including checkpoints. *The run-time overhead due to DICE is overturn by reducing size and number of checkpoints, ultimately resulting in shorter completion times.* 96

3-15 Example voltage traces and device setup. 98

3-16 Performance of the AR application running on Hibernus with a 50uF capacitor. *Performance gains are observed across diverse power traces obtained from different energy sources.* 99

3-17 Performance of the AR application running on Hibernus with a 100uF capacitor. *Compared with Figure 3-16, and as seen in Section 3.7, performance improvements are larger with more intermittent executions.* 100

3-18 Performance of the AR application running on Hibernus with a 50uF capacitor. *Reduced off-time is observed across diverse power traces obtained from different energy sources.* 101

4-1 Impact of supply voltage variations on MSP430G2553 clock speed and power consumption in a single power cycle. *Existing tools typically ignore these behaviors when modeling the energy consumption of TPC.* 108

4-2 Impact of supply voltage variations on MSP430G2553 power consumption and clock speed. *Energy being a product of power (red dotted line) and execution time, which is a function of clock speed (black solid line), the energy cost of a single MCU cycle varies by up to ≈5× depending on the instantaneous supply voltage.* 109

4-3 Dependencies among quantities determining energy consumption. *The direction of the arrows depict the direction of dependency. Power consumption depends upon the input voltage and the current draw. The execution time depends on clock speed. As the supply voltage rapidly varies, additional dependencies are created, shown by black arrows, that perturb an otherwise constant behavior.* 110

4-4 Impact of DCO parameters on clock. 112

4-5 Clock frequency measurements. *The number of clock cycles (f_{DCO}) are counted between A and B, namely, two consecutive low to high transitions of the external crystal oscillator f_{ext}.* 113

4-6 Impact of voltage supply variations on clock, current, and power for all DCO configurations. *The x-axis shows the difference in the corresponding factor when the voltage drops from one extreme of the MCU's operational voltage range to the other.* 113

4-7 Clock behavior. *(a) The sensitivity of clock to voltage increases for a given RSELx when the value of DCOx is increased, as well as across increasing RSELx values. The percentage values represent the difference in frequency between the two extremes of the operational voltage range; (b) The cup-shaped segments, with each cup corresponding to a single power cycle, show that the number of unrepresented cycle increases with the decreasing voltage of the capacitor. Charging times are omitted for brevity.* 115

4-8 Model representations and their behavior compared to empirical measurements @ 8 MHz. *Higher precision lookup tables follow closely the actual measurements. The polynomial model rests within a 2% error bound.* 118

4-9 Dynamic vs. constant energy models @ 8 MHz. *Depending on supply voltage, a constant model may overestimate (shaded blue region) or underestimate (shaded red region) the energy available with a single discharge of a 10μF capacitor.* 119

4-10 EPIC code instrumentation process. *The mapper maps the assembly instructions to the corresponding basic blocks in the source code and outputs a CFG and energy profile based on the models of Section 4.3. The analyzer traverses the CFG to compute best- and worst-case estimates for a given fragment of code and capacitor size.* 121

4-11 Voltage trace from indoor light using a mono-crystalline solar panel. . 126

4-12 HarvOS results. *The benefits of using EPIC within HarvOS apply across different capacitor sizes.* 129

4-13 CleanCut results. *Using EPIC allows CleanCut to avoid false positives when looking for non-termination bugs.* 132

4-14 MementOS results. *Across different capacitor sizes, MSPsim++ constantly indicates lower voltage thresholds than MSPsim.* 137

5-1 Variations in energy consumption per clock cycle with varying supply voltages. *The graph plots the energy consumption for the most commonly used and factory calibrated frequencies of MSP430G2553 MCUs. Higher frequencies are energy efficient but only operate in a limited range of supply voltage. Lower frequencies consumer more energy but enjoy a larger supply voltage range. The highlighted region indicates the best performing clock setting in a certain voltage range.* 144

5-2 First order approximation of clock cycles for static configurations (1 MHz, 8 MHz, 12 MHz, and 16 MHz) and dynamic frequency scaling within a single charge of the capacitor (10 μF). *Dynamic scaling offers a substantial margin for improvement. The plot does not account for the overhead to enable dynamic scaling.* 146

5-3 D^{2}VFS architecture. *Voltage detectors intercept the capacitor voltage to detect changepoints. The D^{2}VFS driver scales the frequency at these changepoints and reconfigures the voltage regulators to deliver the minimum required voltage for reducing energy consumption.* 148

5-4 Voltage supply range and current consumption of factory calibrated frequencies. 149

5-5 Working of D^{2}VFS. *At each changepoint, the D^{2}VFS driver determines whether or not place the MCU in a new operational zone {f (MHz), V}.* 150

5-6 D^{2}VFS schematic implementation in EasyEDA. 152

5-7 Cycles per active epoch. *D^{2}VFS retains its benefits over static frequency settings despite the implementation overhead due to additional hardware and the D^{2}VFS driver.* 157

5-8 Number of checkpoints necessary against varying capacitor sizes. *D^{2}VFS allows the system to progress farther on a single charge, reducing the number of checkpoints needed to complete a workload.* 158

5-9 Smallest capacitor. *A D^{2}VFS-equipped system completes the workload with smaller capacitors. This is due an increase in available clock cycles in a single active epoch.* 158

5-10 Completion time. *The run-time overhead due to D^{2}VFS is overturn by increasing the number of clock cycles available, and therefore the amount of computation possible in a single active epoch, and thus reducing the number of checkpoints. The combined effect shortens completion times.* 160

5-11 Voltage traces used for evaluation. 161

5-12 Performance of the AR app running on Hibernus. *Performance gains with D^2VFS are observed across diverse power traces obtained from different energy sources. The average improvement compared to static frequency settings ranges from 30% to 300% across all traces.* 163

List of Tables

2.1 Feature Comparison of checkpointing strategies: All existing strategies are hampered by their inability to find checkpoint differentials while reducing NVM accesses. DICE addresses this limitation of existing state-of-the-art solutions by proposing a differential checkpointing mechanism that reduces NVM accesses; requiring the smallest checkpointing energy. 44

2.2 Feature Comparison. Existing state-of-the-art solutions assume a constant energy model and require user intervention in one way or another. EPIC provides a tool for precise trigger call placement, incorporated with a dynamic energy model. 53

2.3 Micro-measurements for non-volatile and volatile main memory . . . 58

2.4 Feature Comparison of Hardware Support. Although existing solutions achieve checkpoint size reduction, the gained benefit is lost due to high energy consumption during program execution. 60

3.1 Memory overhead of different techniques in AR application 76

4.1 EPIC's accuracy. 124

4.2 Features and performance of HarvOS-instrumented code with and without EPIC. 127

4.3 CleanCut non-termination bug warnings with and without EPIC. . . 131

4.4 Parameter setting and performance of MementOS applications with MSPsim or MSPsim++. 136

5.1 D^2VFS changepoint decisions. 151
5.2 Salient features of D^2VFS voltage detectors. 153

6.1 DICE Summary: DICE uses a differential checkpointing technique which helps it in reducing checkpoint size. It uses an in-memory data to record checkpoint differentials which reduces NVM accesses. 169
6.2 EPIC's summary: Precise modeling of clock and current behavior enables EPIC to significantly reduce the number of checkpoint required for completing intermittent programs. This reduction translates into gain in speedup by orders of magnitude. 170
6.3 D^2VFS summary: Our approach dynamically adapts to the changing energy condition and enables TPC to execute more number of clock cycles in the given energy budget; resulting reduced completion time for applications. 171

Chapter 1

Introduction

The vision of "smart dust" [139] has driven the development of tiny embedded sensing devices for many years. With the aim of running autonomously for decades, these sensing devices are envisioned to be used in large-scale applications such as wearables [25], implants [25], small satellites [116], and wireless robotic materials [34, 123]. The monitoring of existing infrastructure [141] and the environment using insects carrying the tiny devices has also been recently proposed [37, 78]. Advances in nanotechnology and microelectronics have made it possible for these devices to shrink in size [143]; small enough to fit on the head of a honey bee [35].

Batteries are a problem. Batteries are commonly used to power embedded devices [128, 145]. However, their designs have not scaled in the same way, thus making it hard for them to fit in such devices despite recent developments [37]. Furthermore, the use of batteries nullifies the initial vision of "smart dust" of being invisible and maintenance-free. Modern-day battery designs have only limited number of power cycles i.e., from one power cycle for non-rechargeable to a few thousand for rechargeable batteries. As a result, they have to be frequently replaced from a high number of devices thus preventing a perpetual operation and resulting in high maintenance costs.

To enable the maintenance-free operation, these devices must liberate themselves from batteries by powering from harvested energy [17, 95, 108, 126], e.g., from light, vibrations, and thermal sources. Miniaturized mechanical systems have enabled the

design of tiny energy harvesters at the scale of nanometers that fit into the form factor of these tiny, embedded sensing devices [17], thus making them batteryless. However, the energy that can be harvested from the environment is generally erratic and exhibits high spatial and temporal variations [17]

Intermittent Computing. To ameliorate the resulting fluctuations in energy supply, energy is first collected in a buffer, typically a capacitor. As long as the capacitor voltage is below a predetermined power-up threshold, the device rests dormant until the buffered energy is sufficient to boot. An *active epoch* then starts when the device operates until power fails and the device shuts down. A phase of solely charging then resumes until the device can boot again. These charge-discharge cycles are frequent, as miniaturization required to realize medical implants or visions of "smart dust" prompts energy storage facilities to be minimized [17]. Therefore, one power cycle of this buffer cannot store the energy needed to complete one iteration of program execution on the MCU. The device loses its volatile state when the energy falls below the minimum energy required for normal operation. Inevitably, these devices have to checkpoint their volatile state onto non-volatile memory(NVM) in order to resume it in the next active epoch. In this way, the computations performed by these devices become intermittent; transforming the notion of energy from "limited but continuous" in traditional battery-powered devices to "unrestricted but intermittent" in these batteryless devices. We call these tiny, energy harvesting-based, batteryless, embedded devices as *transiently-powered computers (TPCs)* and the computing paradigm *intermittent computing*.

Maximize Energy Buffer Utilization. The time required by device to recharge its energy buffer during a charging period can range from few milliseconds to minutes, hours or even days. With embedded systems having hard real-time constraints, it becomes extremely important to maximize the number of computations performed in active epoch so that minimum amount of work is left for the next epoch; enabling high availability and reduced execution times.

Previous works have proposed various solutions for reducing the cost and timing of the checkpoint. This dissertation improves the performance of existing systems in

addition to proposing a novel technique for fast execution of program thereby ensuring efficient utilization of the given energy budget.

1.1 Problem Statement

Maximizing energy buffer utilization requires investigation of three main aspects of intermittent computing.(1) Checkpoint-Size (2) Triggering a Checkpoint and (3) Program-Execution.

Checkpoint-size is an integral component in determining the amount of energy required to correctly save application state. Larger the size, more is the checkpointing energy and lesser is the energy available for performing program execution. Strategies employed by existing systems can be divided into three categories. First category [10, 11] copies the entire memory area—including unused or empty portions—onto NVM as part of the checkpoint whereas, second category [18, 130] only copies the *used* segments of main memory onto NVM. Actually, there are very few locations in the application state that are changed from one checkpoint to the next. Therefore, some solutions only copy the changed memory locations by computing checkpoint differential at the time of saving the state; the third category. While this reduces checkpointing energy, these techniques incur significant computational overhead thus losing the benefit gained by reduction in checkpoint size.

TPCs employ compile-time strategies to insert special function calls– also called trigger calls– for tracking the state of energy buffer in order to decide whether to trigger checkpointing or not. These trigger calls perform checkpointing whenever the energy buffer reaches a predefined voltage threshold. Placing these calls in the application program demands accurate estimate of program's energy consumption. Existing approaches lack the ability to model the inherent dynamicity of the incoming transient energy at the time of estimation. Therefore, they have to rely on pessimistic estimates of energy consumption to ensure correct program execution; forcing the system to place unnecessary trigger calls in the program. This results in frequent probing of the energy buffer, in addition to causing redundant checkpoints. As a

result, the completion time for programs increase and, consequently, decreasing the energy-efficiency of the system.

With the aim of maximizing computational progress, recent research efforts propose non-volatile hardware support for application state retention; eradicating the need to checkpoint. While these works enhance energy available for program execution, non-volatile hardware increases the total energy spent on program execution; surpassing the energy saved by reduction in checkpoint size. Furthermore, the hardware employed by these solutions limits TPC's capability to configure the MCU at higher frequencies, as they offer lesser energy-per-cycle. Configuring MCUs at lower clock frequencies result in fewer cycles getting executed per-unit of energy due to their high energy consumption per-cycle. On the other hand, simply configuring systems at higher frequencies does not solve the problem either as they tend to operate in short bursts; inducing higher overhead needed to sustain computations across period of energy availability. TPCs need to dynamically adapt to the changing energy condition in order to execute program at higher frequencies while being operational in the entire voltage range. Therefore, run-time adaptations are of prime importance for TPCs to ensure fast program execution.

1.2 Observations

We observed three key limitations in the existing literature.

Checkpoint size. First, we observed that the actual change in application state of the TPC, from one checkpoint to the next, is very small as compared to the estimates made by the existing literature. Existing techniques either estimate large checkpoint size than the actual change or perform high number of NVM accesses to find the exact change in application state; in both cases, forcing the device to spend a significant energy budget on performing checkpoints. Our observation shows that there are only few statements in the program which are responsible for changing the application state. As a result, the actual change in application state can be estimated in an efficient manner without incurring significant energy overhead.

Triggering a Checkpoint. Second, accurate energy estimation of the program plays an important role in precise placement of trigger calls. Existing techniques assume constant consumption in order to simplify their analysis at the time of trigger call placement. We observed that it is not constant and varies significantly with the variation in input voltage. Our estimates show that the reduction in energy-per-cycle can be as large as 5×, moving from higher end of the operational voltage range to the lower one. This behavior is due to the two fundamental dependencies in the TPC which are, currently, unmodeled in the existing literature. As a result, contrary to the actual energy consumption, existing literature estimates high energy consumption for application program thus losing the available clock cycles which can be used to perform useful computations. For example, our empirical observation shows that the TPC loses 10k cycles in just 10 power cycles; the number keeps on accumulating with increase in the number of power cycles. Therefore, an accurate energy model is required to account for this loss of clock cycles.

Program Execution. Finally, we observed that non-volatile hardware support for eradicating checkpoints consume significant amount of energy during program execution (see Section 2.5.2); implicitly requiring TPCs to be configured at lower frequencies. Configuring the system at a fixed lower clock frequency can result in under utilization of the energy buffer. For example, configuring at 1 MHz, the system consumes ∼1.5× more energy per clock cycle than, when configured at 16 MHz. Our observation shows that dynamically adapting to higher frequencies can result in a minimum of 40% increase in the number of clock cycles available in an active epoch thus allowing the device to make more computational progress than the fixed frequency setting. However, existing hardware architectures for TPC lack the necessary support required to perform such dynamic switching.

These observations form the basis of contributions made by this dissertation and are described in the next section.

1.3 Thesis Statement and Major Contributions

This dissertation seeks to firmly establish the following thesis:

"**Fast and energy-efficient system support can enable long-running computations on the energy-constrained transiently powered devices.**"

To test this thesis statement, we asked following three research questions.

- Can we minimize the amount of program-state that needs to be checkpointed with minimal overhead?
- Can we accurately place trigger calls during program execution to avoid redundant checkpoints?
- Can we maximize the number of program instructions executed in the given energy budget?

This dissertation answers all these questions in affirmative and demonstrates the portability of these approaches in a system stack; from the compiler to the run-time and at the hardware level. Precisely, we propose:

- Run-time technique to perform *differential checkpointing* for **checkpoint size** reduction
- *Accurate Energy modeling* technique for TPCs, incorporated in a compile-time tool, to perform precise **trigger call placement** in the intermittent programs
- An energy efficient *hardware design* to enable faster **program execution** of intermittent programs

1.3.1 Efficient Differential Checkpointing

Since a major chunk of the energy budget is spent on checkpointing by a TPC, we developed a differential checkpointing technique, **DICE**, that tracks checkpoint differentials in the system; allowing them to update the existing checkpoint by writing

only small amount of data on the NVM. Our results show that this approach, not only significantly reduces the total number of checkpoints, but it also causes considerable reduction in the time required by the device to successfully complete one application iteration. This increases responsiveness of these intermittently powered systems even in harsh energy conditions.

1.3.2 Modeling Dynamic Energy Consumption

We present a tool for energy profiling in intermittent computing, **EPIC**, which incorporates the variation of MCU frequency as the capacitor voltage oscillates between one extreme to the other; a phenomenon that was previously unmodeled in the existing literature. By modeling the energy consumption dynamics, this tool provides accurate estimates of program's energy consumption thus allowing precise placement of trigger calls in the intermittent programs; helping the TPC in avoiding unnecessary checkpoints. Our results show that the saved energy is utilized by the device in performing useful computations thus enabling longer execution of programs on the same charge.

1.3.3 Fast Program Execution

Finally, we show that it is possible to avoid high energy consumption during program execution, caused by static configuration of the new hardware designs, in an effort to avoid checkpoints. We argue that the efficient execution of legacy/volatile systems can be leveraged to speedup program execution in the power cycle; helping TPCs in reducing the number of checkpoints required for application completion thus reducing completion time.

We propose a dynamic voltage and frequency scaling technique that adapts to the changing energy conditions in the environment, thereby enabling the TPC to configure itself at the lowest energy-consumption, at any given energy state of the capacitor. Our design for dynamic voltage and frequency scaling at discrete levels, **D^{2}VFS**, allows the TPC to dynamically switch the operating frequency to the best

configuration in the given voltage range. By making more progress in the same energy budget, our hardware design outperforms existing systems which force the system to run on fixed frequency configurations. Our results show that, while existing systems fail to complete application execution under scarce energy environments, D^2VFS enables the device to execute long-running applications in the same energy budget.

1.4 TPC Device Architecture

A commonly-used TPC design consists of a MCU and a harvester which converts some form of ambient energy into electrical energy; used to charge the capacitor. MCU consists of volatile main memory (SRAM) and non-volatile external storage typically based on modern flash technology thus making a ***volatile system***.

Such systems need to save a snapshot of application state onto flash before power blackout; an expensive operation energy-wise as flash requires large sectors of data to be erased before writing. Integrated circuit manufacturers have been considering FRAM as a strong contender for embedded, non-volatile storage, as an alternative to widespread flash, due to former's superior characteristics in terms of power, endurance and read/write speed as shown in table 2.3. These aspects of FRAM are crucial due to TPC's reliance on persistent storage for creating checkpoints used to cross the boundaries of energy unavailability. Recent developments have proposed different modifications in the TPC architecture; discussed in detail in Section 2.5.

This dissertation assumes volatile system as the underlying TPC architecture employing volatile main memory with FRAM/Flash as non-volatile secondary memory. This allows this dissertation to propose solutions for a large set of transiently powered embedded computers which form a major chunk of the IoT deployments .

1.5 Limitations

This dissertation also identifies limitations of proposed approaches. For example, differential checkpointing approach is most beneficial when intermittence in the application's execution is very high. Contrarily, the computational overhead of finding the checkpoint differential overcomes the benefit brought by checkpoint size reduction, when energy available in the environment is abundant. In addition to that, high energy availability increases the time between two consecutive checkpoints. As a result, modifications since the previous checkpoint get accumulated due to increased processing time thus increasing the size of checkpoint. In case of energy modeling, we perform energy profiling for a single MCU family. For a device employing different MCU, we need to measure the energy dynamics of that MCU and have to re-train the energy model.

While performing run-time adaptations, we only focus on increasing the processor frequency. Peripheral devices have different configurations than the processor and require different voltage level for normal operation. Therefore, if a program needs to communicate with the processor, the device must not dynamically switch to a configuration where the peripheral is unable to operate.

We also highlight the limitations of new hardware architectures proposed in recent literature and discuss their application scenarios in Section 2.5.2.

1.6 Structure

The remainder of this dissertation is structured as follows.

Chapter 2 provides background information by revisiting the fundamentals of intermittent computing in addition to defining the scope of the dissertation. It encapsulates the state-of-the-art related work and qualitatively compares them with our proposed mechanisms to establish a formal background for later discussions. It also briefly discusses the trends in hardware support for eradicating checkpoints in TPCs.

Chapter 3 highlights the need for checkpoint size reduction for efficient utilization

of energy budget. It introduces differential checkpointing techniques at the global and local level of application state and presents the design and evaluation of our proposed approach.

Chapter 4 highlights the unmodeled dependencies in the intermittently powered systems, such as dependence of clock frequency on voltage and current on clock frequency. It presents an energy model to capture these dependencies and incorporate it in the tool to allow precise placement of trigger calls in the intermittent programs. It also empirically compares our energy model to the ones used by existing state-of-the-art to show the performance gains.

Chapter 5 presents a dynamic voltage and frequency scaling technique to allow fast execution of programs. It evaluates the efficiency of the systems; both in synthetic and real-world setting to ensure the general applicability of the results.

Chapter 6 concludes this dissertation and points to the future directions for this work.

Chapter 2

Intermittent program execution: Challenges and Taxonomy

In this chapter, we revisit the fundamental concepts of TPCs and their life cycle. We present some of the prominent works that represent the state-of-the-art in the areas of checkpoint size reduction, trigger call placement and fast execution of programs.

There are no articles in the existing literature which encapsulate the challenges and give a taxonomy of solutions related to program-state retention of TPCs. A recent article surveyed several intermittent computing approaches [99] to briefly define challenges and future research directions in the intermittent computing domain. This chapter aims to fill this gap in addition to defining the scope and contribution of this dissertation. It also highlights different challenges of application-state checkpointing for these tiny, batteryless, embedded sensing devices that have entirely different constraints than the ones used in the distributed and parallel computing domain

Overall, we believe that this discussion forms the proper conceptual bases and facilitates a smooth sailing into the technical content that appears in later chapters. This chapter lays a formal background for this dissertation but does not explore all the competing solutions in our target areas: A related work section is devoted in each of the following chapters for this purpose.

The rest of the chapter is structured as follows. Section 2.1 gives the bigger picture of TPC operation while defining basic terminology and challenges faced by

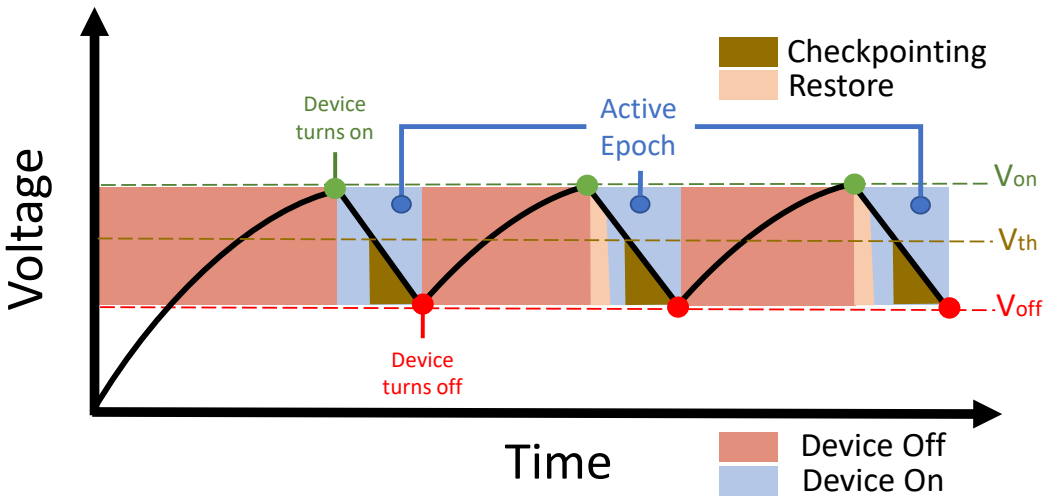


Figure 2-1: TPC's Energy Cycle:*A basic illustration of how energy is consumed during TPC's operation. A major chunk of it is consumed in checkpointing and restoring the state; leaving very little for program execution.*

them in retaining overall system state. Section 2.2 narrows down the scope and explains the criteria of classification for existing solutions for retaining program-state for TPCs. Section 2.3 and 2.4 classify the strategies employed by different intermittent computing solutions to ensure energy-efficiency while performing program-state retention. Section 2.5 discuss non-volatile hardware support employed by TPC to avoid checkpointing and we conclude the chapter in Section 2.6.

2.1 TPC Dynamics: A Bigger Picture

TPCs follow a *sense-compute-send* cycle during their entire life; the device senses value from the sensor (peripheral device)e.g., accelerometer in case of fitness trackers [132], and perform computations on it (MCU) before *send*ing the processed values to the user e.g., fitness trackers [132].

To understand the fundamental operation of TPC, we take the example of an activity recognition (AR) application, which is a classical example of a *sense-compute-send* application commonly used in the fitness trackers.

2.1.1 TPC Fundamentals

Due to variable energy supply, TPCs switch between charging and discharging periods many times during their entire life. Each charging period is followed by a discharging one; forming an energy cycle of these TPCs.

TPC's Energy Cycle. Figure 2-1 shows a typical energy cycle of these TPCs. It starts when a TPC is switched off and is harvesting energy from the environment to charge its buffer. As the capacitor gets fully charged, the voltage level for the capacitor reaches V_{on}– voltage at which the MCU can start executing program instructions– and the TPC is turned on indicating the start of an *active epoch*.

Sense: As program execution of AR application starts, it accesses an accelerometer sensor to collect information about the current position of the device. Once the data is read from the peripheral device, it is time-stamped and saved onto NVM for later use.

Compute: The device then trains its classifier on the sensed data which is used to predict the current activity of the user. This involves transforming the accelerometer samples, extracting features, classification of sample in addition to recording the number of accurate predictions during execution.

Send: After performing application-specific analytics, TPC sends this final result over the network to either the base station or the cloud server using anyone of the heterogeneous network protocols available [77]; marking completion of one application iteration.

These operations must be performed in a continuous manner in order to give correct results. However, the increasing complexity of applications demand diverse set of peripherals and require complex algorithms, such as neural networks, to perform analysis thus increasing the burden on the small energy buffer. Furthermore, these applications also generate huge amount of data to be sent over the network which further puts strain on the energy buffer; exhausting it faster than it could be replenished [53, 75]. When capacitor voltage reaches V_{off}, the device turns off and starts harvesting energy to charge its buffer; indicating the end of active epoch.

With stringent resources at their disposal, TPCs are never able to complete the *sense-compute-send* operation in the given energy budget and face frequent interruptions in one complete iteration of this operation. Checkpointing must be performed during the active period to ensure forward progress of the application. It must be triggered at a particular voltage V_{th}–the threshold voltage indicating energy that is only sufficient for the checkpointing process– and no further program execution can be done afterwards, as shown in Figure 2-1. Any execution of a program beyond this voltage level will result in an inconsistent and faulty checkpointed program-state.

Challenges. Various techniques have been proposed in main-stream computing to efficiently perform state retention [24, 30]. However, the challenges faced by TPCs are completely different from the ones faced by main-stream systems. While they assume continuous power supply, TPCs face frequent power interruptions which can occur at any time during the *sense-compute-send* operation giving rise to unique set of challenges, required to be addressed for successful state retention for TPCs.

- *Peripheral state retention:* TPCs can lose power in the middle of a *sensing* operation e.g., during data acquisition from sensors or data transmission. Peripheral devices require reconfiguration after each power failure and a simple snapshot of peripheral state can result in either cause program liveness or safety issue [21]. MCUs either need to maintain an operational log [133] or require a device context to be saved in NVM to ensure peripheral state retention [14]. However, this still does not guarantee correct program execution as peripheral devices can also operate asynchronously [14, 133].

- *Program-state retention:* While performing *computations*, TPCs require integrated system support to retain the state of the program running on the MCU. This system support helps the TPC in deciding, *when* and *what* amount of program-state is required to be saved for successful *program-state retention*. However, deciding *when* and *what* portion of program state to be saved is an ongoing research problem [10, 11, 16, 18] involving either modified compilation techniques or requiring programmer's effort [32, 100].

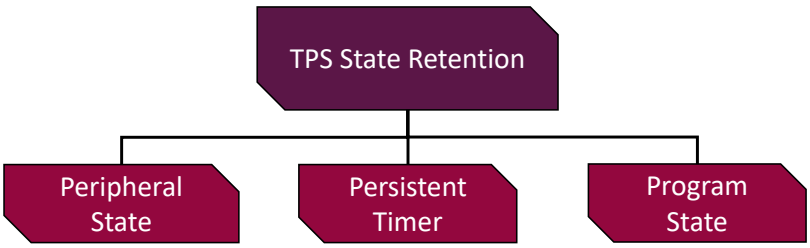


Figure 2-2: TPC's State Retention. *Retaining TPC's state across periods of energy unavailability involves state-retention of three sub-systems of TPC i.e., Peripherals, Timers and Application's Program, to ensure correct resumption in the next active epoch.*

- *Persistent time keeping:* In order to *send* timestamped data, keeping the persistent notion of time on these TPCs is a challenging task, as the traditional timekeeping options i.e., real-time clocks (RTCs), stop working at each power blackout. The time interval between two sensed values can be of seconds, minutes or even hours, if an energy interruption occurs between them [58]; rendering most sensed values stale. TPCs usually perform tasks in a synchronized manner in the network [47]. With incorrect timing information, these devices can get unsynchronized with other devices after each power failure. Thus, data transmission and reception in an unsynchronized, duty-cycled network becomes challenging as both sender and receiver spend much of their time synchronizing themselves. There are solutions proposing languages and run-time to ensure persistent time across reboots, however, it is still an active research area [59,60].

Figure 2-2 categorizes the challenges in retaining the TPC's system state. Catering all of them ensures correct execution of programs on TPC; a difficult task as each one of them is orthogonal to the other.

To facilitate an in-depth discussion, this dissertation focuses only on challenges faced while retaining program-state.

2.1.2 Program-state retention

The program-state in TPC comprise all global variables, system stack, dynamic data structures, general- and special-purpose registers (GPRs and SPRs). Global variables

are placed in **.data**, **.bss** segments of the main memory whereas all dynamic data structures are placed on the **heap** segment. All function call frames are pushed to the **stack** memory segment. Since the main memory is volatile, MCU has to save its current program-state onto NVM just before switching to the inactive period so that it can be revived at the next active epoch. *Checkpoint* of a program-state comprise MCU registers, **.data**, **.bss**, **stack** and **heap** memory segments that reside in the volatile memory. A checkpoint represents the program-state running on TPCs at any point in time and we call, the process of saving the current program-state onto NVM (the checkpoint), as *checkpointing*. All memory segments contain different parts of the program-state and missing anyone will result in a faulty checkpoint. This checkpointed state is *restored* at the start of the next active epoch and requires a very small amount of energy due to cheaper read operations as shown in Figure 2-1.

With promising application scenarios of these embedded sensing devices [25, 35, 116, 139], there is an extensive amount of existing literature proposing various solutions to address the challenges faced by TPCs in retaining program-state. In the next section, we define the rationale for classifying each solution in addition to identifying challenges associated with each class of solution.

2.2 Intermittent program's state retention: Taxonomy of solutions

There are three main challenges in retaining program-state for TPCs, as shown in Figure 2-3.

First, the amount of energy available for TPCs to run program depends on the energy reserved for checkpointing. Larger the amount of energy reserved, lesser is the energy available for the device to run program. A naive way of *checkpointing* program-state is to save the entire main-memory(RAM) onto NVM. With the increasing size of volatile memory in TPC [67], this approach is only going to increase the checkpointing cost. Therefore, a naive solution is neither energy-efficient nor scalable considering the

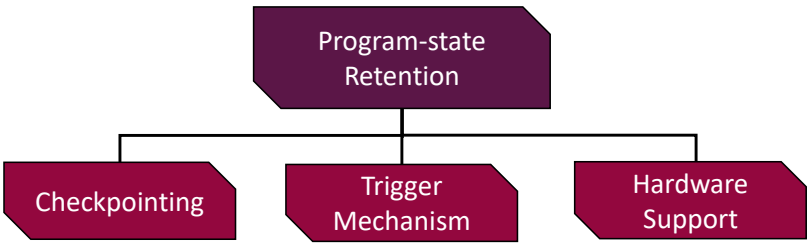


Figure 2-3: Program-state Retention is energy hungry. *With energy being a scarce resource, TPC spend energy on devising efficient checkpointing strategy, its precise triggering mechanism and employing additional hardware support for making the program-state persistent across reboots, to avoid checkpointing in the first place.*

tight energy budget of TPCs; demanding smart checkpointing strategies to achieve the goal of maximizing energy-efficiency.

Second, TPCs either poll the energy buffer through special function calls– trigger calls–or use additional hardware to detect V_{th} in order to trigger checkpointing. Existing state-of-the-art solutions pessimistically assume constant power consumption to estimate program's energy consumption; ending up placing more trigger calls in the program than required thus causing redundant checkpoints. This approach also results in wasted computations as the computations performed, after the last checkpoint till the energy failure, do not become part of any checkpoint; requiring re-execution in the next active epoch. Thus, energy models are required, covering all aspects of dynamic energy consumption on TPCs, to ensure precise placement of trigger calls.

Third, efficient utilization of energy for program execution plays a key role in defining the computational progress made in each active epoch. To maximize the energy available for program execution, existing literature propose non-volatile hardware support for main memory and processors; making program-state persistent across reboots thus eradicating the need to checkpoint. However, such proposals consume significant amount of energy during program-execution and are still far from mass production.

Based on these challenges, existing state-of-the-art solutions can be classified into three main categories. First category discuss, and further classifies, the strategies to reduce the checkpoint size for TPCs. Second category deals with mechanisms

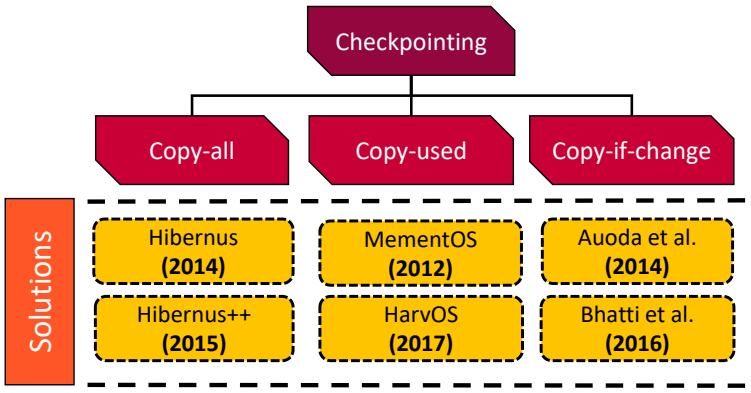


Figure 2-4: Checkpointing Strategies. *Based on the approach to determine change in program-state, existing checkpointing strategies can be divided into three main categories; Copy-all, Copy-used and Copy-if-change.*

for triggering the checkpoint and caters energy estimation for intermittent programs. Third category seeks possibilities of improvement in hardware designs to maximize the computational progress of the device in the given energy budget; a primary goal of TPC.

2.3 Checkpointing Strategy

Checkpointing energy plays a key role in defining the amount of energy available for program execution. As checkpoint size is a key parameter in controlling checkpointing energy, various strategies have been proposed in the existing literature to identify minimal checkpoint size required to be saved, in order to ensure correct application execution. These strategies can be divided into three main categories; namely copy-all, copy-used and copy-if-change, as shown in Figure 2-4. In the next sections, we summarize the contribution of all existing state-of-the-art solutions, while briefly explaining their strategy and discussing their benefits and drawbacks.

2.3.1 Copy-all

An intermittent program can use entire RAM during its execution. Therefore, some of the early approaches in intermittent computing domain proposed saving the entire main memory as part of the checkpoint [10, 11]. This results in a large checkpoint

size thus requiring high checkpointing energy.

Discussion. Feasibility of solutions, belonging to this category, rely on the size of volatile main memory. Smaller the size, lower is the energy required to perform checkpointing. However, recent developments in the TPC's architecture show that the size of volatile main memory is increasing with the increasing complexity of applications [48]; requiring high checkpointing energy for a copy-all approach thus making it unscalable.

2.3.2 Copy-used

Contrary to copy-all, solutions belonging to copy-used category exploit the inherent division of program into different segments.

Ransford et al. [130] identify used memory regions by finding address ranges occupied by each code segment i.e., stack and global variables, at the time of interruption. This allows the system to save only the occupied memory regions of RAM while checkpointing and not the entire RAM; with size of the used memory regions depending on the program-point of interruption. Additionally, they do not track **`heap`** segment of memory thus limiting the programmers by not allowing dynamic data structures.

Bhatti et al. [16] addresses this problem and propose two different variants of copy-used checkpointing strategies, covering all memory segments; namely, Split and Heap Tracker. **Split** makes use of existing segregation of RAM in different segments namely **`.bss`**, **`.data`**, **`heap`** and **`stack`**. This approach is, however, unable to identify memory fragmentation that may exist in heap due to `free()` function. To cater for fragmentation in the heap, Bhatti et al. [16] propose **Heap-tracker**; a technique that keeps track of the heap allocations using wrapper functions for `malloc()` and `free()`. By dividing the memory into blocks, Heap-tracker increments (decrements) the count of the block in which the address is allocated (de-allocated) when `malloc()` (`free()`) is executed.

Discussion. Copy-used solutions reduce the size of checkpoint by only saving used memory regions of the main-memory while bearing minimal computational overhead.

They perform coarse grained analysis of the program to estimate checkpoint size which avoids the need to perform run-time tracking thus reducing the amount of energy required for checkpointing. However, such an analysis overestimates the update size; marking all used memory regions as part of the checkpoint while a large chunk of these regions remain unchanged from the previous checkpoint. Therefore, solutions under this category result in redundant memory writes to the NVM thus wasting energy.

There are other solutions proposing language support to reduce the used memory regions of the program by allowing programmer to make changes in the predefined memory regions only [32,100,104]. However, employing such support gives rise to new set of challenges. First, it demands the user to learn new language thus hampering its wide scale adoption. Second, it increases the run-time overhead which puts a burden on the constrained energy budget. Furthermore, even with these challenges resolved, there is still a possibility of having redundant data in the used memory regions [32].

2.3.3 Copy-if-change

Checkpointing solutions falling under this category remove all unchanged memory locations from the checkpoint size at the time of saving the state. These solutions exactly identify changed memory locations which saves checkpointing energy, but only at the cost of performing run-time tracking.

Aouda et al. [8] propose a technique which makes incremental changes to the checkpointed program-state in the NVM. It divides the program-state into blocks while maintaining a hash of each block. At the time of checkpoint, it compares previously computed hash value with the new ones in order to identify modified blocks in the application-state. Each modified block is then copied to the NVM. To avoid a faulty checkpoint, this approach maintains two images of the checkpoint at a time; active and scratch image. All changes are made to the scratch image and both images switch their roles, once the checkpoint is successfully updated.

As read operations are cheaper than writes [26], there are approaches which read the entire RAM state and compare it with the checkpointed state [16] to exactly

identify changes made after the last checkpoint. To make their approach efficient, they divide the entire RAM into blocks defined by the smallest writable units of NVM. At each checkpoint call, they only update the corresponding chunk of memory in the NVM. However, if nothing changes from the previous checkpoint, this approach still needs to sweep the entire RAM, which wastes energy.

Discussion. Copy-if-change strategies provide the least amount of checkpoint size to be saved at the time of checkpoint thus requiring lower energy than the copy-used category. However, run-time tracking performed by copy-if-change approaches have a significant computational overhead. It is either because of byte-by-byte comparison or due to computation of hashes to find checkpoint differentials. This additional energy consumed by the system is subtracted from the energy budget available for program execution and is directly proportional to the memory access patterns of the application as well as the time interval between two checkpoints. More widespread and frequent accesses of memory can result in a larger change in program-state thereby requiring more energy to save the checkpoint. This will also increase the amount of energy spent in tracking those changes. Therefore, a copy-if-change solution is only feasible if the additional energy overhead caused by run-time tracking is always greater than the amount of energy saved due to checkpoint size reduction.

2.3.4 Key Properties

After discussing the basic operational details of existing checkpointing strategies, we now define key properties which, in our opinion, are essential for the design of an energy-efficient checkpointing strategy. These properties will help us rate the checkpointing energy required by state-of-the-art techniques and shed light on their benefits and drawbacks.

- **Cover Entire Volatile Memory:** Covering all memory segments are essential in maintaining correct program state across reboots. Therefore, it is necessary for each checkpointing solution to save all main memory segments onto NVM.
- **Differential Checkpoints:** High intermittence in energy supply allow very

Technique	Covers Entire Main Memory	Differential checkpoints	Minimizes NVM accesses	Checkpointing Energy
Copy-all				
Hibernus [11]	✓	✗	✗	●●●●●
Hibernus++ [10]	✓	✗	✗	●●●●●
Copy-used				
MementOS [130]	✗	✗	✗	●●●●
HarvOS [18]	✗	✗	✗	●●●●
Copy-if-change				
Auoda et al. [8]	✓	✓	✗	●●●
Bhatti et al. [16]	✓	✓	✗	●●
DICE [3]	✓	✓	✓	●

Table 2.1: Feature Comparison of checkpointing strategies: All existing strategies are hampered by their inability to find checkpoint differentials while reducing NVM accesses. DICE addresses this limitation of existing state-of-the-art solutions by proposing a differential checkpointing mechanism that reduces NVM accesses; requiring the smallest checkpointing energy.

short periods of program execution. This results in very small changes in the main memory when compared to the size of overall program state. Therefore, checkpointing solutions should aim to reduce the update size by saving only the checkpoint differentials.

- **Minimize NVM accesses:** NVM accesses are energy hungry operations which can consume significant amount of energy budget, if performed in a large number. Therefore, a goal of checkpointing solution is to minimize the amount of reads and writes on NVM in order to save energy.

2.3.5 Summary

Table 2.1 shows feature comparison between all existing state-of-the-art approaches and rates the checkpointing energy required by each approach. In the sixth column, we rate the overall checkpointing energy required by each approach. The rating scales from one(very low) to five points(very high).

Unlike copy-used, copy-all covers all memory segments and copy entire main mem-

ory onto NVM at the time of checkpoint; requiring high NVM accesses. This approach makes an implicit assumption of low main memory sizes and hardware support; an assumption which is getting invalidated with the development of new hardware architectures for TPCs [67]. As a result, the amount of energy required by this class of solution is very high(5 points).

Although copy-used approach checkpoints used memory regions, a primary reason for their energy in-efficiency stems from their coarse grained analysis.These approaches are unable to identify differentials in checkpointed state and perform large-sized updates at the time of checkpoint; requiring frequent NVM accesses and high checkpointing energy(4 points).

In contrast, copy-if-change solutions compute checkpoint differentials to reduce the update size at the time of checkpoint. However, solutions belonging to this class require frequent NVM accesses in order to compute checkpoint differentials. Auoda et al. [8] compute hash values at the granularity of a memory block. Therefore, it has to write the entire block onto NVM even if the actual change is very small, compared to the size of the block; resulting in increased checkpointing energy(3 points). On the other hand, Bhatti et al. [16] sweep the entire NVM to compute the checkpoint. In the worse case, this solution still needs to sweep the entire NVM even when nothing has changed from the previous checkpoint. Since NVM reads are cheaper than writes, checkpointing energy for Auoda et al [8] is slightly higher(3 points) as compared to Bhatti et al. [16](2 points).

2.3.6 Qualitative Comparison With DICE

Existing checkpointing strategies are either hampered by their coarse grained analysis or high NVM accesses. They use program's inherent segmentation or require a byte-by-byte comparison to find the checkpoint differentials; both incurring computational and energy overhead. We observed that there are very few memory locations that get modified from one checkpoint to the other; resulting from the execution of a certain set of program statements forming a small portion of the overall program. Based on this observation, we propose a differential checkpointing solution, DICE, that covers

the entire main memory to track all changes that can occur in the application state during program execution at a very small and an almost constant run-time overhead; giving complete freedom to the programmer while writing applications.

To find checkpoint differentials, DICE uses a pre-compiler to insert special function calls before execution of state-modifying instructions. These function calls help DICE to record the address as well as the size of changed memory location in an in-memory data structure. The use of in-memory data structure helps DICE in avoiding frequent NVM accesses thus saving energy. These subtle design choices help DICE in significantly reducing the checkpointing energy and outperforming all existing state-of-the-art solutions. The design and evaluation of our proposed solution are explained in Section 3.

2.4 Checkpoint Triggering Mechanisms

While it is essential to keep the checkpoint size small, it is equally important to trigger the checkpointing process at the right time in order to successfully complete the process. Ideally, it should be triggered at the very last moment when energy in the buffer is just enough to perform a successful checkpoint. This will result in the most efficient utilization of energy buffer.

However, it is difficult to predict program's energy consumption at the time of compilation [17]. As a result, TPCs have to make pessimistic decisions while placing trigger calls in the program. This approach results in unnecessary checkpoints, even in a case when the energy available in the environment was sufficient to complete the execution of a program.

There are two ways of deciding whether to trigger checkpointing or not, as shown in Figure 2-5. We call these triggering mechanisms as Proactive and Reactive checkpointing.

2.4.1 Proactive

TPC inserts special function calls at different code-points in the program, which probe the energy buffer to decide whether it's the right time to checkpoint or not; we call

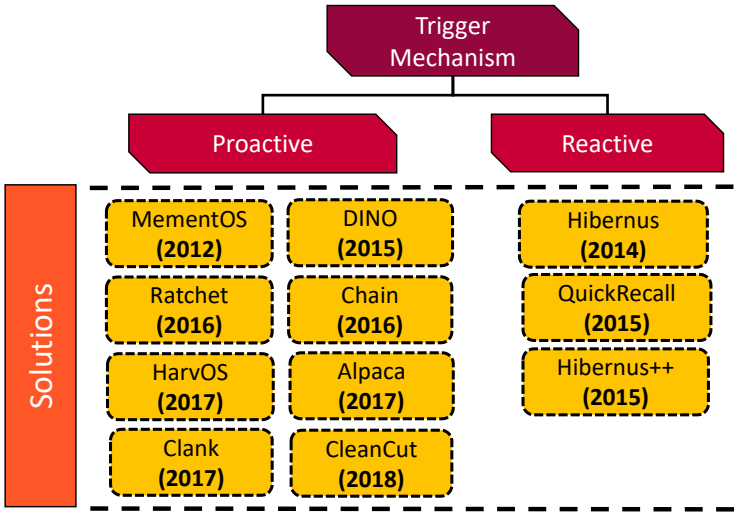


Figure 2-5: Triggering Mechanisms.*TPCs proactively track energy buffer state or employ additional hardware to react to trigger checkpointing, when the energy buffer is low.*

these function calls as trigger calls.

Different solutions propose different candidate code-points for placing these trigger calls in the TPC's program. We can, therefore, categorize proactive triggering mechanism in following three sub-categories based on the choice of code-point.

Loops. A major chunk of computations in any program is performed in loops. Therefore, loops are an important place to insert trigger calls.

Ransford et al. [130] proposed MementOS; a system which statically places trigger calls at the end of loop iterations. It is a location where a program can checkpoint the computations performed in a single iteration of loop. It saves TPC from re-executing the same iteration again in the next active epoch. However, this approach does not ensure correct execution if the energy consumption between any two trigger calls is more than the capacity of the energy buffer; giving rise to live-lock problem.

Bhatti et al. proposed HarvOS [18] to address this issue by placing two trigger calls; First, inside the loop to decide if energy to execute next iteration is available and second, at the end of the loop execution to check if there is sufficient energy to go to the next trigger call. HarvOS [18] uses constant energy model to estimate energy required to reach the next trigger call; assuming each instruction to be executed at the constant maximum voltage regardless of the state of energy buffer.

Branch. Since TPC's MCU is unable to speculate the next instruction after branch, it is unsure whether it would successfully reach the trigger call placed at some other code-point in the program. Therefore, branch instructions are an important code point to make a triggering decision.

HarvOS [18] places trigger calls after each branching construct. If there exists a point with minimum memory allocation inside the scope of branching construct, an additional trigger call is placed at that point. This rule forces the solution to probe energy buffer to find an exact value of remaining energy as it is unaware of the energy cost of the basic block(s) executed in that construct.

Function-return statements are special branch instructions where the stack size is reduced due to removal of stack frame thus requiring small energy cost of saving application state onto NVM. Based on this motivation, MementOS [130] also proposed a strategy to place trigger calls after each function return. However, as discussed earlier, this approach does not ensure correct execution when program execution is unable to reach from one trigger call to the next anywhere in the program.

To ensure correctness, HarvOS [18] estimates the maximum number of cycles (C_{use}), that are available for performing computations, for a particular energy buffer size. They analyze the memory allocation pattern of the program to find maximum checkpoint size (worst case memory allocation). It, then, places two trigger calls in the code; first trigger call is inserted at the point which is at most $\frac{C_{use}}{2}$ cycles away from the previous trigger call; second trigger call is placed after each function return. Hardware and software interrupt are other ways of branching and adds another dimension in challenges as they can come anytime (non-deterministic interrupts) and their execution length is unknown.

To handle interrupts, HarvOS [18] treat them in the same way as function calls and additionally place two trigger calls. The first trigger call is placed right at the very first and the second one is placed at the very last instruction. This is required as ISR has no information on where the execution code was interrupted so it has to read the energy buffer to check the remaining energy. The second trigger is placed in the caller's code and is helpful in dealing the corner case where the remaining energy

was exactly equal to the next trigger call (in the callee's code).

Some solutions [32, 100, 104] provide APIs to the programmer to write the program as a set of atomic tasks. Each task can be considered as a function definition with function boundary acting as a checkpoint call. In this way, checkpoint triggering mechanism is integrated with language semantic thus allowing programmer to know the location of checkpoint during program execution.

Discussion. Existing proactive triggering mechanism only focus on the code-points where either the checkpoint size is minimum or the computational progress is maximum [18,32,100,104,130]. Recent hardware designs for TPCs give rise to idempotence violations; caused due to re-execution of idempotent code sections which are separated by write-after-read data dependence. Therefore, Woude et al. [152] proposed Ratchet: a solution which proactively inserts trigger calls between each write-after-read data dependence in order to avoid re-execution of idempotent code.

Ideally, the distance between two trigger calls must be strictly less than the energy budget of the device. To ensure this distance is within limits, HarvOS [18] models the energy consumption of each basic block by performing manual measurements and assuming constant power consumption across each block. Colin et al. [33] proposed CleanCut which also uses a similar model to hunt non-termination bugs in the systems providing language support. This helps the programmer in identifying the suitable size of energy buffer for device operation. Although a pessimistic constant power assumption ensures correct program execution, it results in high amount of energy wasted in performing computations which never become part of any checkpoint.

To ensure efficient use of energy, a solution needs to model the dynamic nature of ambient energy and the behaviour of TPC while operating under it. This will allow accurate analysis of the application code thus enabling precise placement of trigger calls in the program.

2.4.2 Reactive

To overcome wasted computations/energy in proactive approach, reactive mechanisms employ strategies to trigger checkpointing only when the voltage reaches V_{th}. They either employ hardware or software-timers to trigger checkpointing at V_{th}. Therefore, reactive triggering mechanism can be divided into two subcategories; Interrupt- and Timer-based mechanisms.

Interrupt. Hardware interrupts are an easy way to trigger checkpointing; liberating TPCs from polling the energy buffer at each trigger call [10, 11, 80]. Existing state-of-the-art solutions modify the design of a TPC to integrate voltage comparators. As soon as the current voltage goes below the threshold voltage V_{th}, an interrupt is generated and the device state is checkpointed onto NVM. This allows these strategies to overcome wasted computations/energy of a proactive approach.

The value for V_{th} is set at the time of TPC deployment and is estimated by measuring the energy required to save registers and MCU state. Jayakumar et al. proposed QuickRecall [80]; employing non-volatile main memory thus reducing the threshold voltage as it only needs to save registers and a checkpoint flag. Contrarily, Balsamo et al. proposed Hibernus [11], employing volatile main memory; saving entire program-state (complete RAM along with MCU registers) onto NVM thus requiring higher V_{th}.

Unlike Woude et al. [152], Mathew Hicks proposed Clank [61], to avoid a checkpoint at every idempotence violation. It uses three additional volatile buffers to accumulate writes to all variables, involved in a data dependency. An interrupt is generated when any one of the three buffers overflow in order to write-back data to the persistent memory. This stretches the execution of an idempotent section past its natural limits and reduces the number of checkpoints required by Ratchet [152].

Timer. Interrupt based reactive triggering approaches subtract a significant amount of energy from the energy buffer and is suitable only when the checkpoint size of the device is too small.

Ransford et al. [130] also propose a timer-based approach, in addition to function

and loop based approach, to periodically trigger checkpointing after a fixed interval of time. The timer value is set at the time of deployment and, ideally, it should be equal to the time of capacitor discharge. Mathew Hicks [61] also propose a timer based approach to trigger checkpoint in order to avoid live-lock problem; a scenario which can arise when no single idempotent section is big enough to make the buffer overflow. The proposed approach uses watchdog timers to minimize the gap between two checkpoints in order to avoid the overhead of re-executing the idempotent section thus requiring programmer's effort.

Discussion. One main limitation of an interrupt-based approach is the off-line characterization of the device, required to find the threshold voltage V_{th} for checkpointing and restoring the system state. These systems require user intervention to set the value of V_{th} [11]; a problem in a large-scale deployment.

Balsamo et al. addressed this limitation by proposing Hibernus++ [10]: an adaptive approach to self-calibrate the checkpointing threshold (V_{th}), depending upon the dynamics of energy source and power consumption of the system. This calibration strategy makes the overall system transparent and portable across multiple systems by adapting the voltage threshold at run-time, considering system's power consumption, decoupling capacitance and energy-source's behavior. However, additional energy consumption of an interrupt based reactive triggering mechanism is a major limiting factor as it reduces the energy available for program execution.

To mitigate this additional cost, the second category focuses on timer-based technique to trigger checkpointing. It fires a software interrupt after regular intervals to check the current state of the energy buffer. However, finding the right time interval for a particular program requires repeated execution under varying energy profiles [130]. The rate of energy consumption is different for different programs due to varying complexity of computations. Therefore, it is difficult to set an accurate value at the time of deployment without repeated emulations. This demands manual configuration which has to be repeated after every change in TPC's program thus making it infeasible for perpetual deployment.

2.4.3 Key Properties

After discussing the basic operational details of state-of-the-art triggering mechanisms for checkpointing, we now define key properties to perform a feature comparison between all existing techniques. These properties will help us compare the state-of-the-art techniques and shed light on their benefits and drawbacks.

Each triggering mechanism aim to achieve following properties for efficient utilization of energy buffer.

- **No wasted computations:** All computations performed after the last checkpoint till TPC failure does not become part of any checkpoint, thus, wasting energy. Precise placement of trigger calls, at the point where the TPC is just about to die, can help it reduce these computations which can save energy.

- **No redundant checkpoints:** Accurate energy prediction can help TPCs avoid pessimistic decisions thus reducing the number of redundant checkpoints.

- **Energy prediction model:** Voltage variation is an intrinsic property in TPC operation. Incorporating this behavior at the time of energy prediction plays a key role in achieving accurate energy estimates.

- **No user intervention:** With the vision of perpetual deployment in sight, TPCs must be equipped to achieve all of the above mentioned goals with minimal user/programmer support.

2.4.4 Summary

Table 2.2 shows the summary of the extent to which each trigger mechanism fulfills its goals.

Reactive mechanisms employ hardware modification in existing TPC architecture to trigger the checkpointing exactly at the programmer-defined V_{th} [10, 11, 79]. This liberates the programmer from manual placement of trigger calls in the code [100]. Furthermore, it completely removes wasted computations as it triggers checkpointing

Technique	No Redundant Checkpoints	No Wasted Computations	Energy Prediction Model	No user intervention
Reactive				
Hibernus [11]	✓	✓	-	✗
QuickRecall [80]	✓	✓	-	✗
Hibernus++ [10]	✓	✓	-	✓
Proactive				
Ratchet [152]	✗	✗	-	✓
MementOS [130]	✗	✗	-	✗
HarvOS [18]	✗	✗	Constant	✗
Clank [61]	✓	✗	-	✗
Cleancut [33]	✓	✗	Constant	✓
EPIC [2]	✓	✓	Dynamic	✓

Table 2.2: Feature Comparison. Existing state-of-the-art solutions assume a constant energy model and require user intervention in one way or another. EPIC provides a tool for precise trigger call placement, incorporated with a dynamic energy model.

only when the energy left in the buffer is sufficient for checkpointing. However, the additional hardware constantly compares the energy buffer's voltage with the pre-defined threshold V_{th} to trigger checkpointing. This results in increased energy consumption and dependence on programmer to give a precise estimate of V_{th}; making them unfit for large-scale/perpetual deployments.

We can conclude from our discussion on proactive mechanisms that the choice of different programming constructs, as the criteria to place trigger calls, is not sufficient. Using this approach, existing state-of-the-art proactive triggering mechanisms waste energy on computations that are not part of any checkpoint. The challenge lies in the ability to detect the state of energy buffer at run-time in order to minimize such computations. One way to address this issue is to increase the number of trigger calls placed in the program. This will decrease the number of program instructions between two consecutive calls thereby decreasing wasted computations. However, it will increase the size of program as well as the completion time of the application. Such an approach will only make it difficult to execute long running programs on these resource-constrained devices. Therefore, there is a need for accurate energy

estimation of program's energy consumption in order to place trigger calls before expected power failure, on the given energy budget.

Unfortunately, most of the existing literature for proactive triggering mechanism lack an energy model for predicting program's energy consumption [61, 130, 152]. Those offering a model make an assumption of constant power consumption throughout program's execution; an invalid assumption with highly variable and unpredictable input energy supply.

2.4.5 Qualitative Comparison with EPIC

Capacitor voltage consistently oscillates, between one extreme of operational voltage range to the other, throughout TPC's operation. We observed that energy consumption is not constant and the energy-per-cycle can vary by as much as 5x moving from one extreme of the operational voltage range to the other (Section 4). As a result, this simplistic assumption of constant energy can cause a significant loss of clock cycles to TPCs which keeps on accumulating at each active period e.g, for a TPC running on 10μF capacitor, the loss of number of cycles is approximately 10k in only ten active periods of the device. All computations performed using these clock cycles are wasted and have to be re-performed in the next active epoch. We model this variation of energy consumption and build a tool, EPIC, to allow precise placement of trigger calls in intermittent programs; enabling reduction in wasted computations and more progress of program on the same charge. The detailed design and evaluation of our tool is described in Section 4.

2.5 Non-volatile Hardware Support

Checkpointing is an energy hungry operation as it requires I/O reads and write onto NVM to copy the state of volatile main memory. Previously, we discussed different checkpointing and triggering mechanism with a goal of maximizing the energy available for program execution. Orthogonally, efforts are also being made to design completely new hardware support that removes the need to checkpoint program-state;

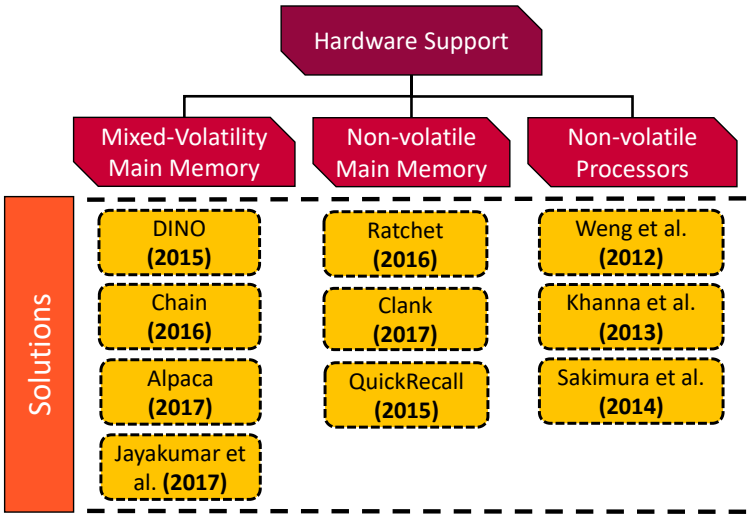


Figure 2-6: Hardware Support. *Three different architectures of transiently-powered computers based on extent of non-volatility of program-state supported.*

making it persistent across periods of energy failure. This will maximize the amount of energy available for program execution thus allowing maximum computational progress.

In this section, we compare the existing state-of-the-art non-volatile hardware designs for removing checkpoints. Based on the extent of non-volatility supported and reduction in checkpoint-size achieved, we have divided existing hardware support into three different categories; Mixed volatility main memory, Fully Non-volatile main memory and Non-volatile processors.

2.5.1 Mixed Volatility Main Memory

The first category of TPCs, ***mixed-volatility systems***, comprise of volatile processor and a hybrid main memory i.e., main memory containing both non-volatile RAM (NVRAM) and volatile RAM; combining the in-memory persistence of NVRAM with fast accesses of RAM. While there are various volatile and non-volatile memory technologies, FRAM and SRAM are the most widely adopted technologies for NVRAM and volatile RAM respectively [67]. With important data residing in NVRAM, mixed-volatility systems significantly reduce the checkpoint size.

These platforms allow the programmer to explicitly define persistent and non-

persistent variables based on application needs. However, increased energy consumption and access latency of NVRAM demands an intelligent placement strategy in order to achieve efficient energy consumption. As programmer is well aware of the memory access patterns of the program, various solutions rely on the programmer for data placement [32, 100, 104].

Jayakumar et al. [81] proposes an approach to find the optimal mapping of code sections in either NVRAM or volatile RAM. It works at the granularity of a function and maps each function's data and/or code to either of the memory, based on where the optimal energy consumption would be achieved. This allows the proposed approach to get benefited from NVRAM's reliability as well as volatile RAM's energy efficiency during code execution.

Discussion. TPCs can be equipped to perform heterogeneous tasks; requiring different program logic in each task. Such behaviour can cause heterogeneous memory-access-patterns thus making it infeasible for a human to devise a energy-efficient data mapping of an application; demanding a different mapping of data on volatile and NVRAM for each task. Additionally, programmers are unaware of the energy consumption dynamics of the underlying hardware at the time of writing the application [2].

On the other hand, solution proposing algorithms for optimal mapping require user support to analyze energy consumption of the underlying hardware. Jayakumar et al. [81] require one-time characterization of the hardware in order to create an optimal mapping of data and code segments. In addition to that, the proposed approach performs poorly when frequently accessed data has to be placed on NVRAM to avoid checkpointing; resulting in increased energy consumption during program execution. Therefore, solutions employing mixed-volatility systems have to make an informed decision about data placement to make efficient utilization of energy budget.

2.5.2 Fully Non-volatile Main Memory

Recent research efforts propose the use of fully non-volatile main memory along with a volatile processor [80, 152]; the second category of TPC's hardware support. This

model eradicates the need to save the state, residing in main memory, thus allowing the device to spend maximum energy on program execution. A TPC, however, still needs to frequently perform light-weight checkpoints of volatile processor state. With persistent program-state on the main memory, intermittent execution of program on a volatile processor gives rise to unique set of challenges; with checkpoint-triggering mechanism playing an important role in conserving energy.

Proactive triggering mechanisms keep on executing the program after saving processor's state, with all computations persisting in the NVRAM across periods of energy failure. As the processor moves forward, the changes in the state of NVRAM are far ahead in time, compared to the checkpointed state of the processor. As a result, the values of variables in NVRAM are from an interrupted, hypothetical future [131]. Re-execution of previously executed program instruction results in inconsistent values of the variables; not possible in a continuous device operation. Therefore, checkpointing NVRAM's state becomes inevitable.

These inconsistencies arise primarily because of the write-after-read data dependencies in the program. Woude et al. [152] and Mathew Hicks [61] trigger a checkpoint after each WAR dependency in the program. Woude et al. uses a proactive mechanism for triggering a checkpoint as it places trigger calls after each WAR dependency in the program; each trigger call acting as a checkpoint call due to small checkpoint size. On the other hand, Mathew Hicks [61] uses three volatile buffers to store all changes, to write back when the buffers overflow.

At the program-state level, Maeng et al. [105] perform liveness analysis of global variables to estimate changes in the program-state. They use code instrumentation to track changes in the non-volatile variables performed after the checkpointing of processor state and are undone after each power interruption thus avoiding idempotence violations. In case of stack, these systems use a differential scheme similar to the one used by Mathew Hicks [61] and Ahmed et al. [3]. However, former uses architecture support [61] whereas the latter uses a volatile stack [3].

In contrast, reactive triggering mechanisms on this platform removes the need to checkpoint the state of NVRAM. QuickRecall [79] uses a hardware comparator to fire

Main Memory	Frequency (MHz)	Access Time (μs)	Current Consumption (mA)	Cycle Energy(nJ)
Non-volatile	16	0.125	1.77	0.66
	8	0.125	1.12	0.42
Volatile	16	0.0625	1.5	0.28
	8	0.125	0.89	0.33

Table 2.3: Micro-measurements for non-volatile and volatile main memory

an interrupt as soon as the voltage reaches V_{th}. This allows QuickRecall to checkpoint state only once in an energy cycle of the system thus avoiding redundant checkpoints. However, the cost of employing voltage comparators only increases the high energy consumption of a NVRAM. For example, access latency and current consumption, for a FRAM used in MSP430FR5969, is almost is 3× and 2× [67] more than that of SRAM respectively. Furthermore, it also limits the TPC to operate within a restricted frequency range i.e., FRAM can only operate up till 8MHz without any delays. For frequencies greater than 8MHz, FRAM uses delay cycles to allow reliable memory operation. Our measurements show that this results in a high energy-per-cycle for a NVRAM based system, making them energy-inefficient(see Table 2.3).

Discussion. It is clear from Table 2.3 that per-cycle-energy for NVRAM based system is significantly higher than that of a fully volatile system. Furthermore, they have to checkpoint the state of NVRAM in order to undo the changes that are not part of any checkpoint thus incurring a high energy cost.

2.5.3 Non-volatile processors

Recent developments in nanotechnology has enabled a new class of embedded systems where the entire processor is designed using a nonvolatile technology, known as Non-Volatile Processors (NVPs).

Wang et al. [154] designed the first NVP chip that uses ferroelectric based technology for the construction of nonvolatile flip-flops (NVFF). This makes the register file, ALU, Timer and other MCU components non-volatile in nature which reduces the sleep and wake-up time with zero standby power for the ambient energy harvesting

devices. This architecture includes a configurable voltage detection system (CVDS) which generates signals for power failure and regain. Sakimura et al. [135] designed an architecture which uses non-volatile magnetic flip-flops (MFFs) to capture the context of MCU thus reducing wake-up time. Khanna et al. [82] propose ferroelectric capacitor-based non-volatile arrays, called NVL arrays, for saving the state before going into sleep mode. The system includes a power management state machine which receives input from power supply detector. It saves the state when the supply is low and restores it when power is regained.

Discussion. The nonvolatile processors (in combination with NVRAM) are the only solution which came near the realization of checkpoint-removal from the life of TPCs. However, a NVP may consume more power than the volatile processor due to the inherently high power consumption of a non-volatile read and write operation; keeping them far from mass production.

2.5.4 Key Properties

After discussing the basic operational details of hardware support for state-retention, we now define key properties which, in our opinion, are essential for the design of an energy-efficient hardware support for state-retention. These properties will help us discuss the pros and cons of existing state-of-the-art techniques.

- **Cost of Checkpoint:** A primary goal of proposing hardware support is to minimize the amount of energy required for application state retention at the time of energy failure. Therefore, a solution must achieve this goal in order to be feasible.

- **Cost of Program Execution:** It also important to minimize the energy consumed during program execution in order to get benefit from the checkpoint size reduction.

- **High Frequency support:** A solution must be able to support higher device frequencies to enable faster program execution.

Technique	Cost of Checkpoint	Cost of Program Execution	Allows Frequency Scaling	Can regulate voltage
Mixed Volatility Main Memory				
DINO [100]	●●●●●	●●●●	✗	✗
Chain [32]	●●●●	●●●●	✗	✗
Alpaca [104]	●●●●	●●●●	✗	✗
Jayakumar et al. [81]	●●●●	●●●●●	✗	✗
Non-volatile Main Memory				
Ratchet [152]	●●●●	●●●●	✗	✗
Clank [61]	●●●	●●●●	✗	✗
Quick Recall [79]	●	●●●●	✗	✗
Non-volatile Processors				
Wang et al. [154]	-	●●●●●	✗	✗
Khanna et al. [82]	-	●●●●●	✗	✗
Sakimura et al. [135]	-	●●●●●	✗	✗
Hardware Support for Volatile Systems				
D^2VFS	●●●●	●●	✓	✓

Table 2.4: Feature Comparison of Hardware Support. Although existing solutions achieve checkpoint size reduction, the gained benefit is lost due to high energy consumption during program execution.

- **Dynamic Voltage Regulation:** As the incoming ambient energy is highly variable, new hardware support must be able to adapt to the changing energy condition to allow better energy utilization.

2.5.5 Summary

Table 2.4 shows the summary of the extent to which each solution proposing non-volatile hardware support and its performance rating on the scale of low(1 point) to high(5 points).

Mixed Volatility Main Memory. Mapping of program-state on volatile and non-volatile requires two different approaches to ensure correct program resumption after each power failure [100]. First, the system needs to checkpoint the volatile state onto NVM. Second, it requires specialized techniques to cope with idempotence violations caused due to intermittent program execution with persistence state. Therefore, the

overhead of employing such a platform is very high(5 points) e.g., Maeng et al. [104] show that it take $\sim 3\times$ more to complete application execution with such an approach.

Colin et al. [32] reduced the energy required to save the non-volatile state by introducing language support; allowing the programmer to write application as a set of atomic tasks which share data through specified memory channels located on the non-volatile memory(4 points). However, it wastes memory as redundant memory channels are created among tasks sharing common data which was later resolved in Alpaca [104].

On the other hand, all solutions using mixed volatility main memory require high cost of program execution due to frequent data accesses residing in NVRAM(4 points). However, Jayakumar et al. [81] has a higher cost of program execution than the rest due to its code migration overhead incurred at the start of every active epoch(5 points).

Non-volatile Main Memory. Hardware designs proposing NVRAM bring significant reduction in checkpoint size. However, the conversion of checkpoint size into overall reduction in checkpointing energy depends on the triggering mechanism employed by the device.

Due to small checkpoint size, Woude et al. [152] uses a proactive approach to place checkpoint calls after each WAR dependency. Since there are high number of WAR dependencies in a program, this approach ends up performing redundant checkpoints, thereby consuming significant amount of energy(4 points). On the other hand, Mathew Hicks [61] avoiding redundant checkpoints by using volatile hardware buffers which are written-back to persistent memory only at the time of buffer overflow(3 points). Jayakumar et al. [79] trigger a checkpoint using a reactive approach and require only MCU registers to be saved thus requiring minimal energy checkpoints(1 point).

In general, solutions employing NVRAM require a higher energy per-clock-cycle and limits the TPC to operate under 8 MHz frequency. This significantly increases the cost of executing program on such hardware support(4 points); rendering the benefit gained by checkpoint size reduction useless.

Non-volatile processors. The goal of complete eradication of checkpoint only materialized with the advent of non-volatile processors. With non-volatile main memory, a TPC employing a non-volatile processor can stop and start its execution anytime under intermittent energy supply without requiring any additional system support. However, this design demands significant amount of energy for program execution in addition to supporting very low frequency for MCUs(5 points).

Comparison in a Nutshell. None of these existing state-of-the-art solutions have been able to devise an energy efficient solution for application-state retention across power failures. While on the one hand, solutions employing mixed volatility systems have to bear code/data migration overhead which becomes a bottleneck in achieving performance gains. On the other hand, solution advocating NVRAM and NVPs require high energy-per-clock cycle for program execution; posing a major hurdle in achieving energy-efficient execution.

Existing system are bound to execute the MCU on a fixed frequency configuration due to limitations of non-volatile technology. Furthermore, all existing hardware support fail to provide voltage regulation mechanism essential for energy-efficient program execution. We observed that MCU frequency increases with the decrease in capacitor voltage in TPCs (See Figure 4-3). This can result in a 5× energy reduction in an operational voltage range [2]. Therefore, configuring the system at the lower end of the operating voltage range can result in least energy consumption per-clock-cycle. However, existing TPC lack necessary hardware support for performing voltage and frequency scaling; required to dynamically adapt to the changing energy conditions in the environment.

2.5.6 Observation

We observed that energy consumed per cycle for higher frequencies is significantly less as compared to that of lower frequencies [66]. This implies configuring the MCU to a higher frequency will not only prove to be more energy-efficient, it will also allow the system to execute clock cycles at faster rate thus allowing improved computational

progress. However, higher frequencies have limited operational range and leave a significant amount of energy in the capacitor as unused.

To address this challenge, we propose a discrete dynamic voltage and frequency scaling technique, D^2VFS, which allows a TPC to dynamically adapt to the changing energy condition of the environment. D^2VFS proposes a set of voltage detectors and regulators to be incorporated in the design of TPC thus enabling voltage regulation at discrete levels; an important step in minimizing energy-per-cycle at a particular frequency. The system starts at the maximum frequency supported by the MCU thus consuming minimum energy per clock cycle. As the system consumes energy, capacitor voltage decreases, therefore, D^2VFS dynamically adapts the system to the best frequency and regulated-voltage configuration at any given capacitor-voltage. We argue that it is still possible to ensure maximum computational progress on a volatile system, along with checkpoints, than the one achieved using a non-volatile hardware support. The detailed evaluation and design of our tool is described in Section 5.

2.6 Conclusion

In this chapter, we presented state-of-the-art solutions discussing checkpointing strategies, their triggering mechanisms, and hardware support for application state retention in TPCs. We also presented taxonomy of solutions in all three categories while proving a qualitative comparison with the contributions of this thesis.

Existing checkpointing approaches waste a lot of energy either by estimating large update size for checkpoint or perform excessive NVM accesses for computing checkpoint differentials. As a result, they demand high energy for checkpointing than what is required to save the actual change in application state. DICE provide an energy-efficient approach which performs differential checkpointing while removing excessive NVM accesses and reducing computational overhead. It uses an in-memory data structure to track checkpoint differentials which plays a key role in efficient tracking.

Triggering mechanisms help TPCs to trigger checkpointing at the right time. These mechanisms work either in a proactive manner, placing trigger calls in the

program, or in a reactive manner forcing the system to take checkpoint when the voltage reaches V_{th}. Each category demands precise analysis of programs for both function call placement or setting the value of V_{th}. We observed that variation of input voltage causes a significant variation in the energy consumption of the MCU. Most existing solutions lack an energy model in the first place. EPICcaptures dynamics associated with capacitor voltage variation in order to predict energy consumption of the program. In this way, it not only allows the system to automatically place trigger calls at the right place but it also helps the programmer in deciding the right value for V_{th}.

With the aim of enabling maximum progress in an active period, different categories of hardware support introduces different levels of non-volatility in the TPC architecture; significantly reducing checkpoint size and shifting energy budget for performing computations. However, the introduction of non-volatility causes marked increase in the energy consumption during program execution. Properties of each category reveal that the benefit gained by eradicating checkpoints gets overshadowed by the increased energy consumption during program execution. Furthermore, it limits the TPC architecture to execute at lower frequencies thus consuming higher energy per clock cycle. We argue that volatile systems are key to achieving high performance during program execution. We propose dynamic voltage and frequency scaling mechanism, D^2VFS, which helps the device to dynamically shift frequency and voltage as per the incoming energy thus enabling faster execution of programs. In the following chapters, we discuss the design and implementation details of DICE, EPIC and D^2VFS in detail.

Chapter 3

Differential Checkpointing

After introducing the problem space and establishing a formal discussion background in the previous chapters, we now turn our focus towards the actual contributions of this dissertation. We begin with our first contribution, i.e., differential checkpoint for reducing the checkpoint size for transiently powered devices, DICE, allowing them to efficiently utilize their energy budget.

In the previous chapter, we learned that existing state-of-the-art checkpointing solutions in intermittent computing lack the ability to find the differentials between current program state and the checkpointed state. This forces them to perform large number of NVM accesses to update the checkpoint on the NVM. Solutions which employ differential checkpointing technique either have to compute the hash of the main memory content [8] or require a byte-by-byte comparison [16], both resulting in a high computational overhead. Therefore, all existing solution have to waste a significant amount of energy on the checkpointing process; leaving a very small amount for program execution.

We argue that it possible to have an efficient differential checkpointing technique that performs all operations in-memory thus minimizing NVM accesses. By having an efficient data structure to record checkpoint differentials, we show that it is possible to significantly reduce the computational overhead incurred by existing approaches in computing the checkpoint differential.

Based on 107,000+ data points, and compared with existing solutions on the same

workload, our results indicate that the reduction in NVM operations enabled by DICE allows the system to shift part of the energy budget towards useful computations. This reflects into 97% fewer checkpoints, which is a direct effect of DICE's ability to use a given energy budget for computing rather than checkpointing. It also enables up to one order of magnitude shorter completion time, increasing system's responsiveness and despite the overhead of code instrumentation. By enabling smaller checkpoints, DICE reduces the size of smallest energy buffer by up to 88% required to run an application on TPC thus enabling smaller device footprints.

The rest of this chapter is structured as follows. Firstly, we motivate the problem space and discuss related work in Sections 3.1 and 3.2, respectively. We then analyze and give a qualitative comparison of our work with existing differential techniques in Section 3.3. From this, we derive the design choices of our proposed technique in Section 3.4 and 3.5 respectively. We explain our implementation details in Section 3.6. Finally, we show the viability of our design and present evaluation of our approach in Section 3.7 and 3.8 before we summarize the discussion in Section 3.9.

3.1 Motivation

System support exists to enable intermittent computing, employing a form of *checkpointing* to let the program cross periods of energy unavailability [11, 130]. This consists in replicating the application state over non-volatile memory (NVM) in anticipation of power failures, where it is retrieved back once the system resumes with sufficient energy.

Due to the characteristics of NVM, checkpoints are extremely costly in energy and time. When using flash memories, for example, the energy cost is orders of magnitude larger than most system operations [16, 41]. FRAM improves these figures; still, checkpoints often represent the dominating factor in an application's energy and time profile [11, 18]. As the cost of checkpoint is subtracted from the energy for useful computations, taming this overhead is crucial.

To that end, differential techniques are commonly employed when providing fault

tolerance in operating systems [39, 90, 120] and when maintaining consistency in distributed databases and transactions [83, 91, 127, 140]. While some of these are simply not applicable in embedded systems due to lack of requisite hardware support, we observe that the energy cost of existing software-based approaches is often not worth the benefit on energy constrained platforms. Therefore, newer techniques must be established to reap the benefits of differential checkpointing in intermittently-powered systems.

3.1.1 Requirements

The requirements and challenges of estimating the checkpoint size in intermittent computing domain are substantially different from the main stream computing. While main-stream computing assume un-interrupted power, transiently powered systems have variable and unpredictable energy supply. This gives rise to following set of questions for a checkpointing strategy.

- What is the size of checkpoint at any point during execution? The size of checkpoint differential changes with the point of power interruption. Therefore, knowledge of where a checkpoint takes place influences what differentials need to be considered, how to track them, and how to configure the system parameters triggering a checkpoint.

- How can we reduce the computational overhead of computing checkpoint differentials? Along with checkpoint size reduction, it is equally important to reduce the computational overhead caused by tracking the checkpoint differentials in order for the checkpointing strategy to be feasible.

- How can we develop a flexible design that can work with existing systems?. As discussed in Section 2.4, there are two types of triggering mechanisms in transiently-powered systems. Both of them demanding different strategies to cope with checkpoint size reduction. Therefore, designing DICE as a plug-in complement to existing system support.

None of these three questions, which we ask here, are answered well by existing checkpointing strategies thus resulting in energy-inefficient solutions.

3.1.2 Major Contributions

To cater for the specific challenges of intermittently-powered devices, we design DICE (DIfferential ChEckpointing), a system that efficiently evaluates the differential between the previous checkpoint data and the volatile application state. DICE uses this information to limit the checkpoint operation to a slice of NVM data, namely, the one corresponding to changed application state. The fundamental contribution of DICE rests in identifying an efficient design point that allows the notion of differential checkpointing to be profitable in intermittently-powered systems, as elaborated in Section 3.4.

To that end, the design of DICE integrates three contributions:

1) unlike previous attempts [8, 16] that access NVM to compute differentials, DICE maintains differential information *only in main memory* and access to NVM is limited to updating existing checkpoint data; we achieve this through an automatic code instrumentation step.
2) DICE capitalizes on the different memory write patterns by employing *different techniques* to track changes in long-lived global variables as opposed to short-lived variables local to functions; the code instrumentation step identifies these patterns and accordingly selects the most appropriate tracking technique.
3) in the absence of hardware support to track changes in main memory, which is too energy-hungry for intermittently-powered devices, our design is entirely implemented in software and ensures *functional correctness* by prudently opting for worst-case assumptions in tracking memory changes; we demonstrate, however, that such a choice is not detrimental to performance.

3.2 Related Work

We elaborate on how intermittent computing shapes the problem we tackle in unseen ways; then proceed with discussing relevant works in this area.

3.2.1 Mainstream Computing

The performance trade-offs in mainstream computing are generally different compared to ours. Energy is not a concern, whereas execution speed is key, being it a function of stable storage operations or message exchanges on a network. Systems are thus optimized to perform as fast as possible, not to save energy by reducing NVM operations, as we do.

Differential checkpointing for multi-processing OSes and virtual machines exist. Here, checkpoints are mainly used for fault tolerance and load balancing. Systems use specialized hardware support to compute differentials [39, 120, 124], including memory management units (MMUs), which would be too energy-hungry for intermittently-powered devices. Moreover, updates happen at page granularity, say 4 KBytes. This is a tiny fraction of main memory in a mainstream computing system, but a large chunk of it in an intermittently-powered one, thus motivating different techniques.

Checkpointing in databases and distributed systems [83, 115, 129] is different in nature. Here, checkpoints are used to ensure consistency across data replicas and against concurrently-running transactions. Moreover, differential checkpointing does not require any tracking of changes in application state, neither in hardware nor in software, because the data to be checkpointed is explicitly provided by the application.

Differential checkpoints are also investigated in autonomic systems to create self-healing software. Enabling this behavior requires language facilities rarely available in embedded systems, let apart intermittently-powered ones. For example, Fuad et al. [44] rely on Java reflection, whereas Java is generally too heavyweight for intermittently-powered devices.

Our compile-time approach shares some of the design rationale with that of Netzer and Weaver [117], who however target debugging long-running programs, which is a

different problem. Further, our techniques are thought to benefit from the properties of proactive checkpointing and to ensure correctness despite uncertainty in checkpoint times in reactive checkpointing. We apply distinct criteria to record differentials depending on different memory segments, including the ability of allowing cross-frame references along an arbitrary nesting of function calls.

3.2.2 Intermittent Computing

We can effectively divide the literature in three classes, depending on device architectures.

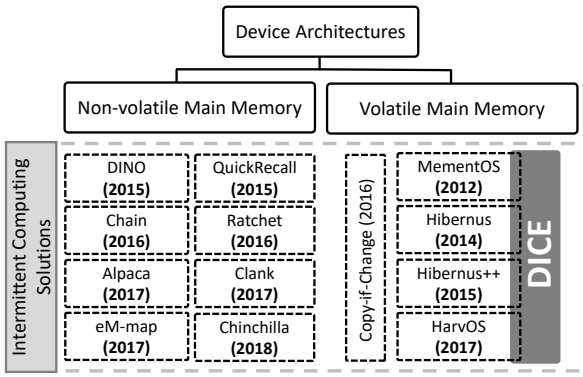


Figure 3-1: Intermittent computing solutions. *DICE targets traditional device architectures with volatile main memory and seamlessly integrates with all the corresponding solutions.*

Non-volatile main memories. As shown in Figure 3-1, solutions exist that target device architectures that employ non-volatile processors [92, 148, 155] or non-volatile main memory [56], normally FRAM. The former relieve the system from checkpoints altogether, yet require dedicated processor designs still far from massive production. Device employing non-volatile main memories trade increased energy consumption and slower memory access for persistence [56]. When using FRAM as main memory with MSP430, for example, energy consumption increases by 2-3× and the device may only operate up to half of the maximum clock frequency [99].

The persistence brought by non-volatile main memory also creates data consistency issues due to repeated execution of non-idempotent operations that could lead to incorrect executions. Solutions exist that tackle this problem through specialized compilers [152] or dedicated programming abstractions [32, 101, 104]. The former may add up to 60% run-time overhead, whereas the latter require programmers to learn new language constructs, possibly slowing down adoption. An open research

question is what are the conditions—for example, in terms of energy provisioning patterns—where the trade-off exposed by these platforms play favorably.

NVM for checkpoints. We target devices with volatile main memories and external NVM facilities for checkpoints [65,94,106,119]. Existing literature in this area focuses on striking a trade-off between postponing the checkpoint as long as possible; for example, in the hope the environment provisions new energy, and anticipating the checkpoint to ensure sufficient energy is available to complete it.

Hibernus [11] and Hibernus++ [10] employ specialized hardware support to monitor the energy left. Whenever it falls below a threshold, both systems *react* by firing an interrupt that preempts the application and forces the system to take a checkpoint. Checkpoints may thus take place at *any* arbitrary point in time. Both systems copy the entire memory area—including unused or empty portions—onto NVM. We call this strategy *copy-all*.

MementOS [130] and HarvOS [18] employ compile-time strategies to insert specialized function calls to check the energy buffer. Checkpoints happen *proactively* and *only* whenever the execution reaches one of these calls. During a checkpoint, every *used* segment in main memory is copied to NVM regardless of changes since the last checkpoint. We call such a strategy *copy-used*.

Improving checkpoints. Unlike our approach of proactively tracking changes in application state, solutions exist that evaluate the differential at checkpoint time; either via hash comparisons [8] or by comparing main memory against a word-by-word sweep of the checkpoint data on NVM [16]. We call these approaches *copy-if-change*.

Note, however, that these systems are *fundamentally incompatible* with both the *reactive* and the *proactive* checkpointing systems that DICE aims to complement. The fundamental limitation is the inability to determine the energy cost of a checkpoint a priori, which is mandatory to decide *when* to trigger a checkpoint and is necessary input to all of the aforementioned systems. DICE provides an estimate of the *actual* cost of a checkpoint at any moment in execution, allowing to dynamically update system parameters triggering a checkpoint.

Compared to *copy-if-change*, DICE also minimizes accesses to NVM. We achieve

this through a specialized code instrumentation step. This inserts functionality to track changes in application state that *exclusively* operates in main memory. Operations on NVM are thus limited to updating the relevant blocks when checkpointing, with no additional pre-processing or bookkeeping required. The checkpoint data is then ready to be reloaded when computation resumes, with no further elaboration.

3.3 Scrutinizing Differential Checkpointing Techniques

Checkpointing system state is widely used to achieve fault tolerance and data consistency in mainstream computing. For example, OS checkpoints are used to recover the system to a stable state after a faulty update [50], distributed system checkpoints are used to combat communication and node failures [150], and database checkpoints are used to maintain a backup at a known good point before applying changes from the log [91]. Although these systems typically do not suffer from limitation of resources such as computation, memory and power, evaluating state differentials is still preferred to spend minimum of these useful resources during checkpoint operations, which restrain the system from doing useful work.

Before embarking on our journey to develop a differential checkpointing solution for resource-constrained embedded devices, we want to scrutinize the existing work for making informed design choices. We discard solutions that rely on specialized hardware support because of their high power requirements, which is prohibitive in energy-constrained platforms. For instance, processors with MMUs expose the state differentials via *dirty-bits* in page tables, clearly specifying the pages altered in a process address space. This information can be exploited to incrementally checkpoint the process state. However, such information is not accessible in embedded MCUs, which usually lack MMU support due to energy limitations [29]. *Software-only* solutions for differential checkpointing also exist that may provide a plausible lead into resolving this limitation of embedded MCUs.

We categorize existing software-only approaches in three broad categories and

analyze their suitability for resource constrained platforms, specifically in the context of intermittently-powered ones.

3.3.1 Hashing

A hashing based approach computes hashes over memory chunks of equal sizes [1, 8, 43]. Since the hash is a function of memory content, a change in memory content of a specific region modifies its hash value. At the time of checkpoint, the current and previous hash values of the target memory region are compared. If these values are different, the corresponding checkpoint in secondary storage is updated along with the newly computed hash. An unmodified hash value indicates that the memory region has not been updated since the last checkpoint, and hence ignored in the interest of reducing expensive write operations in the secondary storage. Multidimensional hashes [46] can also be used to group multiple memory chunks and representing them with a single hash value, which is a function of the hash values of each individual chunk.

Although hashing is a compute-intensive operation, this approach has still proven to be profitable in main stream computing as it avoids checkpointing significant portions of a typically very large main memory. Whether or not this tradeoff plays favorably in embedded systems, remains to be seen.

3.3.2 Tracking

A dynamic, tracking based approach proactively tracks write accesses in main memory. This is often achieved through specialized function calls instrumented in the code at compile time [117]. These function calls capture the memory addresses being modified and records them in a separate data structure. At the time of checkpoint, only the content of recorded memory addresses are updated in secondary storage. If a fault occurs, the last stored checkpoint is used to roll back the process state. This approach is particularly useful for long running programs as it avoids re-execution of the entire program by rolling the state back to the nearest known good point.

Deciding what and when to track plays a key role in defining the granularity of

differential analysis and, thus, the number of instructions required to be re-executed to reach the original state. A fine-grained analysis requires fewer instructions to be replayed to reach the correct program state at the cost of high run-time overhead. On the contrary, a coarse-grained analysis incurs smaller run-time overhead but increases the re-execution overhead. Thus, the frequency of faults can be a defining factor in determining the tracking granularity.

Tracking changes in memory is apparently less compute-intensive than computing hashes. However, memory access patterns of an application may be crucial in establishing the feasibility of this approach in embedded systems. For example, in applications with frequent memory access operations, the tracking overhead may eventually surpass the benefits accrued from smaller checkpoint data.

3.3.3 Static analysis

To mitigate the run-time cost of tracking approaches, static analysis can be used to identify variables that are modified by different program execution paths [20, 22, 149]. A key-value data structure is used to create a mapping between different paths and their variables, where the key is the unique path identity and the value is all the variables modified on that path. The run-time only needs to track the execution of paths, and checkpoint all the variables on the executed paths obtained from the key-value data structure.

This approach reduces energy and computational costs as tracking is limited to coarse-grained program paths instead of per variable. However, the memory overhead may increase exponentially for applications with large number of paths, resulting in large key-value stores that record the complete addresses of variables. This approach may be improved by modifying compilers to intelligently allocate main memory [142, 153], and group variables that are modified on a single path. As a result, the key-value data structure will only need to store the start and end address of adjacently allocated group of variables. However, since variables may be modified on multiple paths, such allocation may not be possible in some applications.

Although this approach is attractive due to its negligible computational require-

ments at run-time, the memory overhead is still a matter of concern on platforms with limited memory capacities.

3.3.4 Comparative Analysis

As an impetus to our quest for developing a differential state-checkpointing solution for resource-constrained devices, we conduct a "feeler" analysis of these software-based techniques. The ultimate goal of this analysis is to ascertain at a macro level whether a full-fledged implementation of a particular approach is worth the effort for embedded devices. We first elaborate on the experimental settings used for this analysis followed by a one-on-one comparison between different techniques.

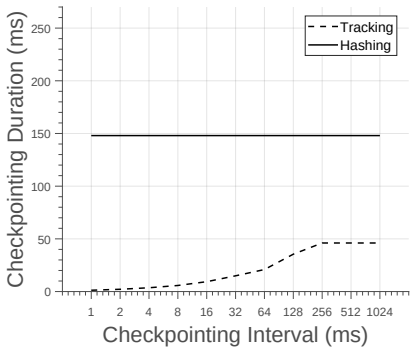


Figure 3-2: Computational overhead of differential checkpointing techniques in FFT application.

Settings. We consider two applications, fast fourier transform (FFT) and activity recognition (AR), widely used as benchmarks in intermittent computing [49, 51]. Both these applications possess useful characteristics needed for our analysis. For example, FFT is a signal processing algorithm that frequently accesses variables in main memory. AR provides large number of paths needed to evaluate the performance of static analysis technique described in Section 3.3.3.

We execute these applications on an MSP430 platform, which is widely considered as a defacto standard in energy-harvesting devices [137]. We are interested in different metrics for different approaches: Since hashing is compute-intensive, we are interested in its *computational* overhead, that is, the time needed to compute and compare hashes of different memory regions and in performing the differential checkpoint operation. Differently, static analysis has a negligible computational overhead but we need to analyze its *memory* requirements for saving the key-value data structure, and whether such requirements can be borne by memory-constrained systems.

Finally, the tracking based approach effects both: it performs additional computations to track memory accesses as well as needs a data structure to record modified memory addresses.

For hashing, we use SHA-1 over memory regions of size 256 bytes. Thus, the entire 10 KB memory is divided in 40 hash regions. The static analysis is performed using the LLVM compiler toolchain to establish a mapping between all program paths and the variables modified by each one of them. We then augment the application with a key-value data structure representing this information at run-time for differential checkpointing. For tracking, we use the code instrumentation technique similar to the one in [117] along with a bit-array data structure to track modifications in memory, where each bit represents one byte (or word) in main memory [3].

Computational overhead. Figure 3-2 shows the additional time taken by checkpointing operations in FFT application. We omit the static analysis approach from this comparison because of its negligible run-time overhead.

Approach	**Memory (bytes)**
Static analysis	2240
Tracking	150

Table 3.1: Memory overhead of different techniques in AR application

For hashing, this overhead is defined by the time taken for computing and comparing fresh hash values with the previous hash values retrieved from the secondary storage, and updating the checkpoint with modified memory regions. Regardless of the checkpoint interval, hashing based approach has an almost constant but high computational overhead, which is dominated by hash computations.

On the other hand, the tracking based approach has a variable but low computational overhead. Tracking is comparatively a lighter operation as it only needs a few cycles to update the in-memory data structure for recording modified memory locations. Here the computational overhead is dominated by the checkpoint updates in secondary storage. This overhead understandably increases with checkpointing interval, as more variables are likely modified when the program executes for longer durations due to fewer checkpoint interruptions in between.

Memory overhead. Table 3.1 shows the additional memory required by different techniques in AR application. This includes the in-memory data structure for recording addresses of modified memory locations and differentially updating the checkpoint with them. The hashing based approach is omitted because it only temporarily occupies memory to compute and compare hash values during checkpoint operations. These hash values are flushed out in secondary storage at the time of checkpoint, and hence, no additional memory is consumed during normal program execution.

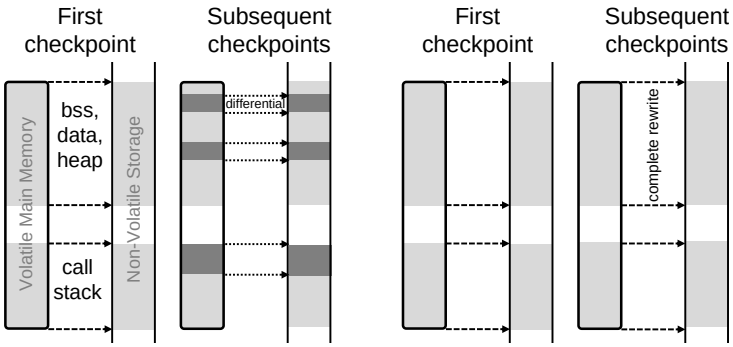
We can see that the static analysis based technique results in unbearable memory overhead: 16× greater than the actual size of application. This overhead is, in principle, a function of the number of paths in a program. The more the number of paths in a program, the more the number of key-value data structures needed to keep path-to-variables mapping.

The tracking based approach performs well in terms of memory overhead as well. The use of bit-array results in restricting the memory overhead of the tracking based approach to at most one-eighth of the memory size.

Summary. Our feeler analysis provides useful observations that can guide our pursuit of a differential checkpointing solution for energy-harvesting devices.

Although the hashing based approach has negligible memory overhead, its heavy computational requirements far exceed the limitations of energy-harvesting devices typically equipped with low-end MCUs. Similarly, the static analysis based approach addresses these limitations of hashing, yet the memory requirements are inconceivable for memory-constrained MCUs. The tracking based approach evidently gets much closer to the sweet spot in this cost benefit spectrum: it efficiently controls both the computational and memory overhead needed to evaluate memory differentials for efficient checkpointing.

While taking note of this feeler analysis, we next present our techniques for differential checkpointing in energy-harvesting embedded systems, whose foundations are firmly held in a tracking based approach. However, the specific constraints of these systems demand specialized tracking techniques based on various memory access patterns and the regions where they occur.



(a) DICE (b) Existing systems

Figure 3-3: DICE fundamental operation. *DICE updates checkpoint data based on differentials at variable level in the global context, or with modified stack frames.*

3.4 DICE In A Nutshell

Figure 3-3 describes the fundamental operation of our approach, DICE. Once an initial checkpoint is available, DICE tracks changes in main memory to only update the affected slices of the existing checkpoint data, as shown in Figure 3-3(a). We detail such a process, which we call *recording differentials*, in Section 3.5.

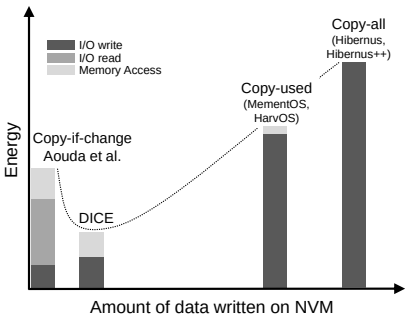


Figure 3-4: Qualitative comparison of the checkpoint techniques. Copy-all *has highest energy costs due to maximum writes on NVM.* Copy-used *avoids copying unused memory areas, reducing the energy cost.* Copy-if-change *further reduces energy costs, using NVM reads to compute differentials. DICE records changes in the stack at frame granularity, but only operates in main memory.*

Differentials. We apply different criteria to determine the granularity for recording differentials. The patterns of reads and writes, in fact, are typically distinct depending on the memory segment [84]. We individually record modifications in global context, including the **`bss`**, **`data`**, and **`heap`** segments. Such a choice minimizes the size of the update for these segments on checkpoint data. Differently, we record modifications in the call stack at frame granularity. Local variables of a function are likely frequently updated during a function's execution. Their lifetime is also the same: they are allocated when creating the frame, and collectively lost once the function returns. Because of

this, recording differentials at frame-level amortizes the overhead for variables whose differentials are likely recorded together.

A dedicated *precompiler* instruments the code to record both kinds of differentials. For global context, we insert DICE code to populate an *in-memory* data structure with information about modified memory areas. Writes to global variables can be statically identified, while indirect writes via pointers have to be dynamically determined. Therefore, we instrument direct writes to global variables and *all* indirect writes in the memory via pointer dereferencing. The precompiler also instruments the code to record differentials in the call stack by tracking the changes to the base pointer.

At the time of checkpointing, the in-memory data structures contain sufficient information to identify what slices of NVM data require an update. Unlike existing solutions [8,16], this means that a checkpoint operation only accesses NVM to perform the actual updates to checkpoint data, whereas any other processing happens in main memory.

Our approach is sound but pessimistic, as we overestimate differentials. Our instrumentation is non-obtrusive as it only reads program state and records updates in a secluded memory region that will not be accessed by a well-behaved program. Similarly, an interrupted execution will be identical to an uninterrupted one because a superset of the differential is captured at the checkpoint and the entire program state is restored to resume execution. The differential is correctly captured because the grammar of the target language allows us to identify all direct or indirect (via pointers) memory writes. Any writes introduced by the compiler, such as register spilling, are placed on the current stack frame that is always captured, as described in Section 3.5.

DICE and the rest. Reducing NVM operations is the key to DICE performance. Figure 3-4 qualitatively compares the energy performance of checkpointing solutions discussed thus far.

Hibernus [11] and Hibernus++ [10] lie at the top right with their *copy-all* strategy. The amount of data written to NVM is maximum, as it corresponds to the entire

memory space regardless of occupation. Both perform no read operations from NVM during checkpoint, and essentially no operation in main memory. MementOS [130] and HarvOS [18] write fewer data on NVM during checkpoint, as their *copy-used* strategy only copies the occupied portions. To that end, they need to keep track of a handful of information, such as stack pointers, adding minimal processing in main memory.

The *copy-if-change* [16] strategy lies at the other extreme. Because of the comparison between the current memory state and the last checkpoint data, the amount of data written to NVM is reduced. Performing such comparison, however, requires to sweep the entire checkpoint data on NVM, resulting in a high number of NVM reads. Because write operations on NVM tend to be more energy-hungry than reads [110], the overall energy overhead is still reduced.

In contrast, DICE writes slightly more data to NVM compared to existing differential techniques, because modifications in the call stack are recorded at frame granularity. However, recording differentials only require operations in main memory and no NVM reads. As main memory is significantly more energy efficient than NVM, energy performance improves. Section 3.7 and Section 3.8 offer quantitative evidence.

3.5 Recording Differentials

We describe how we record differentials in global context, how we identify modified stack frames, and how we handle pointer dereferencing efficiently, while maintaining correctness. The description is based on a C-like language, as it is common for resource-constrained embedded platforms. Note our techniques work based on a well-specified grammar of the target language. We cannot instrument platform-specific inline assembly code, yet its use is extremely limited as it breaks cross-platform compatibility [49].

3.5.1 Global Context

DICE maintains a data structure in main memory, called *modification record*, to record differentials in global context. It is updated as a result of the execution of a

record() primitive the DICE precompiler inserts when detecting a potential change to global context. The modification records are not part of checkpoint data.

Figure 3-5 shows an example. The **record()** primitive simply takes as input a memory address and the number of bytes allocated to the corresponding data type. This information is sufficient to understand that the corresponding slice of the checkpoint data is to be updated. How to inline the call to **record()** depends on the underlying system support.

```
int var,*ptr; //global variables
...
record(&var,sizeof(var));
var++;
...
record(&ptr,sizeof(*ptr));
ptr = &var;
```

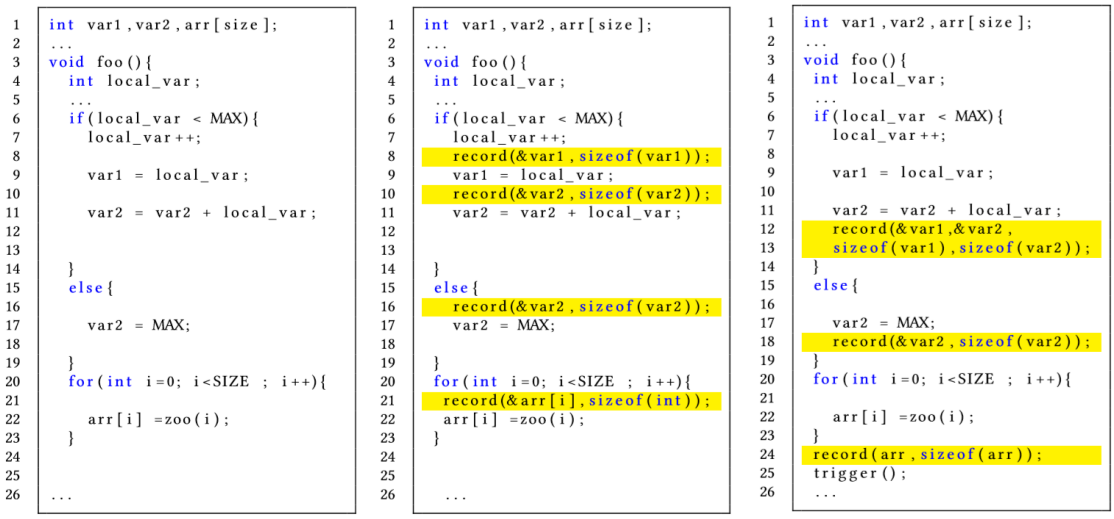
Figure 3-5: Example instrumented code.

Reactive systems. In Hibernus [10, 11], an interrupt may preempt the execution at any time to take a checkpoint. This creates a potential issue with the placement of **record()**.

If the call to **record()** is placed right after the statement modifying global context and the system triggers a checkpoint right after such a statement, but before executing **record()**, the modification record includes no information on the latest change. The remedy would be atomic execution of the statement changing data in global context and **record()**; for example, by disabling interrupts. With systems such as Hibernus [10, 11], however, this may delay or miss the execution of critical functionality.

Because of this, we choose to place calls to **record()** right before the relevant program statements, as shown in Figure 3-6(b). This ensures that the modification record is pessimistically updated before the actual change in global context. If a checkpoint happens right after **record()**, the modification record might tag a variable as updated when it was not. This causes an unnecessary update of checkpoint data, but ensures correctness.

If a checkpoint happens right after **record()**, however, the following statement is executed first when resuming from checkpointed state. The corresponding changes are not tracked in the next checkpoint, as **record()** already executed before. We



(a) Before instrumentation

(b) After instrumentation (reactive)

(c) After instrumentation (proactive)

Figure 3-6: Example instrumentation for reactive or proactive checkpoints. *With reactive checkpoints, each statement possibly changing global context data is preceded by a call to* **`record()`**. *With proactive checkpoints, code locations where a checkpoint may take place are known, so calls to* **`record()`** *may be aggregated to reduce overhead.*

handle this by re-including in the next checkpoint the memory region reported in the most recent **`record()`** call. We prefer this minor additional overhead for these corner cases, rather than atomic executions.

Proactive systems. MementOS [130] and HarvOS [18] insert specialized function calls called trigger calls or **`trigger`**s in the code. Based on the state of the energy buffer, the triggers decide whether to checkpoint before continuing. This approach exposes the code to further optimizations.

As an example, Figure 3-6(c) shows the same code as Figure 3-6(a) instrumented for a proactive system. For segments without loops, we may aggregate updates to the modification record at the *basic block* level or just before the call to **`trigger()`**, whichever comes first[1]. The former is shown in line 8 to 14, where however we cannot postpone the call to **`record()`** any further, as branching statements determine only at run-time what basic block is executed.

In the case of loops over contiguous memory areas, further optimizations are possible. Consider lines 20 to 23 in Figure 3-6: a call to **`record()`** inside the loop body,

[1]The aggregation of updates does not apply to pointers: if a single pointer modifies multiple memory locations, only the last one will be recorded.

necessary in Figure 3-6(b) for every iteration of the loop, may now be replaced with a single call before the call to `trigger()`. This allows DICE to record modifications in the whole data structure at once, as shown in Figure 3-6(c) line 24.

Certain peculiarities of this technique warrant careful consideration. For instance, loops may, in turn, contain branching statements. This may lead to false positives in the modification record, which would result in an overestimation of differentials. Fine-grained optimizations may be possible in these cases, which however would require to increase the complexity of instrumentation and/or to ask for programmer intervention. We opt for a conservative approach: we record modifications on the entire memory area that is *possibly*, but not definitely modified inside the loop.

3.5.2 Call Stack

Unlike data in global context, we record differentials of variables local to a function at frame level, as these variables are often modified together and their lifetime is the same. To this end, DICE monitors the growth and shrinking of the stack without relying on architecture support as in Clank [61].

Normally, *base pointer* (BP) points to the base of the frame of the currently executing function, whereas the *stack pointer* (SP) points to the top of the stack. DICE only requires an additional pointer, called the *stack tracker* (ST), used to track changes in BP between checkpoints. We proceed according to the following four rules:

R1: ST is initialized to BP every time the system resumes from the last checkpoint, or at startup;

R2: ST is unchanged as long as the current or additional functions are executed, that is, ST does *not* follow BP as the stack grows;

R3: whenever a function returns that possibly causes BP to point deeper in the stack than ST, we set ST equal to BP, that is, ST follows BP as the stack shrinks;

R4: at the time of checkpoint, we save the memory region between ST and SP, as this corresponds to the frames possibly changed since the last checkpoint.

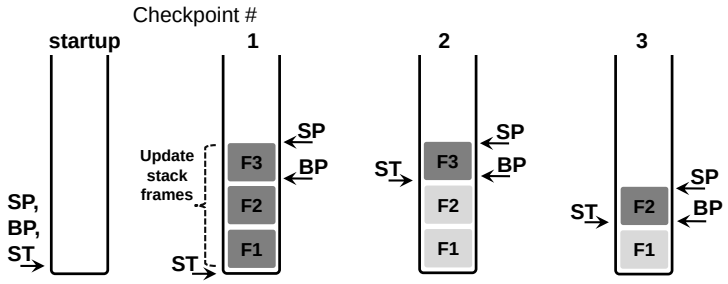


Figure 3-7: Identifying possibly modified stack frames. *The stack tracker (ST) is reset to the base pointer (BP) when the system resumes or at startup. ST does not follow BP as the stack grows, but it does so as the stack shrinks. The dark grey region between ST and the stack pointer (SP) is possibly modified.*

Figure 3-7 depicts an example. Say the system is starting with an empty stack. Therefore, ST, SP, and BP point to the base of the stack as per **R1**. Three nested function calls are executed. While executing `F3`, BP points to the base of the corresponding frame. Say a checkpoint happens at this time, as shown under checkpoint #1 in Figure 3-7: the memory region between ST and SP is considered as a differential since the initial situation, due to **R4**.

When resuming from checkpoint #1, ST is equal to BP because of **R1**. Function `F3` continues its execution; no new functions are called and no functions return. According to **R2**, ST and BP remain unaltered. The next checkpoint happens at this time. As shown for checkpoint #2 in Figure 3-7, **R4** indicates that the memory region to consider as a differential for updating the checkpoint corresponds to the frame of function `F3`. In fact, the execution of `F3` might still alter local variables, requiring an update of checkpoint data.

When the system resumes from checkpoint #2, `F3` returns. Because of **R3**, ST is updated to point to the base of the stack frame of `F2`. If a checkpoint happens at this time, as shown under checkpoint #3 in Figure 3-7, **R4** indicates the stack frame of function `F2` to be the differential to update. This is necessary, as local variables in `F2` might have changed once `F3` returns control to `F2` and the execution proceeds within `F2`.

Note that the efficiency of recording differentials at frame level also depends on programming style. If function calls are often nested, the benefits brought by this technique likely amplify compared to tracking individual local variables. Similarly,

multiple frames may enter and exit the stack without checkpointing in between. In such cases, tracking individual local variables may introduce redundant overhead.

3.5.3 Pointer Dereferencing

Special care is required when tracking changes in main memory through dereferencing pointers. We use a separate **record_p()** primitive to handle this case.

```
void foo(int a){
    int var = a,*ptr; //local variables
    ...
    ptr = &var;
    record_p(ptr,sizeof(*ptr));
    *ptr= a+5; //current stack frame
    ...        //modification
    bar(&var);
    ...
}
void bar(int *lptr){
    int a = 5; //local variables
    ...
    record_p(lptr,sizeof(*lptr));
    *lptr = a + 5; //previous stack frame
    ...            //modification
}
```

Figure 3-8: Example instrumented code to record local variables that are passed by reference.

With **record_p()**, we check if the pointer is currently accessing the global context (i.e., a global scalar or heap) or a local variable inside a stack frame. In the former case, the modification record is updated as described in Section 3.5.1. Otherwise, there are two possibilities depending on whether the memory address pointed to lies between ST and SP. If so, the corresponding change is already considered as part of the checkpoint updates, as per **R4** above. Otherwise, we find ourselves in a case like Figure 3-8 and update ST to include the frame being accessed. As a result, we include the memory changes in the update to existing checkpoint data at the next checkpoint, as per **R4** above. This ensures correctness of our approach even if local variables are passed by reference along an arbitrary nesting of function calls or when using recursion.

3.6 Implementation

We describe a few implementation highlights for DICE, which are instrumental to understand our performance results.

Precompiler. We implement the DICE precompiler targeting the C language using ANTLR [121]. The precompiler instruments the entire code, including the run-time libraries, for recording modifications in the global context as described in Section 3.5.1,

depending on the underlying system support, and for identifying modified regions of the stack, as explained in Section 3.5.2.

As a result of this instrumentation, DICE captures modifications in main memory except for those caused by peripherals through direct memory access (DMA), which bypass the execution of the main code. In embedded platforms, DMA buffers are typically allocated by the application or by the OS, so we know where they are located in main memory. We may either always consider these memory areas as modified, or flag them as modified as soon as the corresponding peripheral interrupts fire, independent of their processing.

The `record()` function. We implement **`record()`** as a variable argument function. In the case of proactive systems, this allows us to aggregate multiple changes in main memory with a single call, as shown in Figure 3-6(c).

Among the many data structures available to store the modification records, we choose to employ a simple bit-array, where each bit represents one byte in main memory as modified or not. This representation is particularly compact, causing little overhead in main memory. Crucially, it allows **`record()`** to run in constant time, as it supports direct access to arbitrary elements. This is key to prevent **`record()`** from changing the application timings, which may be critical on resource-constrained embedded platforms [42].

Such a data structure, however, causes no overhead on NVM, as it does not need to be part of the checkpoint. Every time the system resumes from the previous checkpoint, we start afresh with an empty set of modification records to track the differentials since the time system restarts.

Checkpoint procedures. We also need to replace the existing checkpoint procedures with a DICE-specific one.

Hibernus [11] and Hibernus++ [10] set the voltage threshold for triggering a checkpoint to match the energy cost for writing the entire main memory on NVM, as they use a *copy-all* strategy. HarvOS [18] bases the same decision on a worst-case estimate of the energy cost for checkpointing at specific code locations, as a function of stack size.

When using DICE, due to its ability to limit checkpoints to the slice of the application state that changed, both approaches are overly pessimistic. We set these parameters based on an estimate of the *actual* cost for checkpointing. We obtain this by looking at how many modification records we accumulate and the positions of ST and BP at a given point in the execution.

Differently, MementOS [130] sets the threshold for triggering a checkpoint based on repeated emulation experiments using progressively decreasing voltage values and example energy traces, until the system cannot complete the workload. This processing requires no changes when using DICE; simply, when using DICE, the same emulation experiments will generally return a threshold smaller than in the original MementOS, as the energy cost of checkpoints is smaller.

Similar to existing work [101, 130], we also ensure the validity of a checkpoint by adding a canary value at the beginning and at the end of the checkpoint.

3.7 Benchmark Evaluation

We dissect the performance of DICE using four metrics:

- The *update size* is the amount of data written to NVM during a checkpoint. This is the key metric that DICE seeks to reduce: measuring this figure is essential to understand the performance of DICE in all other metrics.
- The size of the *smallest energy buffer* is the smallest amount of energy that allows the system to complete a workload. If too small, a system may be unable to complete checkpoints, ending up in a situation where the execution makes no progress. However, target devices typically employ capacitors: a smaller capacitor reaches the operating voltage sooner and enables smaller device footprints.
- The *number of checkpoints* is the number of times the system must take a checkpoint to complete a workload. The more the checkpoints, the more the system subtracts energy from useful computations. In contrast, reducing NVM operations allows the system to use energy more for computations than checkpoints, allowing an application to progress further on the same charge.

- The *completion time* is the time to complete a workload, excluding the recharge times that are deployment-dependent. DICE introduces a run-time overhead due to recording differentials. On the other hand, fewer NVM operations reduce both the time required for a single checkpoint and, because of the above, their number.

3.7.1 Settings

Benchmarks. We consider three benchmarks widely employed to evaluate system support for intermittently-powered computing [11, 79, 130, 152]: *i)* a Fast Fourier Transform (FFT) implementation, *ii)* RSA cryptography, and *iii)* Dijkstra spanning tree algorithm. FFT is representative of signal processing functionality in embedded sensing. RSA is a paradigmatic example of security support on modern embedded systems. Dijkstra's spanning tree algorithm is a staple case of graph processing, often found in embedded network stacks [76].

These benchmarks offer a variety of different programming structures, data types, memory access patterns, and processing load. For example, the FFT implementation operates mainly over variables local to functions and has moderate processing requirements; RSA operates mainly on global data and demands great MCU resources; whereas Dijkstra's algorithm mainly handles integer data types as opposed to variable-precision ones, but exhibits much deeper levels of nesting due to loops and function calls. This diversity allows us to generalize our conclusions. All implementations are taken from public code repositories [107].

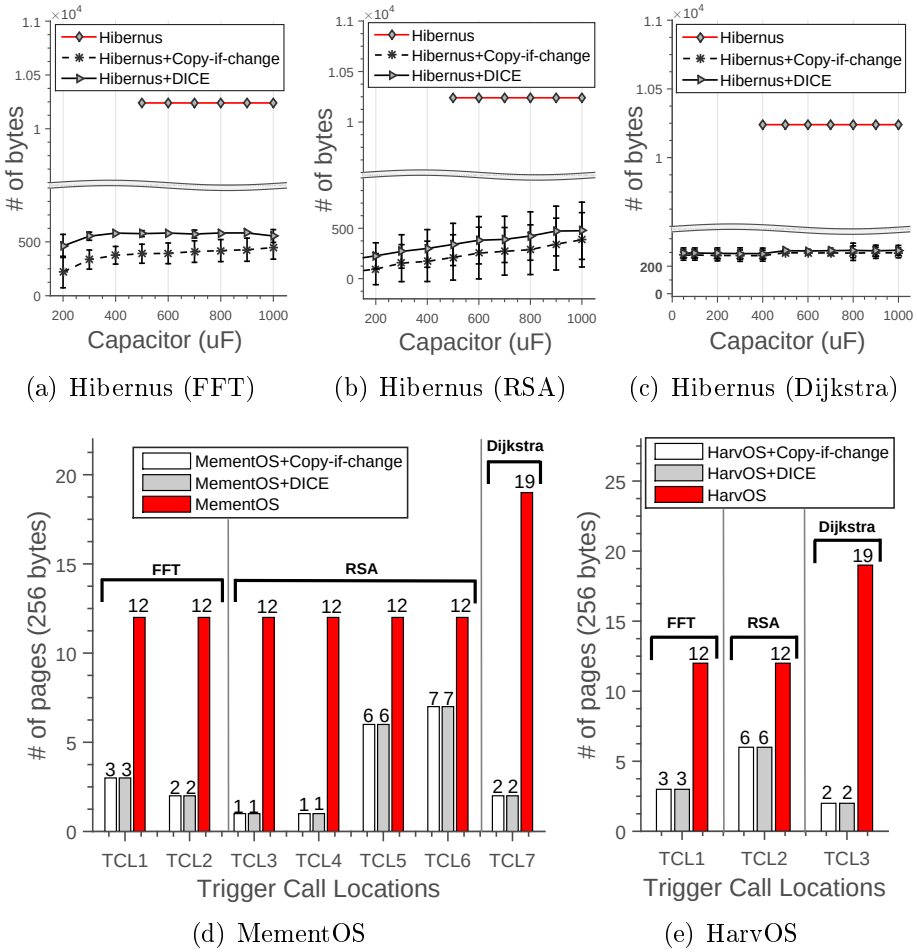
Systems and platforms. We measure the performance of DICE with both reactive (Hibernus) and proactive (MementOS, HarvOS) checkpoints, investigating the different instrumentation strategies in Section 3.5.1. We consider as baselines the unmodified systems using either the *copy-all* or *copy-used* strategies. We also test the performance of *copy-if-change* [16] with either of the existing systems. To make our analysis of MementOS independent of the energy traces used to identify a suitable voltage threshold, we manually sweep the possible parameter settings with steps of 0.2V, and always use the best performing one.

We run Hibernus on an MSP430-based TelosB interfaced with a byte-programmable 128 KByte FRAM chip, akin to the hardware originally used for Hibernus [11]. MementOS and HarvOS run on a Cortex M3-based ST Nucleo with a standard flash chip, already used to compare MementOS and HarvOS [18]. Both boards offer a range of hooks to trace the execution, enabling fine-grained measurements. Further, our choice of platforms ensures direct comparison with existing literature. In the same way as the original systems [10, 18, 130], our experiments focus on the MCU. Peripherals may operate through separate energy buffers [55] and dedicated solutions for checkpointing their states also exist [102, 146].

For these experiments, we use a foundational power profile often found in existing literature [16, 79, 130, 152]. The device boots with the capacitor fully charged, and computes until the capacitor is empty again. In the mean time, the environment provides no additional energy. Once the capacitor is empty, the environment provides new energy until the capacitor is full again and computation resumes.

This profile generates paradigmatically intermittent executions [11, 79]. The more the environment provided energy while the device is computing, therefore postponing the time when a checkpoint is needed, the more the execution would resemble a traditional one, where no checkpoints are needed. Besides, this profile is also representative of a staple class of intermittently-powered applications, namely, those based on wireless energy transfer [23, 130]. With this technology, devices are quickly charged with a burst of wirelessly-transmitted energy until they boot. Next, the application runs until the capacitor is empty again. The device rests dormant until another burst of wireless energy comes in.

To accurately compute the metrics above, we trace the execution on real hardware using an attached oscilloscope along with the ST-Link in-circuit debugger and Kiel μVision for the Nucleo board. This equipment allows us to ascertain the time taken and energy consumption of every operation during the execution, including checkpoints on FRAM or flash memory. The results are obtained from 1,000 (10,000) benchmark iterations on the MSP430 (Cortex M3) platform.



(a) Hibernus (FFT) (b) Hibernus (RSA) (c) Hibernus (Dijkstra)

(d) MementOS (e) HarvOS

Figure 3-9: Update size. *The size of NVM updates is significantly smaller when using DICE compared to the original systems. Compared to* copy-if-change*, it remains the same or marginally larger.*

3.7.2 Results → Update Size

Figure 3-9 shows the results for reduction in update size enabled by DICE. With Hibernus, the code location where a checkpoint takes place is unpredictable: depending on the capacitor size, an interrupt eventually fires prompting the system to checkpoint. Figure 3-9(a), 3-9(b), and 3-9(c)[2] thus report the average update size we measure during an experiment until completion of the workload, as a function of the capacitor size. Compared to the original *copy-all* strategy, DICE provides orders of magnitude improvements. These gains are a direct result of limiting updates to those

[2]Some data points are missing in the charts for the original design of Hibernus, as it is unable to complete the workload in those conditions. We investigate this aspect further in Section 3.7.4.

determined by the modification records. On the other hand, using *copy-if-change* with Hibernus provides marginal advantages over DICE, because modifications in the call stack are recorded at word-, rather than frame-granularity.

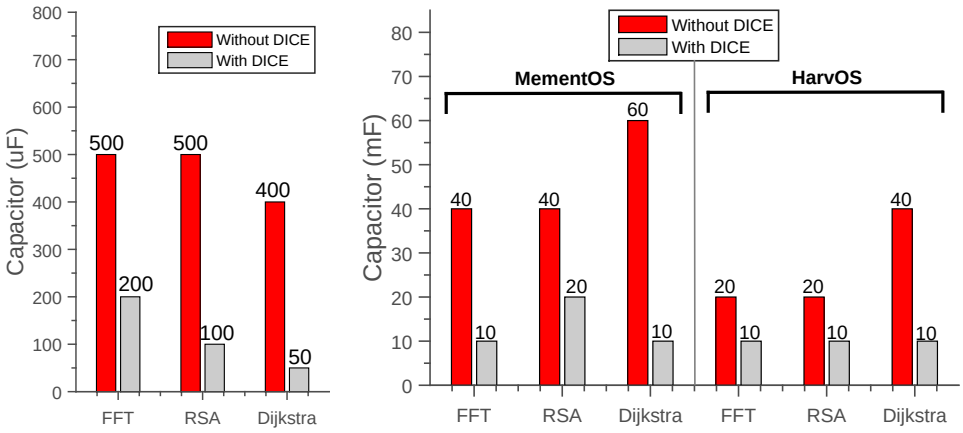
With the original MementOS and HarvOS, the update size is a function of the location of the trigger call, because the stack may have different sizes at different places in the code. Figure 3-9(d) and Figure 3-9(e)[3] show that DICE reduces the update size to a fraction of that in the original *copy-used* strategy, no matter the location of the trigger call. The same charts show that the performance of *copy-if-change* when combined with MementOS or HarvOS is the same as DICE. This is an effect of the page-level programmability of flash storage, requiring an entire page to be rewritten on NVM even if a small fraction of it requires an update.

Overall, the cost for *copy-if-change* to match or slightly improve the performance of DICE in update size is, however, prohibitive in terms of energy consumption. *Copy-if-change* indeed requires a complete sweep of the checkpoint data on NVM before updating, and even for FRAM, the cost of reads is comparable to writes [110]. As an example, we compute the energy cost of a checkpoint with the data in Figure 3-9(b) to be 93% higher with *copy-if-change* than with DICE, on average. For Hibernus, *copy-if-change* would result in an energy efficiency worse than the original *copy-all* strategy. Similar considerations apply when using the technique of Aouda et al. [8], due to the operation of the garbage collector. As energy efficiency is the figure users are ultimately interested in, we justifiably narrow down our focus to comparing a DICE-equipped system with the original ones.

3.7.3 Results → Smallest Energy Buffer

Figure 3-10 reports the minimum size of the capacitor required to complete the given workloads. A DICE-equipped system constantly succeeds with smaller capacitors. With Hibernus, DICE allows one to use a capacitor that is up to 88% smaller than the one required with the original *copy-all* strategy. Similarly, for MementOS and

[3]For MementOS, the trigger call locations refer to the "function call" placement strategy in MementOS [130]. We find the performance with other MementOS strategies to be essentially the same. We omit that for brevity.



(a) Hibernus (b) MementOS and HarvOS

Figure 3-10: Smallest capacitor. *A DICE-equipped system completes the workload with smaller energy buffers. This is due to a reduction in the energy cost of checkpoints, enabled by the reduction in update size.*

HarvOS, the smallest capacitor one may employ is about half the size of the one required in the original designs. Smaller capacitors mean reaching operating voltage faster and smaller device footprints.

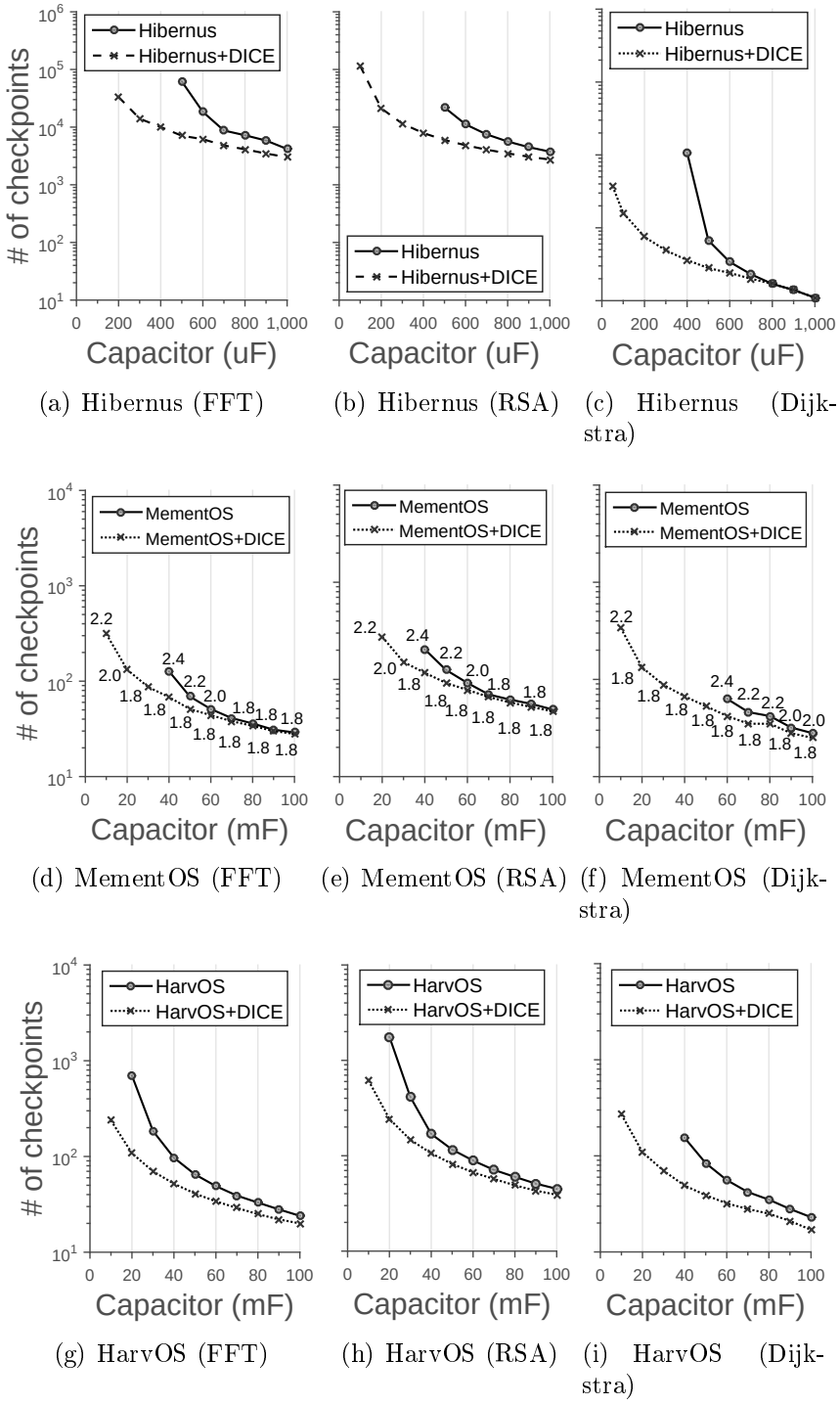
Such a result is directly enabled by the reduction in the update size, discussed in Section 3.7.2. With fewer data to write on NVM at every checkpoint, their energy cost reduces proportionally. As a result, the smallest amount of energy the system needs to have available at once to successfully complete the checkpoint reduces as well. Provided the underlying system support correctly identifies when to start the checkpoint, the workload can be completed with a smaller capacitor.

3.7.4 Results → Checkpoints

Figure 3-11[4] depicts the reduction in the number of checkpoints against variable capacitor sizes.

These results are directly enabled by the reduction in update size, discussed in Section 3.7.2. Hibernus adopts the copy-all strategy to checkpoint entire main memory onto NVM regardless of its occupation, as discussed in Section 3.4. As a result, it ends up spending more energy on checkpoints rather than program execution. This ultimately results in Hibernus consuming more checkpoints than MementOS and HarvOS

[4]For MementOS, we tag every data point with the minimum voltage threshold that allows the system to complete the workload, if at all possible, as it would be returned by the repeated emulation runs [130].

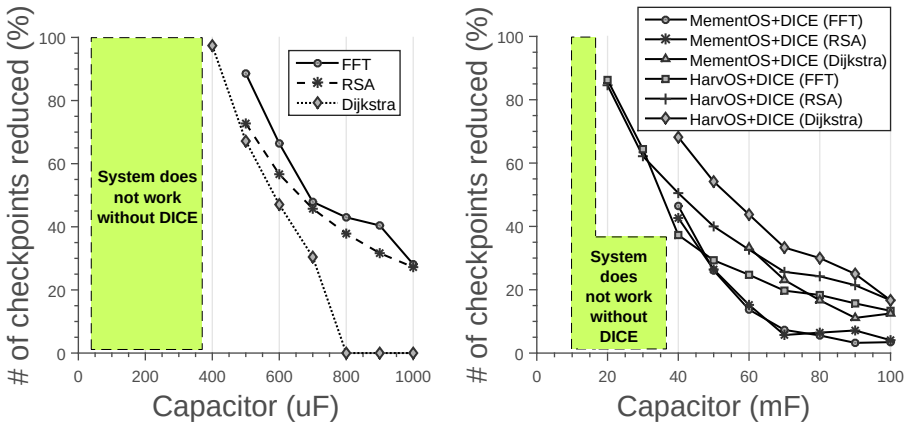


(a) Hibernus (FFT) (b) Hibernus (RSA) (c) Hibernus (Dijkstra)

(d) MementOS (FFT) (e) MementOS (RSA) (f) MementOS (Dijkstra)

(g) HarvOS (FFT) (h) HarvOS (RSA) (i) HarvOS (Dijkstra)

Figure 3-11: Number of checkpoints necessary against varying capacitor sizes. *With smaller capacitors, highly-intermittent executions greatly benefit from DICE.*



(a) Hibernus (b) MementOS and HarvOS

Figure 3-12: Improvement at different capacitor sizes enabled by DICE. *The improvements are larger in highly-intermittent executions.*

for the same workload, as shown in Figure 3-11(a), Figure 3-11(d) and Figure 3-11(g).

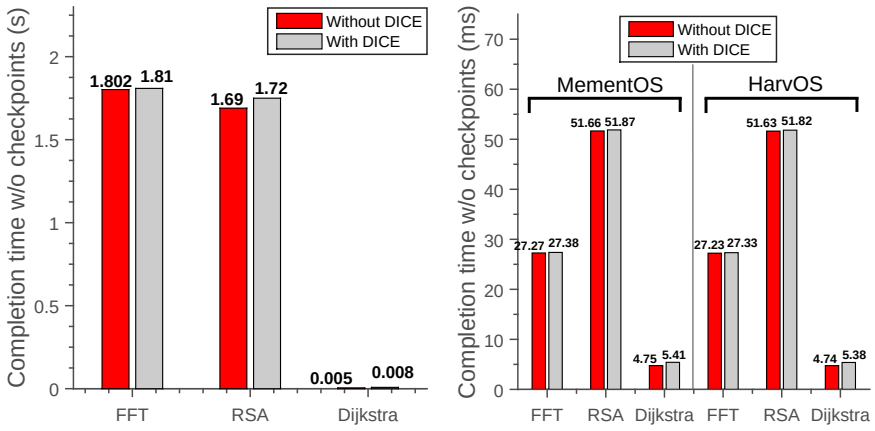
Although MementOS and HarvOS only checkpoint allocated portions of memory, this is still much larger than the actual change in application state. A DICE-equipped system recognizes these state differentials, thus minimizing the amount of energy spent on and the number of checkpoints.

Interestingly, a significant area of these charts only shows the performance of the DICE-equipped systems, as the original ones are unable to complete the workload with small capacitors. As soon as a comparison is possible, the improvements for DICE with small capacitors are significant and apply consistently across benchmarks as visible in Figure 3-11(a), Figure 3-11(b) and Figure 3-11(c).

With fewer data to write on NVM at every checkpoint, the energy cost of this operation reduces proportionally. This has two direct consequences. First, the smallest amount of energy the system needs to have available at once to complete the checkpoint reduces as well. Smaller capacitors mean reaching operating voltage faster and smaller device footprints. Second, the system can invest the available energy to compute rather than checkpointing, improving the overall energy efficiency. In this setting, DICE provides the greatest advantages.

With larger capacitors, the improvements in Figure 3-11 are smaller, but still appreciable[5]. This is expected: the larger is the capacitor, the more the applica-

[5]Note the log scale on the Y axis of Figure 3-11.



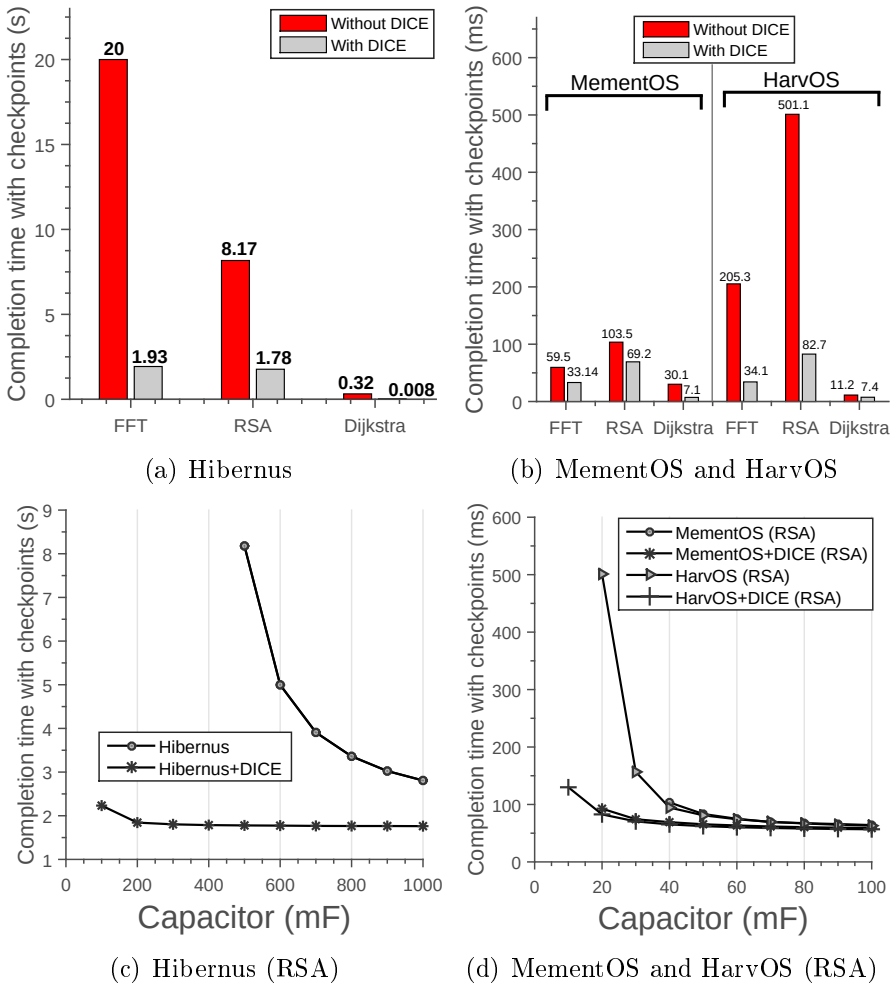
(a) Hibernus (b) MementOS and HarvOS

Figure 3-13: Completion time without concrete checkpoints. *The additional run-time overhead due to code instrumentation is limited.*

tion progresses farther on a single charge, thus executions are less intermittent and checkpoints are sparser in time. As a consequence, modifications since the previous checkpoint accumulate as a result of increased processing times. The state of main memory then becomes increasingly different than the checkpoint data, and eventually DICE updates a significant part of it.

Figure 3-12 provides a summary view on these results, plotting the percentage reduction unlocked by DICE against variable capacitor sizes and across systems. A significant part of the chart is empty as no comparison is possible, because the workload cannot be completed in the original designs. When a comparison is possible, improvements are largest with smaller capacitors, topping above 80% in most cases. The curves ultimately tend to flatten with larger capacitors for the aforementioned reasons.

This performance also allow systems to reduce the time invested in checkpoint operations, because of a reduction in the time taken for single checkpoints due to fewer NVM operations and in their number. In turn, this reduces the time to complete the workload, as we investigate next.



(a) Hibernus (b) MementOS and HarvOS

(c) Hibernus (RSA) (d) MementOS and HarvOS (RSA)

Figure 3-14: Completion time including checkpoints. *The run-time overhead due to DICE is overturn by reducing size and number of checkpoints, ultimately resulting in shorter completion times.*

3.7.5 Results → Completion Time

DICE naturally imposes a cost for the benefits reported thus far. Such a cost materializes as run-time overhead due to recording differentials, which may increase the time to complete a given workload. On the other hand, based on the above results, DICE enables more rapid checkpoints as it reduces NVM operations. In turn, this allows the system to reduce their number as energy is spent more on completing the workload than checkpoints. Both factors should reduce the completion time.

Figure 3-13 investigates this aspect in a single iteration of the benchmarks where the code executes normally, but we skip the actual checkpoint operations. This way, we observe the net run-time overhead due to executing **`record()`**. The chart shows

that this overhead is generally limited. This is valid also for reactive checkpoints in Hibernus, despite the conservative approach at placing **`record()`** calls due to the lack of knowledge of where the execution is preempted.

Figure 3-14 includes the time required for checkpoint operations with the smallest capacitor allowing both the DICE-equipped system and the original one to complete the workload, as per Figure 3-10. The overhead due to executing **`record()`** calls is not only compensated, but actually overturn by fewer, more rapid checkpoints. Using these configurations, DICE allows the system to complete the workload much earlier, increasing the system's responsiveness.

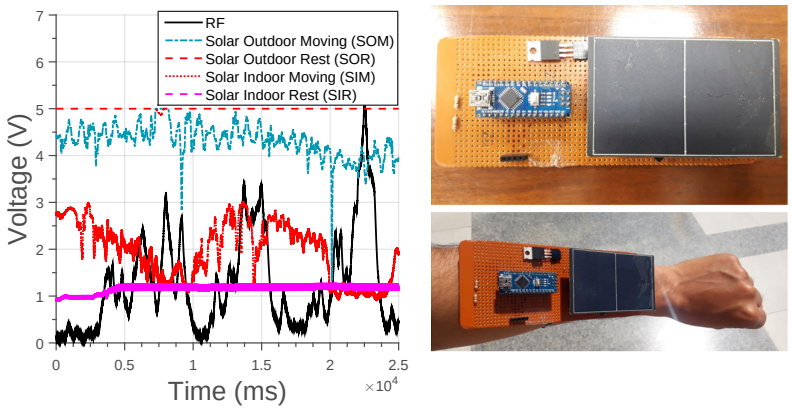
Differently, Figure 3-14(c) and Figure 3-14(d) provide an example of the trends in completion time against variable capacitor sizes. The improvements are significant in a highly-intermittent computing setting with smaller capacitors. Similar to Figure 3-11, two factors contribute to the curves in Figure 3-14(c)and Figure 3-14(d) approaching each other. Larger capacitors allow the code to make more progress on a single charge, so the number of required checkpoints reduces. As more processing happens between checkpoints, more modifications occur in application state, forcing DICE to update a larger portion of the checkpoint data. Eventually, these two factors compensate each other.

3.8 Application Evaluation

We investigate the impact of DICE in a full-fledged application, using a variety of different power profiles. We build the same activity recognition (AR) application used for evaluating several support systems for intermittent computing [32, 101, 104, 130], including relying on the same source code [49]. We use an MSP430F2132 MCU, that is, the MCU used in the WISP platform [137] for running the same AR application. The rest of the setup is as for Hibernus in Section 3.7.

We focus on number of checkpoints and completion time, as defined in Section 3.7.1, in a single application iteration.

Traces. We consider five power traces, obtained from diverse energy sources and in



(a) Voltage traces excerpt. (b) Device setup.

Figure 3-15: Example voltage traces and device setup.

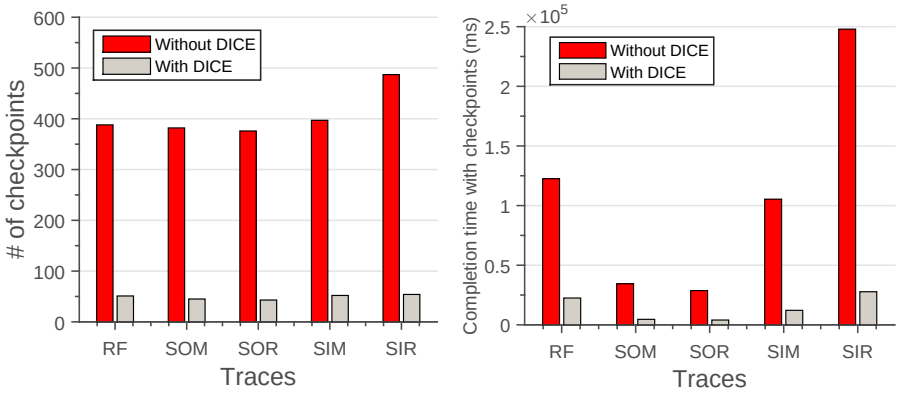
different experimental settings.

One of the traces is the RF trace from MementOS [88, 130], recorded using the RF front end of the WISP 4.1. The black curve in Figure 3-15(a) shows an excerpt of this trace, plotting the instantaneous voltage reading at the energy harvester over time. The curve is oscillating as it was recorded by a person carrying the WISP while walking around an RFID reader within about eight feet of range.

We collect four additional traces using a mono-crystalline high-efficiency solar panel [147], shown in Figure 3-15(b), in different settings. The solar panel has high efficiency ($\approx 20\%$) with good response to both indoor and outdoor lightning. We use an Arduino Nano [9] to measure the voltage output across a 30kOhm load, roughly equivalent to the resistance of an MSP430F2132 in active mode, attached to the solar panel.

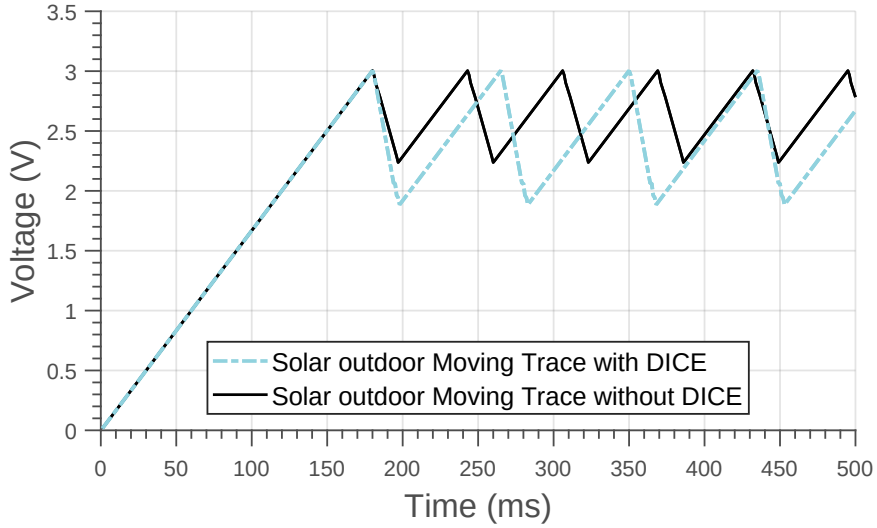
Using this device setup, we experiment with different settings. We attach the device to the wrist of a student to simulate a fitness tracker. The student roams in the university campus for outdoor measurements (SOM), and in our research lab for indoor measurements (SIM). Alternatively, we keep the device on the ground right outside the lab for outdoor measurements (SOR), and at desk level in our research lab for the indoor measurements (SIR). The curves other than the black one in Figure 3-15(a) exemplify the trends.

Overall, Figure 3-15(a) visually demonstrates the extreme variability and consid-



(a) Number of checkpoints.

(b) Completion time.

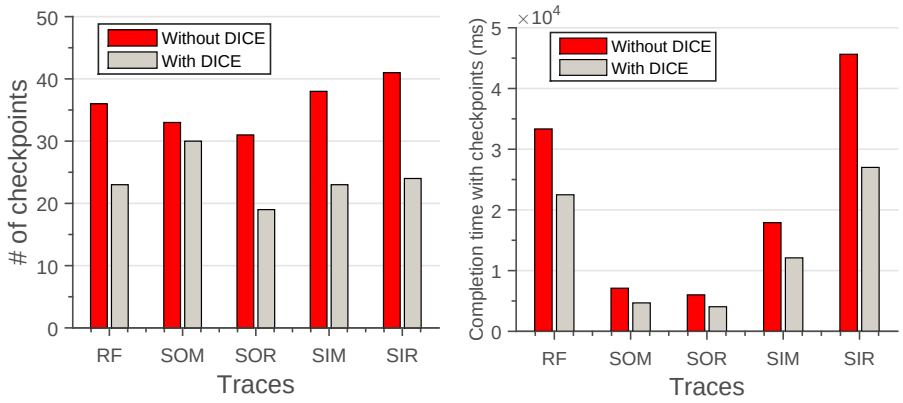


(c) Voltage Variation

Figure 3-16: Performance of the AR application running on Hibernus with a 50uF capacitor. *Performance gains are observed across diverse power traces obtained from different energy sources.*

erable differences among the power traces we consider. For example, the RF power trace often reaches a 0V output, due to multipath loss worsened by RF harvester mobility. Conversely, the solar traces always maintain a non-zero output, yet with crucial differences. When operating outdoor, the output voltage rarely falls below the operating voltage of the hardware we use, whereas this happens frequently in an indoor setting. Orthogonal to this, mobility induces substantial variations in the voltage output, due to shadows and occlusions, whereas the output is nearly constant in a static setup.

Results. Across all power traces, we find out that a 10uF capacitor is sufficient for



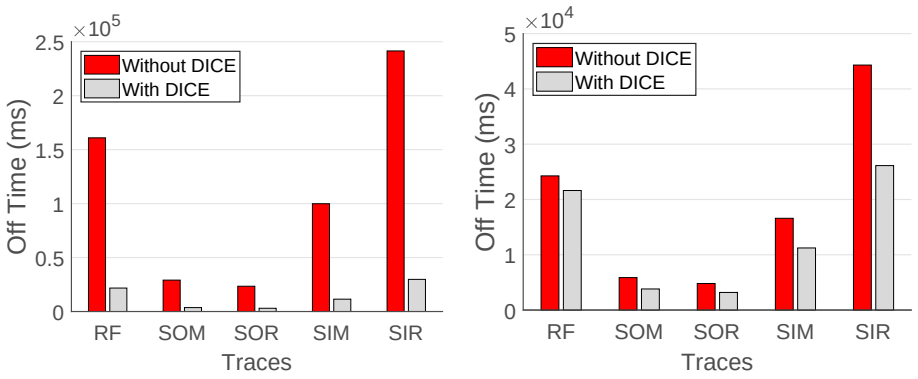
(a) Number of checkpoints. (b) Completion time.

Figure 3-17: Performance of the AR application running on Hibernus with a 100uF capacitor. *Compared with Figure 3-16, and as seen in Section 3.7, performance improvements are larger with more intermittent executions.*

a DICE-equipped Hibernus to complete an iteration of the AR application. Without DICE reducing the size of checkpoints, and thus their energy cost and time, Hibernus needs a five times larger capacitor to complete the same workload.

Figure 3-16 shows the performance with a 50uF capacitor, where both the original Hibernus and the DICE-equipped one can complete the workload. The number of required checkpoints diminishes by one order of magnitude using DICE, as shown in Figure 3-16(a), and we appreciate similar improvements for completion time as well, as shown in Figure 3-16(b). Fewer checkpoints and shorter completion times mean better energy efficiency and increased reactivity to external events, ultimately providing increased quality of service to end users.

Best absolute performance in Figure 3-16 is obtained with the outdoor solar power trace in a static setup, as expected in that it supplies the largest energy. However, DICE constantly improves over the original Hibernus, regardless of which of the diverse power traces is considered. This performance evidently originates from the ability of DICE to abate the energy costs of and time taken for checkpoints. We achieve this by limiting NVM operations to updating the relevant slices depending on changes in the application state, as described in Section 3.4. This observation is confirmed also by looking at the voltage threshold Hibernus uses with DICE: on average, checkpoints start at 2.07V, rather than 2.8V as required in the original



(a) Off time for 50uF. (b) Off time for 100uF.

Figure 3-18: Performance of the AR application running on Hibernus with a 50uF capacitor. *Reduced off-time is observed across diverse power traces obtained from different energy sources.*

Hibernus as shown in 3-16(c).

The trends we discuss in Section 3.7 with larger capacitors are confirmed here. Figure 3-17 plots the performance using a 100uF capacitor. Improvements are reduced compared with Figure 3-16: applications progress farther on a single charge, checkpoints are sparser, and DICE records more changes to the application state between checkpoints. This reflects also in the voltage threshold used for triggering a checkpoint: the DICE-equipped Hibernus triggers a checkpoint at 1.93V, whereas the original Hibernus starts checkpoints at 2.4V.

We also analyze the impact of reduced checkpoint size on *off-time*, the total amount of time the device spends in recharging the buffer. It reflects on device's ability to react to external events, network on time, and quality of service in general. Figure 3-18(a) shows the off-time on a 50uF capacitor. A DICE equipped system achieves significantly smaller off-times as compared to original Hibernus, as shown in Figure 3-18. This ability stems from DICE's ability to reduce the number of checkpoints and, therefore, the number of recharge cycles needed to complete the workload. Figure 3-18(b) plots the performance using a 100uF capacitor. Improvements are reduced compared with Figure 3-18(a) as the larger capacitor also benefits the original systems.

3.9 Summary

To achieve energy-efficient checkpointing, today's checkpointing strategies limit the programmer to not use specific memory segments. On the other hand, they perform a coarse grained analysis to find the application-state required to be checkpointed in an attempt to save the computational overhead required for a fine-grained analysis. However, these approach have two major pitfalls. First, performing a coarse grained analysis forces the system to perform large sized updates at the time of checkpoint thereby overcome the amount of energy saved by performing a coarse grained analysis. Second, excluding specific memory segments our of the checkpoint create a risk of faulty checkpointed state which can result in unpredictable application behaviour upon device's resumption after energy failure.

In order to overcome these limitations of state-of-the-art checkpointing strategies, we present DICE; a solution to track changes in the application state at run-time. To do so, our compile-time instrumentation tracks changes in application state at run-time and records them in main memory. Based on this, DICE updates the existing checkpoint data by limiting NVM operations to those strictly necessary, helping existing systems complete a given workload with *i)* fewer checkpoints, thus better energy efficiency, and *ii)* shorter times, thus increased reactiveness and better quality of service.

Our benchmark evaluation, based on a combination of three benchmarks across three different existing system support and two different hardware platforms, provides quantitative evidence. For example, using DICE, HarvOS can complete the execution of the RSA algorithm with 86% fewer checkpoints, resulting in better overall energy utilization and a 34% reduction in completion time. These improvements are confirmed in concrete applications, yielding better energy efficiency and increased reactivity to external events. Experiments based on an activity recognition application show order of magnitudes improvements when using Hibernus and against power traces as diverse as RF-based wireless energy transfer and solar radiation.

In the next chapter we move the focus of our discussion towards the placement

of **`trigger()`** calls for checkpoints. We develop a compile-time tool which uses a precise energy model that helps transiently powered devices in making an informed decision about when to trigger checkpoint. We also show integration of our energy model helps existing state-of-the-art systems enables significant performances improvement of those systems.

Chapter 4

Modeling Dynamic Energy Consumption

Transiently-powered systems aim to trigger checkpointing precisely when they only have energy left to successfully save the program state. Existing solutions use a pessimistic approach to trigger checkpointing thus causing redundant checkpoints. Hence, accurate estimation of the energy consumption for checkponting routine is required to ensure timely execution of checkpointing procedure, at the same time, svaing energy.

In chapter 3, we presented the checkpoint size reduction technique that allows the device to maximally utilize the energy budget for program execution by spending lesser energy on checkpointing. This chapter takes the next step to accurately estimate the energy consumption of an intermittent program; enabling accurate placement of trigger calls. To this end, we introduce a compile-time tool for energy prediction in intermittent computing, EPIC, allowing the device to place trigger calls at locations where the device is predicted to die on the given energy budget. Furthermore, the developed energy model can easily be integrated into the state-of-the-art systems which helps them improve their performance. The main purpose of the energy model is to incorporate the energy dynamic of harvested energy thus allowing precise analysis of intermittent programs.

Our evaluation shows that EPIC drastically improves run-time energy efficiency,

eventually leading up to a 350% speedup in the time to complete a fixed workload for a system employing proactive triggering mechanism. Further, when used with an existing debugging tools, CleanCut [33], programmers were able to avoid unnecessary program changes that hurt energy efficiency. Conversely, we built MSPsim++ to be used in MementOS [130] instead of of the original MSPsim. MementOS uses MSPsim to determine the most efficient system-wide voltage threshold when to trigger a checkpoint. The voltage threshold identified with MSPsim++ is typically much smaller, as it is aware of the increase in energy efficiency with the variation of voltage. The energy saved from sparing checkpoints is shifted to useful computations, yielding up to more than 1000% speedup in workload completion time.

The remainder of this chapter is structured as follows. We motivate the problem space and describe a key observation, in Section 4.1, which forms the basis of the contribution of this chapter. Section 4.2 gives a background on the related work. Section 4.3 presents the variation of energy consumption, its measurement and representation of our energy model to capture it. Section 4.4 encapsulates the workflow of EPIC. We present evaluation of our model with two existing state-of-the-art systems in Sections 4.5 while integrating our model with simulator in Section 4.6. Finally, we discuss the broader impact of EPIC in the intermittent computing domain and related work in section 4.6 and 4.7 respectively, while, concluding the discussion in Section 4.8.

4.1 Motivation

TPCs may rely on a great variety of energy harvesting mechanisms, often characterized by strikingly different performance and unpredictable dynamics across space and time [17]. As much as using solar cells may yield up to $240mW$, but only at specific times of the day and certain environmental conditions [52], harvesting energy from RF transmissions solely produces up to $1\mu W$ whenever the transmitter is sufficiently close [6].

Hardware and software must be dimensioned and parameterized according to these dynamics. Capacitors are used as ephemeral energy buffers. Smaller capacitors yield

smaller device footprints and quicker recharge times, at the expense of smaller overall energy storage. The microcontroller units (MCUs) also feature numerous configuration parameters. For example, lower clock frequencies allow one to exploit larger operating ranges in supply voltage, but slow down execution. The MSP430-series MCUs run with supply voltages as low as 1.8V at 1 MHz, but are unable to run any lower than 2.9V at 16 MHz.

As TPCs increasingly employ separate capacitors to power the MCU or peripherals [56], the ability to accurately forecast the energy cost of a certain fragment of code is key to decide on the size of capacitors powering the MCU and on its frequency settings. Peripheral operations may often be postponed if their capacitors have insufficient energy [56], whereas the MCU coordinates the functioning of the entire system. Accurate energy forecast information aids the efficient placement of function calls that checkpoint the MCU state on non-volatile memory to cross periods of energy unavailability [11, 18, 130]. Programmers may alternatively rely on task-based programming abstractions that offer transactional semantics [32, 101, 104]. Thus, they need to know the worst-case energy costs of given task configurations to ensure completion within a single capacitor charge, or forward progress may be compromised.

4.1.1 Observation

Modeling energy consumption of TPCs is an open challenge [33, 57, 99]. Existing tools are mainly developed for battery-powered embedded devices, which typically enjoy consistent energy supplies for relatively long periods. In contrast, capacitors on TPCs may discharge and recharge several times during a single application run. A single iteration of a CRC code may require up to 17 charges and consequent discharges when harvesting energy from RF transmissions [130]. Single executions of even straightforward algorithms thus correspond to a multitude of rapid sweeps of an MCU's operating voltage range [18, 130].

We *experimentally observe* that such a peculiar computing pattern causes severe fluctuations in an MCU's energy consumption. Figure 4-1 shows example fluctuations we measure on an MSP430G2553 MCU running at 1 MHz in a single *power cycle*,

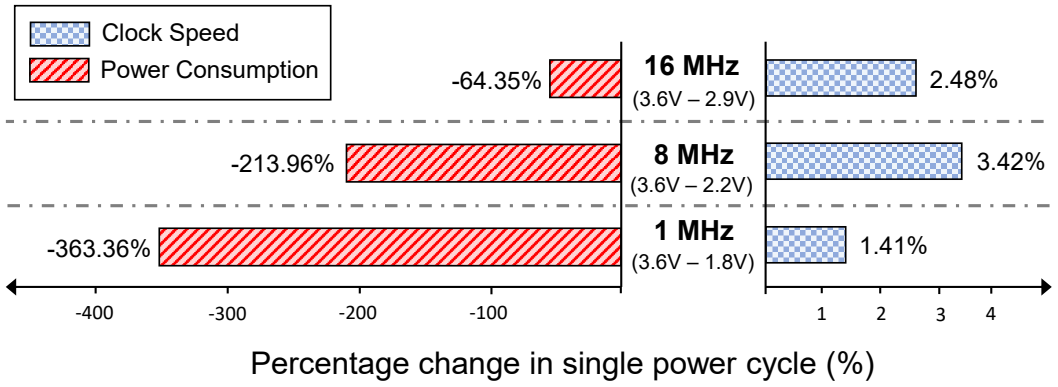


Figure 4-1: Impact of supply voltage variations on MSP430G2553 clock speed and power consumption in a single power cycle. *Existing tools typically ignore these behaviors when modeling the energy consumption of TPC.*

that is, as it goes from the upper to the lower extreme of the operating voltage range. Power consumption varies significantly; it reduces by a factor of up to 363.36%. Clock speed also changes; it increases by a factor of up to 3.42%. This means the same instruction takes different times depending on the supply voltage.

The combined fluctuations of power and clock in Figure 4-1 cause the energy cost of each MCU cycle to drop by up to 5× in a power cycle. Figure 4-2 shows this behavior as a function of supply voltage, again in a single power cycle. Even for a single application run, as mentioned earlier, the system may require thousands of power cycles; the net effect thus accumulates in the long run.

Unlike dynamic frequency or voltage scaling [118, 122] in mainstream platforms, these dynamics happen regardless of the current system workload and the software layers have no control on them. Fluctuations in power consumption are exclusively due to the dynamics in supply voltage and clock speed. In turn, the latter are due to the design of the digitally-controlled oscillators (DCOs) that equip TPCs such as the MSP430-based ones. In fact, TI designers confirm that many of their MSP430-series MCUs employ DCOs that cause the clock speed to increase as the supply voltages approaches the lower extreme[1] [73]. This yields better energy efficiency at these regimes, at the cost of varying execution times.

Reasoning on such dynamic behavior is not trivial. For simplicity, a vast fraction

[1]We verified this observation on three MSP430 models: G2553, G2452, and FR5969.

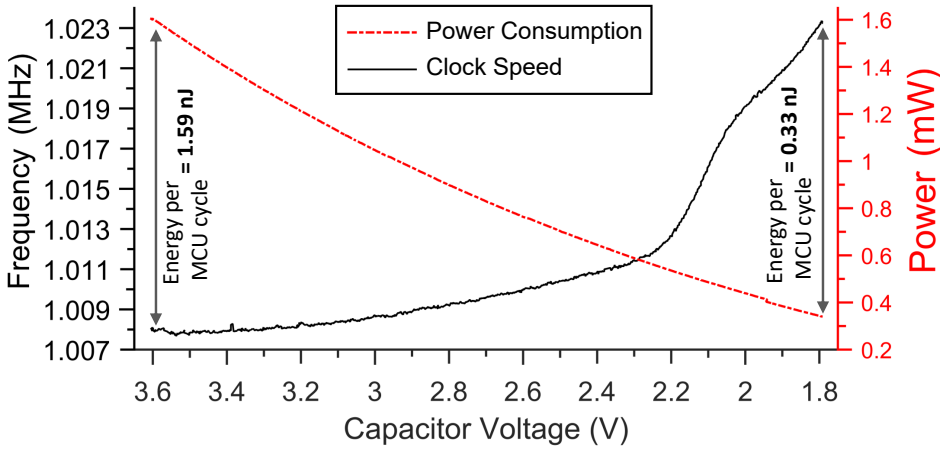


Figure 4-2: Impact of supply voltage variations on MSP430G2553 power consumption and clock speed. *Energy being a product of power (red dotted line) and execution time, which is a function of clock speed (black solid line), the energy cost of a single MCU cycle varies by up to ≈5× depending on the instantaneous supply voltage.*

of existing literature overlooks these phenomena [18, 33, 40, 130]. Many systems are designed and parameterized in overly-conservative ways, by considering a constant power consumption no matter the supply voltage, and fixed clock speeds. We argue, however, that considering these dynamics is crucial, as their impact magnifies for TPCs with rapid and recurring power cycles.

4.1.2 Major Contributions

We demonstrate that it is practically possible to accurately model and concretely capitalize on these dynamics. As a result, we obtain significant performance improvements in a range of situations, *exclusively enabled* by considering phenomena that are normally not accounted for. Following background material in Section 4.2, our contribution is three-pronged:

1. We empirically measure the impact of varying voltage supplies on *clock speed* and *power consumption* for all possible clock configurations, and derive an accurate energy model, described in Section 4.3. Analytically modeling these dynamics is difficult. Clock speed varies as a result of the DCO design, discussed earlier. Power consumption varies according to a nonlinear current draw by the

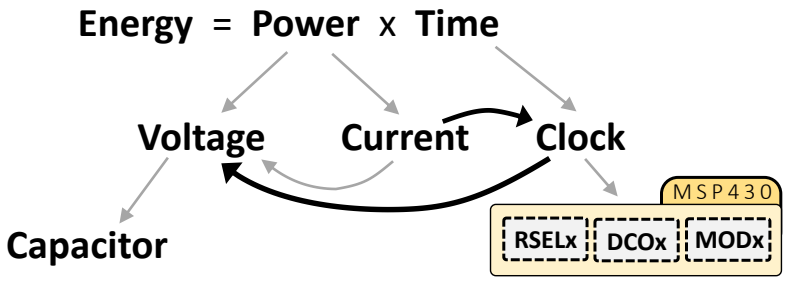


Figure 4-3: Dependencies among quantities determining energy consumption. *The direction of the arrows depict the direction of dependency. Power consumption depends upon the input voltage and the current draw. The execution time depends on clock speed. As the supply voltage rapidly varies, additional dependencies are created, shown by black arrows, that perturb an otherwise constant behavior.*

MCU, caused by inductive and capacitative reactance of the clock module that uses internal resistors to control the clock speed.

2. Our energy model enables the design and implementation of EPIC[2], an automated tool that provides accurate compile-time energy information, described in Section 4.4. EPIC first augments the source code with energy information at basic-block granularity. It then allows developers to tag a piece of code to determine best- and worst- case energy consumption.

3. We present MSPsim++ in Section 4.5, an integration of our energy model with MSPsim [40], a widely used instruction set emulator based on static power and clock models. This is to cater for the several systems that leverage run-time information. MSPsim++ emulates the dynamic behaviors of supply voltage, clock speed, and power consumption on a per-instruction basis. This is the finest possible granularity without resorting to real hardware.

4.2 Background

We describe the factors that concur to determining the energy consumption of TPCs at a fundamental level, to investigate the dependencies affected by the dynamics of supply voltage.

[2]Energy Prediction for Intermittent Computing

Calculating energy. Figure 4-3 graphically depicts the dependencies among the relevant quantities. Predicting energy consumption relies on precise values of both power consumption and execution time, as their impact is multiplicative. In embedded MCUs, energy consumption is typically estimated by deriving the execution time from the number of clock cycles taken, while power consumption is calculated by multiplying a supposedly constant voltage supply with the current draw for a given MCU resistance.

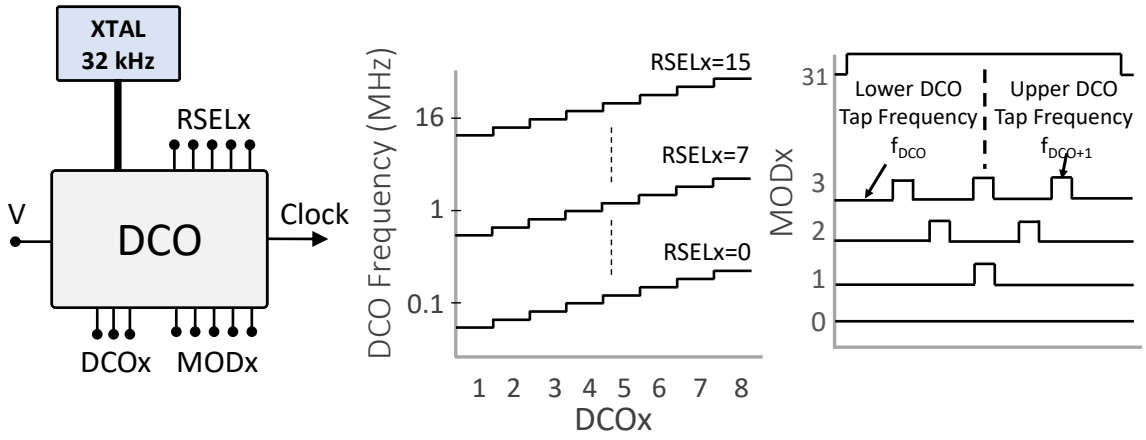
Supply voltage, however, depends on the charge of the energy storage facility, which is most often a capacitor in TPCs. Due to their extensive usage in electronics, the accuracy of existing capacitor models—related to (dis)charging behavior of the capacitor and voltage drop between the plates—is well studied [62].

The current drawn by the MCU depends on both the supply voltage and the clock speed. While the former dependency is straightforward (V=IR), the latter stems from internal resistance typically controlled through a clock-control register. Section 4.3 further discusses this aspect for MSP430-class MCUs. These are arguably de-facto standard on TPCs [56, 137] and represent the only case of commercially-available embedded MCU with non-volatile main memory, which aids saving state across power cycles.

The time factor used to calculate the energy consumption depends on the actual clock speed. Embedded MCUs offer specific parameters to configure the clock speed. For example, MSP430-series MCUs provide three parameters, called RSELx, DCOx, and MODx, as explained in Section 4.3.

No More Constant Power $\times$ Time. Existing energy estimation tools [40, 144] do model the dependencies shown by grey arrows in Figure 4-3. For simplicity, they tend to overlook the dependencies indicated by black arrows and consider constant values for the involved factors.

The same observation applies to many system solutions in TPCs [130]. As extensively discussed in Section 4.3, we experimentally observe that the assumption of constant power and execution time is actually not verified in TPCs. The impact of this is significant in TPCs, as the supply voltage may potentially traverse the whole



(a) MSP430's DCO module. (b) RSELx steps and DCOx range. (c) Modulator pattern.

Figure 4-4: Impact of DCO parameters on clock.

operational range multiple times during a single application run.

As the supply voltage varies wildly, a ripple effect creates that spreads the variability over time to clock speed and, in turn, to current draw. This essentially means that the phenomena traverse the dependencies in Figure 4-3 backward, eventually impacting both power consumption and execution time. As a result, these figures are no longer constant, but their values change as frequently as the supply voltage delivered by a capacitor. Nevertheless, both figures ultimately concur to determine energy consumption.

In the next section, we describe the empirical derivation of an energy model that accounts for these phenomena.

4.3 Energy Model

We describe the methodology to derive models accounting for the dependencies shown by black arrows in Figure 4-3. First, we discuss modeling the dependency between supply voltage and clock speed. Next, we describe the case of clock speed and current, which ultimately impact power consumption.

To make the discussion concrete, we target MSP430-class MCUs as arguably representative of TPC platforms, although our methodology applies more generally and

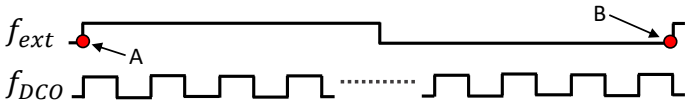
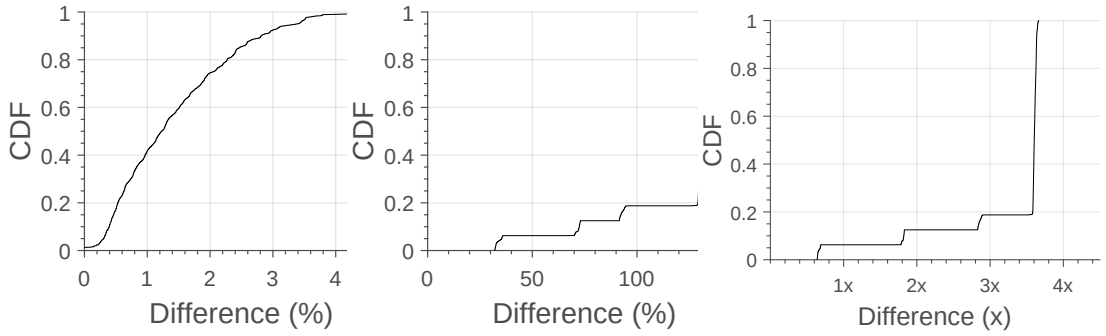


Figure 4-5: Clock frequency measurements. *The number of clock cycles (f_{DCO}) are counted between A and B, namely, two consecutive low to high transitions of the external crystal oscillator f_{ext}.*



(a) Frequency. (b) Current. (c) Power.

Figure 4-6: Impact of voltage supply variations on clock, current, and power for all DCO configurations. *The x-axis shows the difference in the corresponding factor when the voltage drops from one extreme of the MCU's operational voltage range to the other.*

has a foundational nature. Once an energy model is derived for other MCUs, the design of the tools we describe in the remainder of the chapter remains the same. The quantitative discussion that follows refer to the energy model we obtain for MSP430G2553 MCU which employs a 16-bit RISC instruction-set architecture with no instruction or data-cache while application data always resides in volatile main-memory; we find our conclusions to be equally valid for MSP430G2xxx MCUs, based on repeating the same modeling procedures.

4.3.1 Modeling Clock Drift

We first describe the basic operation of the clock module on MSP430 MCUs.

MSP430 clock module. As shown in Figure 4-4(a), MSP430 MCUs employ a digitally controlled oscillator (DCO) that can be configured to deliver frequencies from only a few KHz up to 16 MHz.

DCOs on MSP430 MCUs are configured using three parameters: RSELx, MODx,

and DCOx, which are together hosted in two MCU registers called DCOCTL and BCSCTL1. RSELx stands for resistor-select and is used to configure the DCO for one of the sixteen nominal frequencies in the range 0.06 MHz to 16 MHz. DCOx uses three bits to further subdivide the range selected by RSELx into eight uniform frequency steps as shown in Figure 4-4(b). Finally, MODx stands for DCO modulator, and enables the DCO to switch between the configured DCOx and the next higher frequency DCOx+1. The five bits of MODx define 32 different switching-frequencies, as depicted in Figure 4-4(c), to achieve fine-grained clock control.

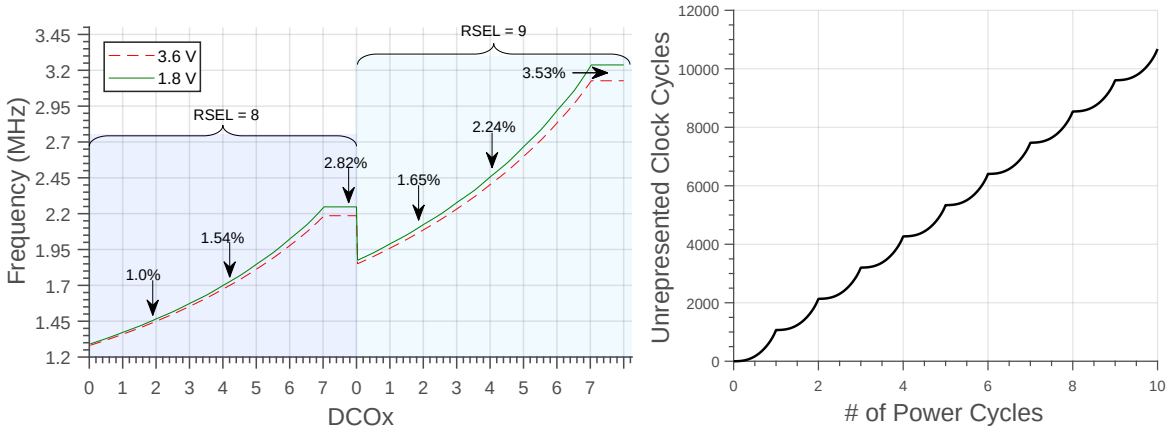
Measurement procedure. Measuring clock speeds is challenging as the oscilloscope probes, when hooked to the clock pin, perturb the DCO impedance. This results in fluctuating measurements.

We thus employ a verified software-based measurement approach used by Texas Instruments for DCO calibration [71]. It consists in counting the number of MCU ticks within one clock cycle of an external crystal oscillator, as shown in Figure 4-5. In MSP430 MCUs, the external crystal oscillator is a very stable clock source offering a frequency of 32.768 KHz. Since the time period of this oscillator is greater than the time period of MCU ticks, we use it to count the number of MCU ticks during its period.

We initialize the Capture/Compare register of Timer_A in the MSP430 to Capture mode. The output of DCO is wired to this register and captures the value of Timer_A when a low-to-high transition occurs on the reference signal, that is, the external oscillator. The captured value is the number of clock cycles between two consecutive low-to-high transitions of the reference signal.

Empirical model. To comprehensively model the clock behavior, we sweep the parameter space and empirically record the range of frequencies generated by different DCO configurations.

Altogether, 4096 discrete DCO frequencies can be generated using all possible combinations of RSELx, DCOx, and MODx. As observed in Figure 4-2, however, the supply voltage impacts the actual clock speed given a certain clock configuration. For each of these 4096 configurations, we evaluate this impact for the entire operational



(a) Sensitivity of clock to voltage

(b) Unrepresented clock cycles @ 8 MHz

Figure 4-7: Clock behavior. *(a) The sensitivity of clock to voltage increases for a given RSELx when the value of DCOx is increased, as well as across increasing RSELx values. The percentage values represent the difference in frequency between the two extremes of the operational voltage range; (b) The cup-shaped segments, with each cup corresponding to a single power cycle, show that the number of unrepresented cycle increases with the decreasing voltage of the capacitor. Charging times are omitted for brevity.*

range of the MCU at 0.001V intervals, and record over 69,888 unique frequencies. These fine-grained measurements allow us to analyze the sensitivity of the clock to supply voltage, and ultimately derive an accurate model. Our analysis is centered around two key question: *i) is the sensitivity of changes in clock speed to changes in supply voltage consistent across all frequencies?* and *ii) how essential is it to model the clock behavior?*

To answer the first question, in Figure 4-6(a) we plot the cumulative distribution of the *percentage difference in clock speed* when the voltage drops from one extreme of the MCU's operational voltage range to the other, across all possible DCO settings that determine a given MCU frequency. The arc-shaped curve in Figure 4-6(a) implies that the sensitivity of changes in clock speed, which ranges from ≈0% to ≈4.5% (x-axis), is not consistent across all frequencies.

We note, however, that this apparent inconsistency is not the outcome of a random clock behavior, but a predictable DCO artefact that is observable in most low-power MSP430 MCUs. Figure 4-7(a) shows how clock speed changes for a given RSELx when

DCOx increases, as well as across increasing RSELx values, for the two extremes of the operational voltage range i.e., 3.6 to 1.8. TI designers confirm this DCO drift pattern, which is conceded for power conservation and is specified as DCO *tolerance*. The exact circuit design causing this behavior is protected by intellectual property [74].

To answer the second question, we quantify the number of clock cycles that a constant clock model, that is, one that does not incorporate the changes in clock speed as the voltage drops, would not account for. Figure 4-7(b) shows that this figure increases linearly with every power cycle; more than 10k clock cycles would be unrepresented in only ten power cycles. This could be extremely critical for correctly designing and dimensioning systems that may undergo countless power cycles throughout their lifetime.

4.3.2 Modeling Dynamic Power Consumption

Besides affecting the time component of the energy equation, changes in supply voltage and clock speed also impact current draw, and hence power consumption. A precise current model is thus crucial in determining accurate power consumption, as instantaneous power consumption is a product of voltage and current. While the amount of current drawn by the MCU naturally decreases with voltage and can be calculated using Ohm's law, measuring the impact of changes in clock speed on current draw is not immediate.

In MSP430 MCUs, clock speed is mainly controlled by the DCO impedance, which is controlled using the parameters described earlier. This results in varying amounts of current drawn at different frequencies. However, the impedance of the DCO, which can be modeled as an RLC circuit, cannot be derived theoretically since the values of ohmic resistance and reactance are unknown.

Measurement procedure. Similar to the clock model, we employ an empirical approach to model the cumulative impact of changes in supply voltage and clock speed on current consumption. Our measurement setup includes a 0.1μ ampere resolution multimeter, which measures and automatically logs the current drawn by the MCU.

Since the correctness of measurements is critical to derive an accurate model, our approach also caters for the *burden voltage*—the voltage drop across the measuring instrument. We add the burden voltage (V_b) to the supply V_s. However, this leads to the current measurements of the MCU at a higher voltage (V_b+V_s), whereas we need the current draw precisely at V_s. As we know the values of V_b and the the current draw (I_m), we calculate the resistance (R_i) of the measuring instrument. We then simply calculate the current draw of the MCU as $I_{mcu} = I_m - \frac{V_b}{R_i}$.

Empirical model. Unlike the sensitivity of clock to changing supply voltage in Figure 4-6(a), the current's sensitivity to changes in clock speed and supply voltage is quite consistent and above 100% for a large fraction of frequencies, as shown in Figure 4-6(b).

The inconsistent behavior for less than 20% observations is due to DCO's unstable behaviors for very slow frequency configurations (below 1 MHz), which are typically neither calibrated nor used with MSP430 MCUs [70]. Figure 4-6(c) highlights the multiplicative impact of changes in supply voltage and current on an MCU's power consumption. Power consumption may vary as much as 3.5× within a single power cycle. This demonstrates that existing tools, as they fail to model such behaviors, tend to be provide inaccurate inputs to the design and dimensioning of TPCs.

4.3.3 Model Representation

We consider two choices to represent the results of our measurements for either model. Either we use a lookup table or train a model with RSELx, DCOx, MODx and supply voltage (V_s) as inputs. The former is of course, exhaustive, but unlikely to fit on an embedded MCU with little main memory, for example, to be used at run-time to implement energy-adaptive behaviors [23, 28]. For instance, a large lookup table with a precision of three decimal places and a small lookup table with a precision of one decimal place would consume 28 MB and 288 KB in main memory, respectively.

We thus also explore the derivation of a compact model based on linear regression able to fit within the limited memory budget of embedded MCUs. We observe that

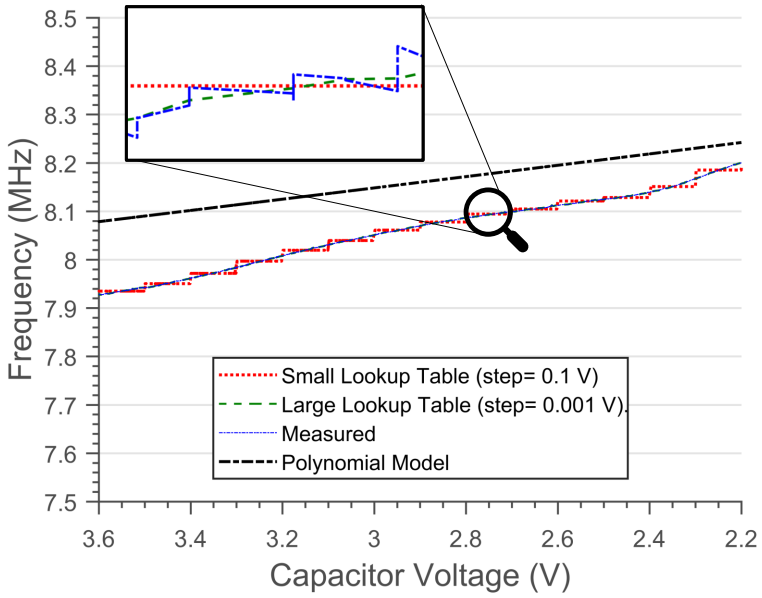


Figure 4-8: Model representations and their behavior compared to empirical measurements @ 8 MHz. *Higher precision lookup tables follow closely the actual measurements. The polynomial model rests within a 2% error bound.*

a degree 7 polynomial is sufficient to fit a model with error bound to ±6%. This can readily be reduced to ±1% with a degree 3 polynomial when used for common DCO frequencies such as 1 MHz, 8 MHz, and 16 MHz.

Figure 4-8 highlights the behavior of these different representations of the clock model during a single power cycle. A large lookup table with 0.001V precision accurately follows the measurements, whereas a small look up table with 0.1V precision predictably mimics a step function. The polynomial model, in this particular setting, achieves an average error below 2%.

What model representation to employ is, therefore, to be decided depending on desired accuracy and intended use. Compile-time or off-line analysis may use the lookup table representation, which faithfully describes the results of our measurements. Whenever the models are to be deployed on an embedded MCU with little memory, the polynomial model is way easier on memory consumption.

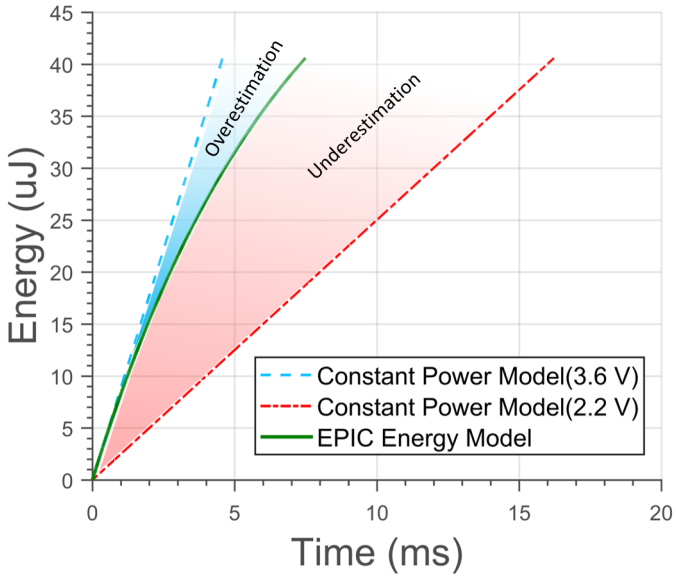


Figure 4-9: Dynamic vs. constant energy models @ 8 MHz. *Depending on supply voltage, a constant model may overestimate (shaded blue region) or underestimate (shaded red region) the energy available with a single discharge of a 10μF capacitor.*

4.3.4 Summary of Findings

We conclude that modeling these dynamic behaviors is essential for designing and dimensioning TPC systems, as accounting for them may lead to improved performance. Before providing quantitative evidence of this in Section 4.4 and Section 4.5, we reason about the general impact of the dynamic model described above when compared with a range of assumed constant models in Figure 4-9.

The green line in the middle shows the accurate prediction of energy consumption during a single power cycle by our model. The shaded blue region represents the range of constant models that overestimate energy consumption [18,33,40,130]. Most existing tools lie on the left-most, dashed blue line as they are designed for worst-case analysis, namely, they compute energy consumption by assuming a constant input voltage at 3.6V. Systems designed and configured with these models may significantly under-perform; for example, an overestimation of energy consumption may lead to overly conservative placements of expensive checkpoint calls in the code [18, 130], as we demonstrate in both Section 4.4 and Section 4.5.

Differently, the shaded red region below the green line shows the range of constant

models that would underestimate energy consumption, by considering a constant lower voltage within the MCU's operational range. TPC systems based on such models may fail; for example, an underestimation of energy may lead to non-termination in task-based systems, where a wrongly-defined task may require more energy than a capacitor can store [33].

4.4 EPIC

Based on our energy models, we develop EPIC: a compile-time analysis tool that accurately predicts energy consumption of arbitrary code segments. Existing solutions employ time-consuming laboratory techniques [18], and yet they overlook the impact of variations in supply voltage. EPIC provides this information in an automated fashion and by explicitly accounting for such dynamics.

4.4.1 Workflow

EPIC is implemented in two Java modules, the *mapper* and the *analyzer*, as shown in Figure 4-10.

Mapper. Inputs for the mapper module are the empirical energy model, a portion of the source code marked by developers for analysis, and the corresponding assembly code generated for a specific platform. Instructing the compiler to include debugging symbols in the assembly allows the mapper to establish a correspondence between each assembly instruction and the corresponding source code line [89, 144]

The mapper initially analyzes the assembly code to find the basic blocks at the level of source code that corresponds to a given set of assembly instructions. Using this mapping and information on the number of MCU cycles required by each assembly instructions, EPIC computes the total number of MCU cycles per basic block at the level of source code. Such a mapping process is non-trivial; we defer a more detailed discussion to Section 4.4.2.

Next, the mapper relies on the empirical energy model to predict the energy consumption of each basic block. The output of this step is a separate file, called *energy*

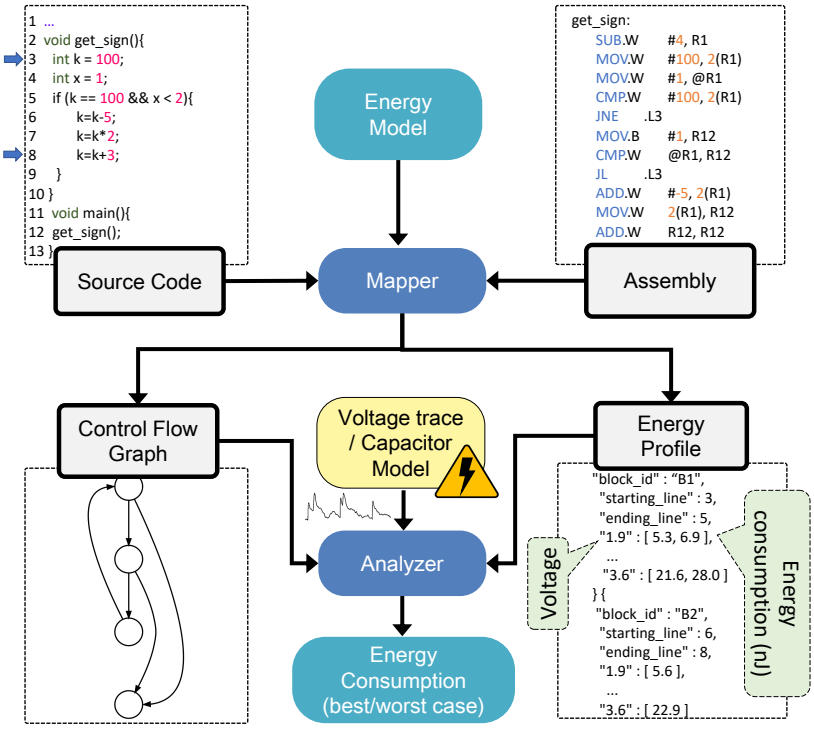


Figure 4-10: EPIC code instrumentation process. *The mapper maps the assembly instructions to the corresponding basic blocks in the source code and outputs a CFG and energy profile based on the models of Section 4.3. The analyzer traverses the CFG to compute best- and worst-case estimates for a given fragment of code and capacitor size.*

profile, which contains the energy consumption of each basic block at arbitrary supply voltages within the operational range of the target MCU. We store this information in JSON format to facilitate parsing by compile-time tools relying on the output of EPIC. Finally, the mapper also generates the control-flow-graph (CFG) of the code using ANTLR.

Analyzer. The goal of the analyzer is to determine best- and worst-case estimates of energy consumption for each node in the CFG. To that end, the analyzer takes as input the energy profile output by the mapper and a trace of supply voltage values used to determine which value to choose from the energy profile for a particular basic block. Two choices are available for this: either using a capacitor model that simulates the underlying physics and specific (dis)charging behaviors, or relying on a user-provided energy trace that indicates the state of capacitor's charge over time.

Similar to other compile-time tools, the best- and worst-case energy consumption

estimates are the best output EPIC can provide. Nevertheless, there are cases where these estimates depend on run-time information; for example, in the presence of loops whose number of iterations is not known at compile time. If a user-specified piece of code also includes any of these programming constructs, EPIC prompts the user for ascertaining the number of loop iterations to be considered in a given analysis.

4.4.2 Finding MCU Cycles per Basic Block

Established techniques exist to determine a mapping between basic blocks at the level of source code and assembly instructions, for example, as used in simulation tools such as PowerTOSSIM [144] and TimeTOSSIM [89]. These techniques, however, do not accurately handle cases where a given basic block of source code may correspond to the execution of a variable number of instructions in assembly; therefore, a single node in the CFG is translated into multiple basic blocks at the assembly level. Short-circuit evaluation, `for` loops, and compiler-inserted libraries are examples where issues manifest that may cause a loss of accuracy in energy estimates.

To handle these cases, EPIC further dissects the mapping process to extract additional information useful to reason on the energy consumption of a given basic block, as described below.

Short-circuit evaluation. These include arbitrary concatenations of logical operators, such as "&&" or "||", where the truth value of a pre-fix determines the truth value of the whole expression, independent of the truth value of the post-fix. In these cases, the assembly code generated by the compiler skips the evaluation of the post-fix part of the expression as soon as the pre-fix determines the truth value of the whole expression.

To handle these cases, the mapper reports the energy consumption of all possible evaluations of the statement involving these operators. This is shown in the energy profile of Figure 4-10 for the `if` statement at line 5 of the source code. Two separate values are reported for the corresponding basic block (B1), one for only the pre-fix and one for the complete evaluation of the expression.

This information is then useful for the analyzer module, which may choose the appropriate value depending upon the type of analysis required. For example, when considering the best-case energy consumption, the analyzer considers the energy consumption for executing the minimal pre-fix that determines the truth value. Differently, the analyzer accounts for the energy consumption of executing every sub-expression when computing the worst case.

Loops using `for`. The execution of these loops may incur a different number of MCU cycles at the first or at intermediate iteration(s). This is because the initialization of the loop variable is only performed at the first iteration, whereas all other iterations incur the same number of MCU cycles as they always perform the same operations; for example, incrementing a counter and checking its value against a threshold.

The mapper accurately identifies the set of instructions needed for these two different types of execution of the `for(;;)` statements, and reports them in the energy profile as two separate entries for the corresponding basic block. The analyzer then utilizes this information to accurately calculate the energy consumption of different loop iterations.

Compiler-inserted functions. For programming constructs that are not supported natively in hardware, compilers insert their own library functions to emulate the desired functionality in software. For example, if an MCU does not feature floating point support, the compiler automatically replaces the corresponding statements with its own assembly code. However, the execution-time is dependent on the operands of the library functions. Since there are only a handful of such library functions typically, we address this problem by profiling these with arbitrary inputs as function arguments and record their best- and worst-case energy consumption. Therefore, we avoid the problem of energy misprediction, due to varying operands, by predicting energy consumption in both, optimistic and pessimistic ways, thus helping the programmer in the analysis.

These simple design decisions form a foundation for EPIC's scalability even for very large applications. These decisions allow EPIC to only make a single pass of the

Applications	Voltage (V)	Measured Energy (μJ)	Predicted Energy (μJ)	Error (%)
Bubble Sort	2.3	0.9016	0.9014	-0.03
	2.5	1.0332	1.0354	0.21
	3.0	1.5538	1.5556	0.12
	3.5	2.1960	2.1956	-0.02
CRC	2.3	0.3709	0.3697	-0.31
	2.5	0.4267	0.4248	-0.45
	3.0	0.6389	0.6382	-0.12
	3.5	0.9027	0.9007	-0.22
FFT	2.3	12.9904	13.0194	0.22
	2.5	14.9520	14.9554	0.02
	3.0	22.4640	22.4695	0.02
	3.5	31.5952	31.7129	0.37
AES	2.3	37.3888	37.5609	0.46
	2.5	42.9240	43.1463	0.52
	3.0	64.3968	64.8247	0.66
	3.5	90.9664	91.4920	0.58

Table 4.1: EPIC's accuracy.

entire code for energy prediction therefore its run-time is always a linear function of code size. Assessing the utility of EPIC for complex architectures, such as MCU with instruction pipelines and caches, would be challenging, yet, very interesting future work.

We next evaluate the accuracy of energy profile information returned by EPIC across four benchmark applications, before employing the entire EPIC workflow for compile-time analysis in two concrete cases.

4.4.3 Microbenchmarks

We first investigate the accuracy of the energy profile information returned by the mapper module in EPIC. In turn, this depends on *i)* the accuracy of the empirical energy model described in Section 4.3, and *ii)* the effectiveness of the mapping techniques between source code and assembly code illustrated in Section 4.4.2. Understanding whether the energy profiles are accurate is a stepping stone to investigate

the use of EPIC in a concrete case study.

Our benchmarks include open-source implementations of Bubblesort, CRC, FFT, and AES, which are often employed for benchmarking embedded systems [51], and specifically system support for TPCs [11, 79, 130]. We run each application on real hardware at different supply voltages and record a per-basic-block trace of execution time and current draw to compute the energy consumption.

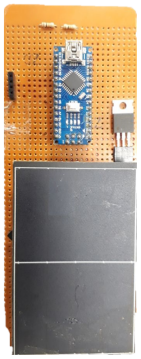
Table 4.1 compares the results returned by the mapper module of EPIC at different supply voltages with the empirically measured ones on an MSP430G2553 running at 8 MHz. The error is generally well below 1%. The results demonstrate that the information to the analyzer module is accurate, as a result of the accuracy of the empirical model and the mapping between source code and assembly we adopt.

4.4.4 EPIC with HarvOS

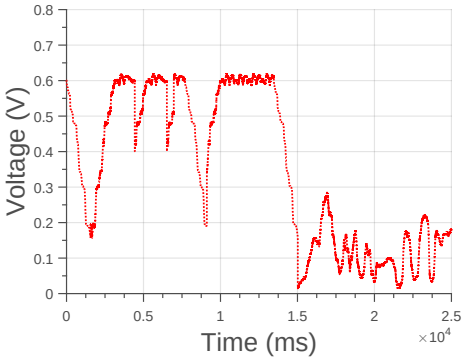
We integrate EPIC with HarvOS [18], an existing system support for TPCs. HarvOS relies on compile-time energy estimates to insert function calls—termed *trigger* calls—that possibly execute a checkpoint whenever a device is about to exhaust the available energy. The triggers include code to query the state of the energy buffer for deciding whether or not to checkpoint. Checkpoints are costly in energy and additional execution time. Nonetheless, regardless of whether a checkpoint takes place, these calls represent an overhead, as merely querying the energy buffer consumes energy [130].

In HarvOS [18], the placement of trigger calls is based on an efficient strategy that requires a worst-case estimate of the energy consumption of each node in the CFG. Similar to existing literature, HarvOS normally employs a manual instrumentation process based on a static energy consumption model, namely, overlooking the dynamic behavior of power consumption and clock speeds. We use EPIC to substitute for such manual instrumentation process.

Setup. We use two applications: *i)* an *Activity Recognition* (AR) application that recognizes human activity based on sensor values, often utilized for evaluating TPC



(a) HW setup.



(b) Excerpt of the indoor light voltage trace.

Figure 4-11: Voltage trace from indoor light using a mono-crystalline solar panel.

solutions [32, 101], and *ii)* an implementation of the *Advanced Encryption Standard* (AES), which is one of the benchmarks used in HarvOS to investigate its efficiency [18]. We execute EPIC by relying on two different types of voltage traces as an input to the analyzer module.

First, we use a fundamental voltage trace often found in existing literature [16, 79, 130, 152]. The device boots with the capacitor fully charged, and computes until the capacitor is empty again. In the meantime, the environment provides no additional energy. Once the capacitor is empty, the environment provides new energy until the capacitor is full again and computation resumes. This energy provisioning pattern generates executions that are highly intermittent, namely, executions that most paradigmatically differentiate TPCs from other embedded and mainstream platforms. Further, this profile is representative of a staple class of TPC applications based on wireless energy transfer [19]. With this technology, devices are quickly charged with a burst of wirelessly-transmitted energy until they boot. Next, the application runs until the capacitor is empty again. The device rests dormant until another burst of wireless energy comes in. We call this trace the *decay* trace.

Second, we use a voltage trace collected using a mono-crystalline high-efficiency solar panel [147], placed on a desk and harvesting energy from light in an indoor lab environment, as shown in Figure 4-11. We use an Arduino Nano [9] to log the voltage output across the load, equivalent to the resistance of an MSP430G2553 in active

	# of trigger calls		# of checkpoints		Speedup in completion
Application	Manual	EPIC	Manual	EPIC	**time** (%)
Decay Trace					
AR	2	1	85	42	51.66
AES	3	0	2	0	354.97
Light Trace					
AR	2	1	64	32	49.35
AES	3	0	1	0	203.31

Table 4.2: Features and performance of HarvOS-instrumented code with and without EPIC.

mode, attached to the solar panel. We call this trace the *light* trace.

Results. By comparing the features and the performance of HarvOS-instrumented code using the manual energy profiling technique in the original HarvOS or EPIC, based on an MSP430G2553 running at 8MHz, we draw the following observations:

1. Using EPIC, the number of trigger calls inserted by HarvOS in the original code reduces;

2. A more accurate code instrumentation due to EPIC leads to fewer checkpointing interruptions.

The first two columns in Table 4.2 show the results of the instrumentation process when using the smallest capacitor size needed to complete the given workloads. We consider this as TPCs typically prefer smaller capacitors as energy buffer, because they reach the operating voltage more quickly and yield smaller device footprints. However, if the capacitor is too small, a system may be unable to complete checkpoints, ending up in a situation where applications cannot make any progress.

For the AR application, Table 4.2 indicates that EPIC's accurate energy estimates halve the number of trigger calls that HarvOS places in the code. This reduction occurs due to EPIC's ability to accurately model the varying number of clock cycles at different voltages that a static model would not consider. As the voltage of the capacitor decreases, the speed of the clock increases. This results in more clock cycles becoming available per unit of time at lower supply voltages. Not accounting for such

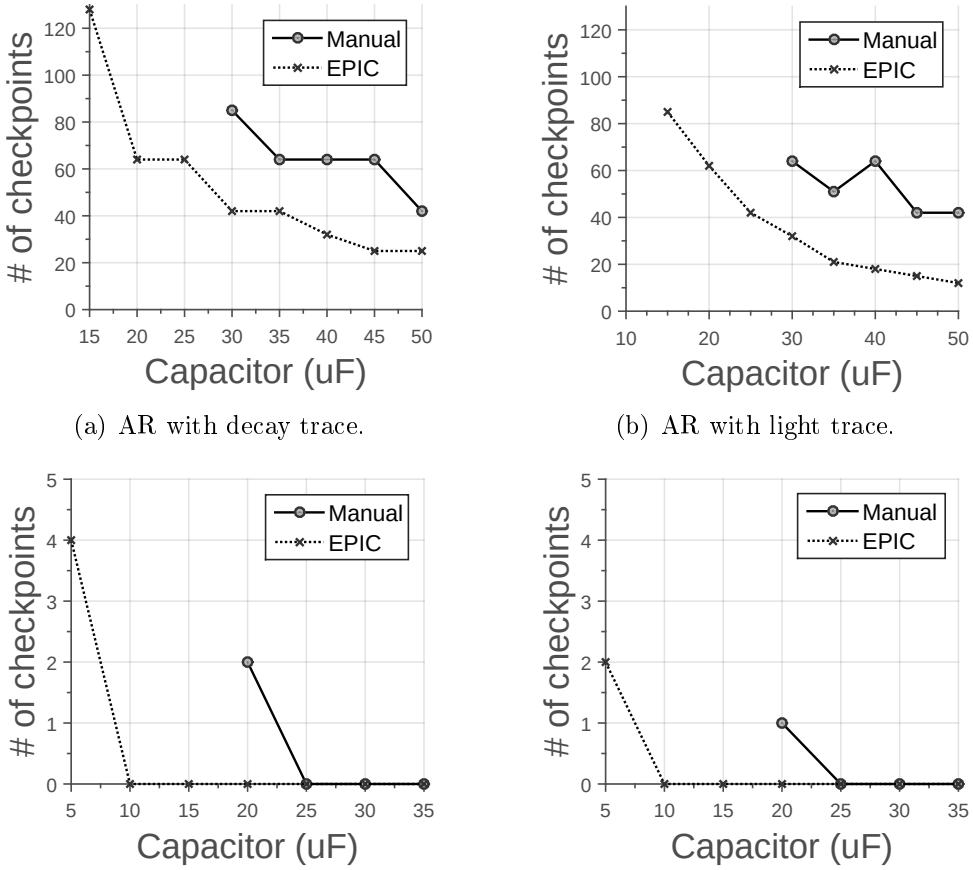
dynamic behavior significantly underestimates the number of clock cycles available within a single time unit in conditions with low supply voltages.

The results for AES instrumentation are revealing: based on the energy estimates provided by EPIC, HarvOS decides to place *no trigger calls*. This means EPIC indicates that the energy provided by the capacitor is sufficient for the AES implementation to complete in a single power cycle, and thus no checkpoints are ever necessary. This sharply contrasts the outcome of the HarvOS compile-time analysis whenever based on the assumption of constant power consumption and clock speed. In that case, HarvOS would still place three trigger calls within the AES code, uselessly incurring the corresponding overhead. This shows how not accounting for the dynamic behaviors represented by our energy model profoundly misguides compile-time analyses.

The impact of the trigger call placement with or without EPIC has marked consequences when running the instrumented applications. The third and fourth columns in Table 4.2 show the corresponding results for either voltage trace. The AR application instrumented by HarvOS based on the energy estimates of EPIC completes the execution with nearly 50% fewer checkpoints. Similarly, the successful completion of the AES implementation without a single checkpoint confirms the validity of the energy estimates of EPIC, which prompted HarvOS not to place any trigger call.

Checkpoint operations are extremely energy consuming, as they incur operations on NVM. Fewer checkpoints allow the system to spend the corresponding energy budget in useful computation cycles, which are otherwise wasted due to inaccurate insertion of trigger calls in the original HarvOS. Thus, the system progresses faster towards the completion of the workload. The right most column in 4.2 quantifies the benefits in terms of speedup of completion time for the given workload, primarily enabled by correct dimensioning of HarvOS using EPIC.

Figure 4-12 further investigates the execution of either application at increasing capacitor sizes. These results affirm that the benefits of using EPIC within HarvOS are not just limited to the smallest capacitor that ensures completion of a given code. The apparent outlier at 40uF in Figure 4-12(b) is due to a specific behavior of HarvOS



(a) AR with decay trace.

(b) AR with light trace.

(c) AES with decay trace.

(d) AES with light trace.

Figure 4-12: HarvOS results. *The benefits of using EPIC within HarvOS apply across different capacitor sizes.*

whenever larger capacitors simultaneously yield a change in the placement of trigger calls and in their overall number [18]. For the AR application, the corresponding speedup in completion time range from 14% at 35uF with the decay trace, to 159% at 40uF using the light trace. For the AES implementation, the figures in Table 4.2 already show the performance range, as no checkpoints are needed using capacitors larger than 20uF.

Finally, Figure 4-12 also shows that in a number of situations, the compile-time instrumentation generated by HarvOS when using the energy estimates of EPIC yields an operational system at much smaller capacitor sizes, as compared with the original HarvOS. The cost for the overly-conservative estimations in the latter, based on static power consumption and clock speeds, materializes in the inability to make

any progress in these applications when using small capacitors. In contrast, EPIC captures the dynamic behavior of these figures and offers accurate estimations to HarvOS; this results in more informed decisions on trigger call placement and on whether to checkpoint when executing a trigger call.

4.4.5 EPIC with CleanCut

An alternative to using automated placement of trigger calls is to employ task-based programming abstractions offering transactional semantics [32, 101, 104]. Programmers are to manually define tasks that are guaranteed to either complete by committing their output to NVM or to have no effect on program state.

CleanCut [33] is a compile-time tool that helps programmers using these abstractions identify non-termination bugs. These exist whenever a task definition includes execution paths whose energy cost exceeds the maximum available energy, based on capacitor size. In these cases, if no new energy is harvested while the task executes, that may never complete and thus the program ends up in a livelock situation, always resuming from the task beginning.

CleanCut relies on an energy model obtained through hardware-assisted profiling. The model estimates the energy consumption of each basic block at near maximum voltage supply, to avoid underestimations. The estimates are then convoluted across the possible execution paths in a task to find energy distributions for individual tasks. Based on this, CleanCut returns warnings whenever it suspects a non-termination bug. Programmers must then defend against these; for example, by refactoring the code to define shorter tasks. This is not just laborious, but also detrimental to performance, as every task boundary incurs significant energy overhead due to committing a task's output on NVM.

Colin et al. [33] argue that an analytical model may provide more accurate warnings, but was out of scope. Using EPIC with CleanCut, we prove this argument. For long execution paths in a task that use the capacitor to near depletion, the difference between the actual used energy—accurately modeled by EPIC—and CleanCut's estimations obtained as described above, can be significant. This results in false posi-

Application	Task boundaries	Paths	CleanCut warnings	EPIC warnings
AR	4	147	**8**	**0**
RSA	4	8	**3**	**0**
CF	11	25	0	0
CEM	11	20	0	0

Table 4.3: CleanCut non-termination bug warnings with and without EPIC.

tives, that is, non-termination bugs are suspected for paths that may safely complete with the given energy budget. Such warnings simply do not appear using EPIC.

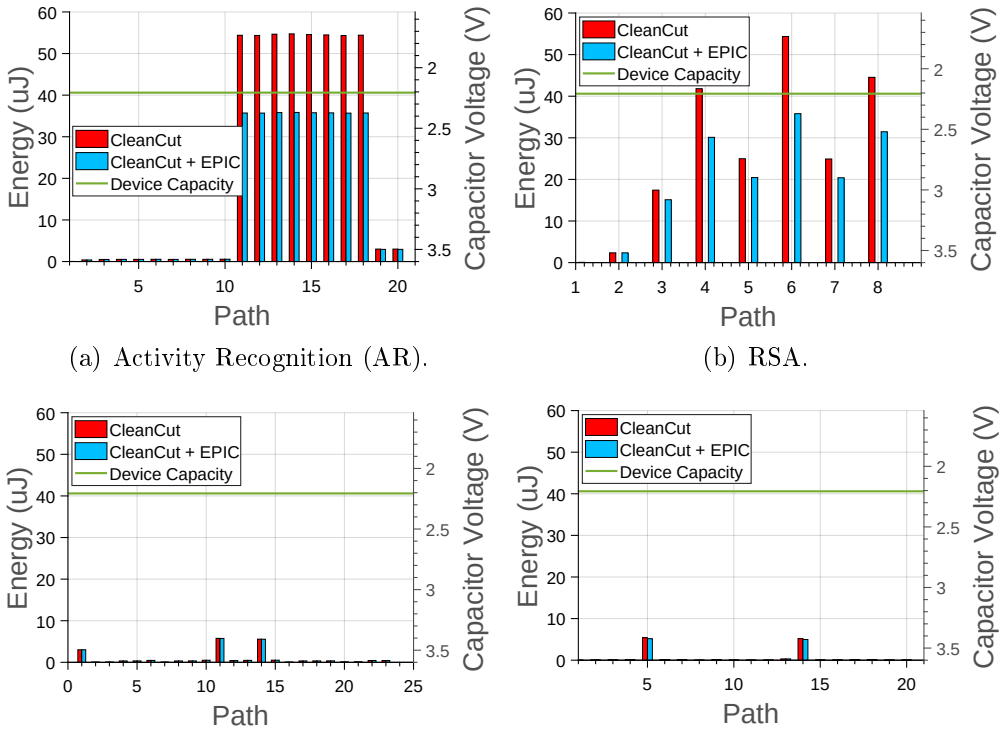
Setup. We use the same applications as in CleanCut [33]: *i)* an *Activity Recognition* (AR) application similar to Section 4.4.4, *ii)* a *Cuckoo Filter* (CF) that efficiently tests set membership, *iii)* a *Coldchain Equipment Monitor* (CEM) application, and *iv)* an implementation of the *RSA algorithm*.

The placement of task boundaries is the one of CleanCut [33]. We then estimate the energy consumption of every possible execution path in a task using CleanCut with EPIC, compared with CleanCut using a synthetic model that safely approximates that of CleanCut, whose hardware and source code are not available. This model follows the recommendation that the energy consumption of each basic block should be estimated near the maximum voltage to avoid underestimations [33].

We use a 10μF capacitor, which can be found on Intel WISP 4.1 devices, and consider an MSP430G2553 running at 8 MHz.

Results. Table 4.3 summarizes our results, which are further detailed in Figure 4-13. By comparing how CleanCut and CleanCut using EPIC build up the energy estimates for a task, we note that the two start identical, but as paths become longer and nears the capacitor's limit, CleanCut starts to overestimate. This is because it uses the same energy model for every basic block, regardless of where it appears on the path.

The results for the AR application in Figure 4-13(a) indicate that CleanCut returns eight false positives, in that it estimates the energy consumption of those paths to exceed the available capacitor energy. Developers would then need to break those tasks in smaller units, investing additional design and programming effort, and caus-



(a) Activity Recognition (AR).

(b) RSA.

(c) Cuckoo Filter (CF).

(d) CEM.

Figure 4-13: CleanCut results. *Using EPIC allows CleanCut to avoid false positives when looking for non-termination bugs.*

ing increased overhead at run-time due to more frequent commits to NVM at the end of shorter tasks. This is not the case with CleanCut using EPIC, which verifies that the same execution paths may complete successfully. This means the task placement includes no non-termination bugs; therefore, developers need not to spend any additional effort and the system runs with better energy efficiency. Similar considerations apply to the case of the RSA algorithm, wherein CleanCut returns three false positives, as shown in Figure 4-13(b).

In the case of CF, processing is generally lighter compared to AR and RSA. Further, the existing task definition includes very short tasks already. As a result, the estimates of CleanCut and CleanCut using EPIC are close to each other, as shown in Figure 4-13(c) No warnings for possible non-termination bugs are returned in either case. Similar considerations apply also to the CEM application, whose results are shown in Figure 4-13(d).

CleanCut also provides a task boundary placer that automatically breaks long tasks. We cannot evaluate the impact of EPIC there, since the implementation is not available, but we argue that the benefits would be even higher. In fact, CleanCut's placer is actively trying to bring task definitions to their optimal energy point, that is, closer to the capacitor limit. This is precisely where the difference in estimates between CleanCut and CleanCut with EPIC is maximum. Thus, the probability of paths landing in the region where CleanCut declares a task to be too long, but CleanCut using EPIC says the opposite, is likely higher compared to checking a manual placement.

4.5 MSPsim++

While compile-time analysis is important early on, testing down the road may necessitate cycle-accurate emulation to understand the run-time behavior.

We demonstrate the role of our energy models in this context within MementOS [130], an existing system support for TPCs that heavily relies on emulation to determine when a checkpoint is to be taken. Enabling this investigation is MSPsim++, an extension to the existing MSPsim emulator [40] for MSP430 MCUs. Our analysis reveals the enormous discrepancies that might arise in the performance of MementOS-supported applications by disregarding the dynamic behaviors of power consumption and clock speed.

4.5.1 Design

MSPsim is a widely used Java-based instruction level emulator for MSP430-based platforms. Its energy prediction model is based on static power consumption and clock speed. We replace these with our own energy model, described in Section 4.3, and refer to this extension as MSPsim++. The capacitor model we use is borrowed from MementOS [130].

Computing the energy figures during an execution proceeds as follows. At the conclusion of each instruction execution, MSPsim++ subtracts the energy consumed

by its execution from the capacitor. Next, it calculates the new supply voltage based on the capacitor model, and uses the new supply voltage value as the index into a table-based model representation at 0.1V resolution, as explained in Section 4.3.3. With the updated power consumption and clock speed, MSPsim++ executes the next instruction. This means MSPsim++ emulates the dynamic behaviors of supply voltage, power consumption, and clock speed on a per-instruction basis. In an instruction-level emulator, this is the finest possible granularity.

Cycle-accurate emulation does not suffer from the potential inaccuracies in mapping source code to assembly and back, described in Section 4.4.2. MSPsim++ executes the exact same binary instructions the real hardware executes. Therefore, our model integration in MSPsim++ is inherently more accurate than then compile-time results illustrated in Table 4.1.

4.5.2 MSPsim++ with MementOS

MSPsim++ replaces the original MSPsim in MementOS. Using MSPsim++, we seek to understand the impact of modeling the dynamic behaviors of power consumption and clock speed on MementOS-supported applications.

Setup. MementOS uses checkpointing to protect a program from state losses due to energy depletion. The decision to checkpoint is based on a voltage threshold obtained through repeated emulation experiments and user-supplied voltage traces. These experiments use MSPsim to try and execute a MementOS-supported program with progressively decreasing thresholds to trigger checkpoints. This procedure returns the lowest threshold leading to completion of the program, with what is estimated to be the minimum number of required checkpoints.

Our two benchmarks use the same code base as the original MementOS [130], and are[3]: *i) Cyclic Redundancy Check* (CRC) that computes a CRC16-CCITT checksum over a 2 KB region of flash memory, and *ii) RSA cryptography* that uses iterative

[3]The results in [130] are not repeatable because neither the specific hardware platform nor the tools used for integration of MementOS with MSPsim are supported anymore. We contacted the hardware supplier and the MementOS developers, who encouraged us to apply the corresponding upgrades in the hardware and software tool-chain.

left-to-right modular exponentiation of multiple-precision integers to encrypt a 64-bit message under a 64-bit public key and 17-bit exponent. In both cases, the results we discuss next use the "function call" strategy to inline the calls that possibly checkpoints [130]. We obtain similar results using other MementOS strategies. While we only use two benchmarks for this evaluation, we observed during our experiments that adding more applications did not offer any newer insights and the results were redundant.

We analyze the behavior of MementOS using either MSPsim or MSPsim++. We expect the voltage threshold determined at the end of these emulation experiments to be significantly lower using MSPsim++, since it recognizes the gain in clock speed and reduction in power consumption as the voltage supply drops. We use the same decay and light traces used in Section 4.4.4 as input.

Results. The outcome of our investigation leads us to the following observations when using an MSP430G2553 running at 8MHz:

1. MSPsim++ causes MementOS to use lower voltage thresholds to trigger a checkpoint;

2. Lower voltage thresholds reduce the run-time checkpointing overhead, shifting the energy budget to useful computations.

The two left-most columns in Table 4.4 are obtained by considering the smallest capacitor size where MementOS using MSPsim or MSPsim++ is operational.

The voltage thresholds confirm our hypothesis; as the original MSPsim does not model the dynamic behaviors of power consumption and clock speeds, the corresponding thresholds are overly conservative. Intuitively, with this configuration, MementOS would trigger a checkpoint too early: the execution might continue and a checkpoint be triggered later, once the supply voltage dropped further, and still complete successfully. In contrast, MSPsim++ returns lower voltage thresholds due to a more accurate modeling of energy consumption. A lower threshold means that part of the energy budget may be shifted from checkpointing operations to useful computations, as the checkpointing can be delayed until the capacitor voltage reaches a lower value.

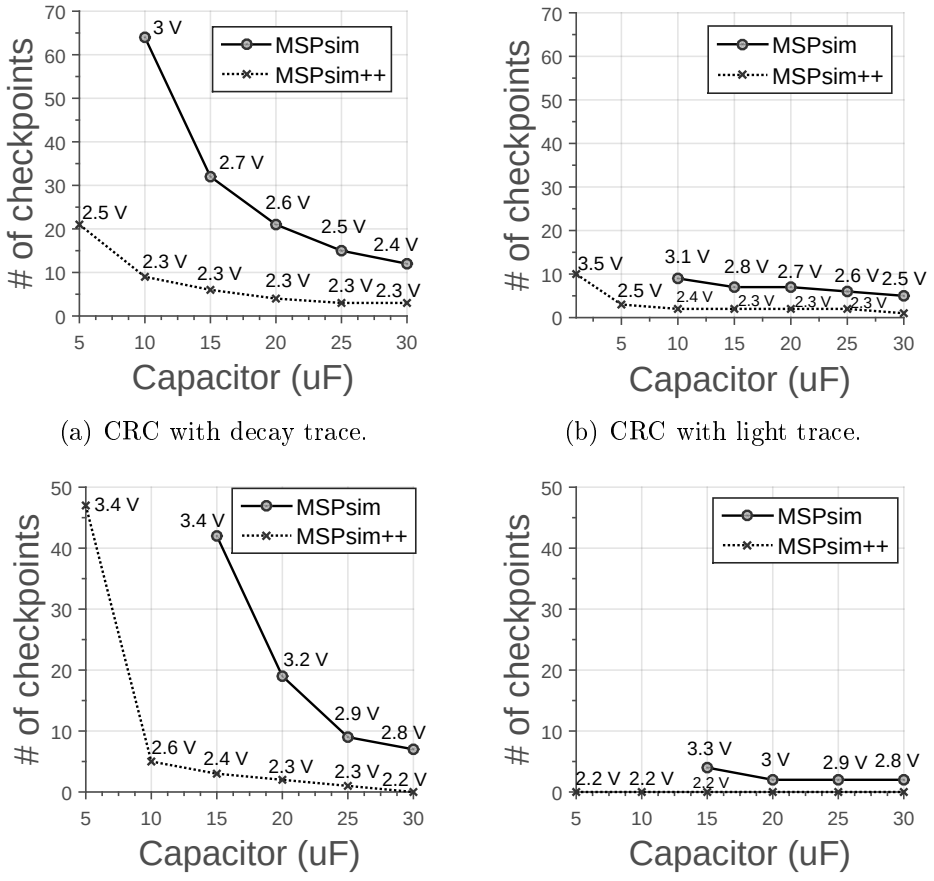
	Vthresh (V)		# of checkpoints		Speedup in completion time (%)
Apps	MSPsim	MSPsim++	MSPsim	MSPsim++	
Decay Trace					
CRC	3.0	2.3	64	9	392.88
RSA	3.4	2.4	42	3	152.76
Light Trace					
CRC	3.1	2.4	9	2	715.16
RSA	3.3	2.2	4	0	1037.43

Table 4.4: Parameter setting and performance of MementOS applications with MSPsim or MSPsim++.

This behavior shows in the the next three columns of Table 4.4. Lower thresholds enable the system to complete the application execution with up to *one order of magnitude* fewer checkpoints. As a result, available energy is rather spent for useful computations, thus making faster progress towards the completion of the workload. The corresponding speedup reaches more than 1000%.

Figure 4-14 confirms that these improvements apply across different capacitor sizes. The difference in voltage thresholds is less pronounced as the capacitor size increases. A little analogy may help: say a capacitor is akin to a water reservoir, and voltage represents the vertical level of water. With bigger capacitors (reservoirs), even a small difference in voltage may correspond to a large amount of energy (water). Thus, the difference in voltage thresholds derived using MSPsim or MSPsim++ shrinks. In the tests of Figure 4-14, for the CRC implementation, the minimum speedup in completion time is 51% at 30uF with the light trace, whereas 4.4 shows the maximum. For the RSA implementation, the speedup range from 279% at 25uF with the decay trace, to 1264% at 30uF with the light trace.

Similar to the investigation of EPIC within HarvOS in Section 4.4, Figure 4-14 shows that the energy estimates of MSPsim++ allow MementOS to become operational using smaller capacitor sizes. Similar considerations as in Section 4.4.4 apply here.



(a) CRC with decay trace.

(b) CRC with light trace.

(c) RSA with decay trace.

(d) RSA with light trace.

Figure 4-14: MementOS results. *Across different capacitor sizes, MSPsim++ constantly indicates lower voltage thresholds than MSPsim.*

4.6 Discussion

We demonstrate that static models for estimating energy consumption of TPCs lead to overly-conservative designs and parameter settings, which yield sub-optimal performance. We provide next due considerations on how our work is cast in the larger intermittent computing domain.

MCUs and peripherals. We focus on the MCU as it coordinates the functioning of the entire system when using the emerging federated energy architectures [56]. For the MCU, accurately forecasting the energy cost of a certain fragment of code is key to dimensioning capacitors and setting its running frequency. Peripheral operation may be postponed when the energy is insufficient, or the system may impose atomic

executions on peripheral operations [32, 104]. Dedicated works exist that ensures the correct intermittent operation of peripherals [21, 102, 146, 156].

We also model the active mode of the MCU as this is the only mode where the MCU executes the code [64]. TPCs primarily use this mode to maximize throughput during power cycles that may be as short as a few ms. Other low-power modes are typically used in battery-powered platforms for conserving energy when idle.

Voltage regulation. Voltage regulators are commonly found in computer power supplies to stabilize the supply voltage. Despite the availability of efficient voltage regulators with minimal dropout [72], they are typically not employed in TPCs [56] because step-up regulators reduce the power cycle duration and step-down regulators can critically fail some on-chip components.

When deploying Mementos [130] on the voltage-regulated WISP platform, the authors report a 50% reduction in the duration of power cycles when using a 2.8V step-up regulator, and checkpointing failures with a 1.8V step-down regulator due to failing to meet the voltage requirements of flash memory. An open research question is what are the conditions, for example, in terms of energy provisioning patterns, where the trade-off exposed by dynamic voltage regulation play favorably.

4.7 Related Work

Static models for supply voltage, power consumption, and clock speed tend to prevail in existing literature. This is particularly true for compile-time tools, simulators, and emulators. These are often used as input to other systems [18, 130] or to guide the programming activities [32, 101, 104]. The influence of inaccurate models thus percolates down to the run-time performance.

High-level simulation. Popular network simulators, such as ns3 and OMNeT++, may employ various energy harvesting models [13, 138, 151]. These tools are, however, unable to capture the node behavior in a cycle-accurate manner and rather rely on simple approximations, such as coarse-grain estimations of a node's duty cycle, to enable analysis of energy consumption. These approximations do adopt the assumption

of static voltage supply.

Alizai et al. [4] extend PowerTOSSIM [144] with a framework that can be plugged-in with different types of energy harvesting models to simulate networked TPCs. They rely on existing models of PowerTOSSIM, which are derived from static power consumption measurements of different device components in different operational modes. This would yield inaccuracies akin to Section 4.5.

Cycle-accurate emulation. SensEH [36] extends the COOJA/MSPsim framework with models of photovoltaic harvester, obtained using sunlight and artificial light traces from a sensor network deployment inside a road tunnel. The authors explicitly mention the use of a static power consumption and clock speed models. As it is also based on MSPsim, SensEH may benefit from our work on MSPsim++ as much as MementOS.

There exist numerous similar efforts for emulating the behavior of energy harvesting in different environments [27, 38, 109, 111]. Allen et al. [5] compare many of these with each other and discuss their limitations with regard to the representation of energy harvesting dynamics and power consumption modeling. They also emphasize the need for more accurate modeling, simulation, and emulation techniques for TPCs, which we provide here.

Hardware emulation. To improve the accuracy of pre-deployment analysis, existing works explore the use of direct hardware emulation for TPCs. For example, Ekho [54] is a hardware emulator capable of recording energy harvesting traces in the form of current-voltage surfaces and accurately recreating those conditions in the lab. This allows developers to generate harvesting-dependent program behaviors. Ekho is also shown to integrate with MSPsim to serve as input to its capacitor model [45] . This integration would suffer from the same issues we point out in Section 4.5 and would similarly benefit from MSPsim++.

Custom hardware debuggers [31] for TPCs also exist. Such tools offer the highest accuracy due to their direct installation on the target hardware platform. However, they do not offer the level of convenience and automation desired at the early stages of development. Ideally, accurate compile-time analysis tools such as EPIC and high-

fidelity emulators like MSPsim++ should complement in-field debugging.

4.8 Summary

We demonstrated that it is practically possible to capitalize on the dynamic energy consumption patterns of TPCs. We experimentally built an accurate energy model, accounting for variations in power consumption and clock speed. The instruments we used to quantify the impact of these models are: *i)* EPIC, a compile-time tool based on our energy model that, when used with HarvOS or CleanCut, allows the former to achieve up to 350% speedup in workload completion times, while avoiding unnecessary program changes that ultimately hurt energy efficiency with latter, and *ii)* MSPsim++, an extension to the popular MSPsim emulator that allows us to show inaccurate parameter settings in the original MementOS, resulting in more than 1000% speedup in workload completion times.

Based on the evidence we collect with EPIC and MSPsim++, we conclude that it is possible to account for the dynamic behaviors of energy consumption without sacrificing simplicity of analysis. In both cases, the modifications to existing systems were limited to the energy estimation tool, and did not impact other functionality.

Chapter 3 and 4 conclude the discussion on checkpoint size reduction and its precise triggering methods to achieve energy-efficient checkpointing of program-state. While this shifts a significant chunk of energy for program execution, we explore a novel hardware design, in chapter 5, to maximally utilize the available energy, by performing maximum computations possible.

Chapter 5

Fast Program Execution

In the previous chapters, we presented a detailed design and evaluation of our checkpointing techniques and their triggering mechanisms. However, our contribution so far has been focused on reducing the energy spent on checkpointing; whether it is the energy spent on an individual checkpoint or the energy spent on redundant checkpoints. While it improves energy-efficiency, it is equally important to maximize the computational progress in the given energy budget in order to ensure efficient utilization.

Recently proposed non-volatile hardware support for program-state-retention requires the device to be operated on low and fixed frequency, in order to ensure energy-efficient execution; they offer wider voltage range for operation and require low current consumption thus allowing longer operation on the same charge. However, lower frequencies have larger cycle length which result in higher energy-per-cycle consumed and increased completion time of the application. On the other hand, higher frequencies, although providing reduced completion time and energy-per-cycle consumption, can only operate in a limited voltage range.

We argue that it is still feasible to execute programs on a volatile system, along with checkpoints; by allowing the system to operate on higher frequencies thus increasing the clock cycles executed in an active epoch. We observed that dynamically switching, the frequency and voltage, to the best configuration in a given voltage range allows the device to perform more number computations, when compared to

the fixed frequency configuration. However, existing hardware support available to these TPC does not allow dynamic switching of frequency and voltage under variable energy supply. Statically setting their clock frequency fails to achieve energy efficiency, as the setting remains oblivious of fluctuations in capacitor voltage and of their impact on a microcontroller operating range. To this end, we developed a run-time technique, D^{2}VFS, to intelligently regulate supply voltage and accordingly reconfigure clock frequency of intermittently-computing devices. D^{2}VFS captures these dynamics and places the microcontroller in the most efficient configuration by regulating the microcontroller supply voltage and changing its clock frequency.

Our evaluation shows that D^{2}VFS increase clock cycles per active epoch, by up to 9×, increasing the amount of computations performed within a single capacitor charge. This offers two-fold improvement in the number of required checkpoints, as we allow the system to complete a larger portion of the workload before a checkpoint is required. Furthermore, It enables 300% shorter completion times for a given workload, increasing system's responsiveness, despite the additional overhead of D^{2}VFS. This translates into, up to one-sixth smaller energy buffer to complete the same workload, cutting the time to reach the operating threshold voltage and enabling smaller device footprints.

The remainder of this chapter is structured as follows. In Section 5.1, we motivate the problem. Section 5.2 introduces the design space, presents the background and challenges associated with a discrete dynamic voltage and frequency scaling technique; D^{2}VFS. We describe D^{2}VFS's architecture and implementation details in Section 5.3. Section 5.4 thoroughly evaluates D^{2}VFS regarding reduction in number of checkpoints and completion time. We also evaluate D^{2}VFS in a real-world setting and results from our evaluation are discussed in Section 5.5. Finally, we discuss prominent related work in Section 5.6 before concluding the chapter in Section 5.7.

5.1 Motivation

Recent efforts have introduced new hardware designs to alleviate checkpoints from the energy cycle of a TPC. However, as discussed in Section 2.5.3, these designs incur high energy cost during program execution; losing the benefit gained by reduction in checkpoint size.

On the other hand, volatile systems offers high efficiency in terms of processing speed and data accesses [67]. A key limitation in employing these system lie in the volatility of program-state which has to be retained across periods of energy unavailability. Specialized software support is required to sustain computations in such intermittent settings, for example, using checkpoints [10,18,130] or programming abstractions with transactional semantics [32, 101, 104]; a burden on the constrained energy budget.

By alleviating checkpoints or reducing its energy, both systems aim to maximize the energy available for program execution thereby allowing the TPC to make maximum computational progress. We observed that it possible to maximize the computational progress, in an active epoch, on a volatile system along with checkpoints. By dynamically switching to MCU frequency as per the incoming energy, TPC can perform more computations compared to existing hardware designs; allowing TPCs to reduce the number of checkpoints required to complete the given workload in a fixed energy budget.

5.1.1 Challenge and Opportunity

Opportunity. Energy efficiency of TPC depends on both the current capacitor voltage and processor speed [2]. Compared to battery-powered devices, capacitor voltage drops way more rapidly as the device extracts energy from it. Microcontroller units (MCUs) configured with static clock frequency fail to react to these changes, compromising energy efficiency.

As shown in Figure 5-1, higher frequencies are more energy efficient, but limit the MCU operation in a narrow interval of the microcontroller's supply voltage values.

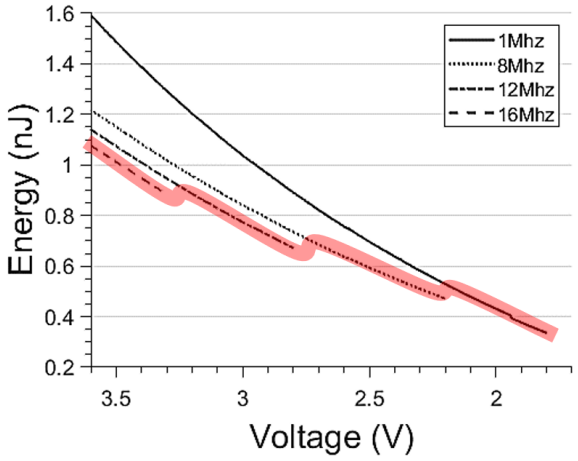


Figure 5-1: Variations in energy consumption per clock cycle with varying supply voltages. *The graph plots the energy consumption for the most commonly used and factory calibrated frequencies of MSP430G2553 MCUs. Higher frequencies are energy efficient but only operate in a limited range of supply voltage. Lower frequencies consumer more energy but enjoy a larger supply voltage range. The highlighted region indicates the best performing clock setting in a certain voltage range.*

Differently, lower frequencies consume more energy due to longer duration of clock cycles, but enable a larger range of supply voltage values. For instance, the popular MSP430-series MCUs run with supply voltages as low as 1.8 V at 1 MHz, but are unable to run any lower than 3.3 V at 16 MHz.

Nonetheless, MCUs using higher clock frequencies tend to operate in short bursts, inducing higher overhead needed to sustain computations across power cycles, for example, when checkpointing the application state on NVM [130]. Configuring MCUs with lower clock frequencies does not solve the problem either, as fewer cycles are executed per unit of energy, besides reducing processing speed.

Challenges. Employing a technique to dynamically switch voltage and frequency configuration for TPCs is, though, not straightforward. MCUs employed for intermittently-computing platforms lack hardware support for monitoring and controlling supply voltage.

Energy constraints are hard, imposing firm restrictions on the power dissipated by any additional hardware to be deployed to that end, such as voltage detectors and regulators. The former are needed to detect when the voltage crosses marked

boundaries that could trigger frequency scaling, whereas the latter are necessary for undervolting the MCU at the minimum required voltage given the operating frequency. Nonetheless, the narrower the distance between these voltage boundaries, the more fine-grained frequency scaling can become, but the higher is the overhead of voltage detection and regulation.

Thus, we are to identify a sweet spot in this cost-benefit spectrum. In addition, frequency scaling incurs processing overhead, as the software must read the frequency configurations provided by MCU manufacturers from some form of flash memory and update appropriate clock registers. This overhead needs to be minimized as well.

5.1.2 Major Contributions

To cater for the unique needs of TPCs, we present a D^{2}VFS[1] technique that seeks to take advantage from the energy consumption dynamics of TPCs [2], as we further discuss in Section 5.2. In Figure 5-1, D^{2}VFS dynamically reconfigures the system to make it proceed through the highlighted route of clock frequencies, harnessing the maximum range of supply voltage values, and yet selecting the highest possible frequency in a given range. This effectively makes the system operate in the most efficient configuration given a certain capacitor voltage. Our approach gradually de-accelerates the MCU to maximize the number of available clock cycles, and hence the computations performed in an active epoch.

To achieve maximum computational progress, the design of D^{2}VFS integrates following contributions.

1) Unlike existing systems [79, 100, 152], D^{2}VFS dynamically switches frequency and voltage configurations at discrete steps, as per the changing energy condition of the buffer; executing instructions at a higher frequency when the buffer is charged and reconfiguring the system to lower frequency when the energy level gets low.
2) To cope with lack of hardware support, D^{2}VFS employs a novel and energy-efficient hardware-software co-design, as described in Section 5.3. Our design enables min-

[1]Discrete Dynamic Voltage and Frequency Scaling

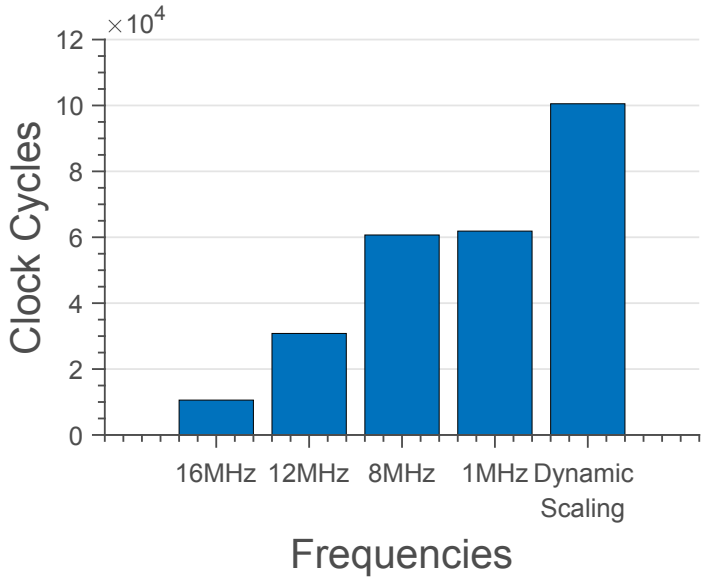


Figure 5-2: First order approximation of clock cycles for static configurations (1 MHz, 8 MHz, 12 MHz, and 16 MHz) and dynamic frequency scaling within a single charge of the capacitor (10 μF). *Dynamic scaling offers a substantial margin for improvement. The plot does not account for the overhead to enable dynamic scaling.*

imal energy overhead on the energy buffer; allowing us to reap maximum benefit from our approach.

3) Even with additional hardware employed, D^{2}VFS reduces the peak energy demand, thus requiring smaller energy buffers; *up to one-sixth* in the size of the energy buffers necessary for completing a given workload, cutting charging times and enabling smaller device footprints.

5.2 Overview

We first present relevant background on dynamic voltage and frequency scaling (DVFS). We then scrutinize the potential benefits of employing DVFS in intermittently-computing devices, before highlighting the key challenges in preserving those benefits in a practical implementation.

Background. DVFS is a combination of two power saving techniques, i.e., voltage scaling and frequency scaling, originally meant to conserve battery in mobile devices. It configures the processor to work in different *operational zones* that are defined by a

tuple consisting of a frequency f and a voltage V ($\{f,V\}$). This type of power saving is different from standby or hibernate power states, as it allows devices to continue performing tasks with a reduced amount of power. The technology is used in almost all modern computer hardware to improve battery life while still maintaining ready-to-compute performance.

The power dissipated per unit of time by a processor chip is $P = C \cdot V^2 \cdot A \cdot f$, where C is the capacitance being switched per clock cycle, V is voltage, A is the activity factor indicating the average number of switching events undergone by transistors in the chip, and f is the switching frequency. Voltage therefore dominates power. The voltage required for stable operation is, however, determined by the frequency at which the processor is clocked and can be reduced if the frequency is also reduced. Modern computers allow software control of supply voltages and frequency, but embedded processors may require hardware modifications, such as voltage regulators, to do so.

Scrutinizing DVFS. The benefits of DVFS are well understood in mainstream computing. Intuitively, it also appears to be an attractive proposition for intermittently-computing devices, where capacitor voltage uncontrollably straddles across a microcontroller's operational range.

As a first order approximation of the benefits of DVFS in this domain, using existing models [2] we calculate the number of clock cycles accumulated in a single active epoch for the dynamic frequency route highlighted in Figure 5-1, but without accounting for additional overhead required to enable this functionality. Figure 5-2 depicts the results, which are encouraging compared to the most common static frequency configurations for the MSP430 platform. Ideally, the increase in the number of available MCU cycles ranges from 60% to *one order of magnitude*, as we stretch our comparison from the lowest (1 MHz) to the highest (16 MHz) configurable frequency. These approximate results support our hypothesis that DVFS for intermittently-computing devices is indeed worth a try.

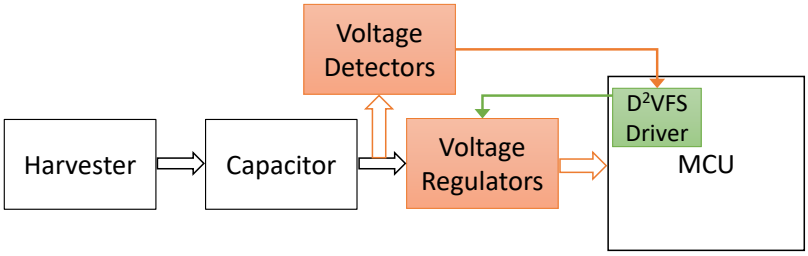


Figure 5-3: D^{2}VFS architecture. *Voltage detectors intercept the capacitor voltage to detect changepoints. The D^{2}VFS driver scales the frequency at these changepoints and reconfigures the voltage regulators to deliver the minimum required voltage for reducing energy consumption.*

5.3 D^{2}VFS

We begin with describing the high level architecture of D^{2}VFS, followed by platform-specific implementation details. To make the discussion concrete, we target MSP430-class MCUs as arguably representative of intermittently-computing platforms for both academic [56, 137] and commercial purposes [12]. Nonetheless, our techniques apply more generally and have a foundational nature.

5.3.1 Architecture

Voltage and frequency scaling in D^{2}VFS is achieved through a hardware-software co-design.

Components. The hardware part is responsible for *detecting* discrete changepoints in capacitor voltage as well as for *regulating* the supply voltage to the MCU. The software part is responsible for *reconfiguring* the MCU clock frequency, the voltage detectors, and the voltage regulators.

The block diagram in Figure 5-3 shows the main hardware and software components that enable D^{2}VFS for embedded MCUs. The capacitor voltage is intercepted by voltage detectors. When these detect a changepoint, an interrupt is fired whose service routine triggers the D^{2}VFS driver. This ascertains the current changepoint by scanning the state of all detectors and decides whether or not to scale the voltage and frequency at the current changepoint. If scaling is required, the D^{2}VFS driver

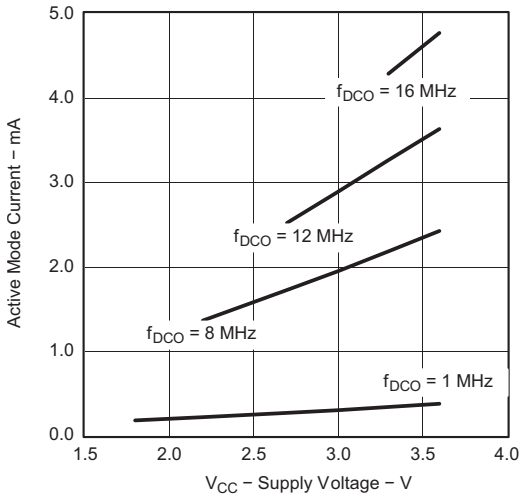


Figure 5-4: Voltage supply range and current consumption of factory calibrated frequencies.

reconfigures both the frequency and the output of the voltage regulators to place the MCU into a different operational zone.

Tradeoffs. The granularity of voltage detection and regulation is key to the efficiency of D^{2}VFS, as both of the corresponding hardware components incur energy overhead.

Components enabling fine-grained control over scaling also consume more energy. The operational voltage range of most MSP430-series MCUs, for example, is between 3.6 V and 1.8 V. In the absence of on-chip DVFS support, detecting and regulating even at every 100 mV drop in this range is impractical, as it would require a large number of accurate voltage detectors and narrow regulators.

Worse still, MSP430 clock registers can be configured to generate 4096 discrete clock frequencies in the range 1-16 MHz, each having its own specific operational voltage range. Since all such frequencies are not factory calibrated, one must empirically calibrate them for each MCU, which is a daunting task, and precisely derive their relationship with an MCU supply voltage to safely scale at run-time.

We therefore restrict frequency scaling in D^{2}VFS to the four factory calibrated frequencies, i.e., 1 MHz, 8 MHz, 12 MHz and 16 MHz. This still gives us enough leverage to scale the MCU clock speed for energy efficiency, while keeping the associated overhead low. For example, we demonstrate that four voltage changepoint detectors and regulation levels (at 3.6 V, 3.3 V, 2.7 V, and 2.2 V) suffice to effectively

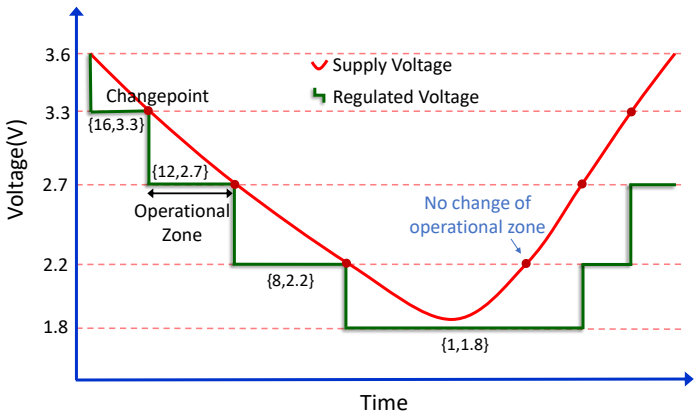


Figure 5-5: Working of D^{2}VFS. *At each changepoint, the D^{2}VFS driver determines whether or not place the MCU in a new operational zone {f (MHz), V}.*

achieve DVFS on intermittently-computing devices.

The factory calibrated configurations for these frequencies are permanently burnt into the on-chip flash, thus, enabling direct reconfiguration of the MCU speed at runtime by merely writing these configurations into the clock registers. Their operational voltage range and power consumption are also well understood, as shown in Figure 5-4, and illustrated in the data sheet [64]. This also spares the developers from tedious calibration efforts.

Operation. Figure 5-5 explains the working of D^{2}VFS. Say the active epoch begins at a capacitor voltage of 3.6 V. The MCU is thus configured to run at 16 MHz and the supply voltage is regulated at 3.3 V, which is the minimum voltage required to operate with this frequency.

In the absence of energy harvested from the ambient, the capacitor voltage continuously drops until 3.3 V, i.e., the first changepoint, where an interrupt fires. The D^{2}VFS driver scans and compares the current state of voltage detectors with the previous one to determine if the current changepoint is reached due to dropping or rising capacitor voltage. In the former case, we place the MCU into a new operational zone by scaling down the MCU speed to 12 MHz and reconfiguring the voltage regulators to deliver 2.7 V to the MCU. A similar scale-down process applies at subsequent changepoints if they are triggered by a dropping voltage due to a capacitor discharge.

If a changepoint is reached from the opposite direction, namely, due to a capac-

Changepoint	**Operational Zone** {f (MHz), V}
3.6	{16, 3.3}
3.3	{12, 2.7}
2.7	{8, 2.2}
2.2	{1, 1.8}

Table 5.1: D^{2}VFS changepoint decisions.

itor charge, we hold back scaling up of the operational zone until the immediately higher changepoint is reached, as shown in Figure 5-5. This effectively implements an hysteresis that prevents dangerous oscillations around a changepoint.

Suppose, for example, a changepoint is reached due to capacitor recharge, as in the case of the changepoint at 2.2 V on the right of Figure 5-5. If the D^{2}VFS driver immediately places the MCU in the new operational zone, that is, it transitions from {1 MHz, 1.8 V} to {8 MHz, 2.2 V}, a sudden discharge of the capacitor is likely to happen due to an increase in power consumption. This may result in the same changepoint to be hit from the opposite direction, triggering a scale down of the MCU to the previous operational zone. In the worst case, this may result in oscillating behaviors.

This technique may result in the MCU operating in different operational zones for the same capacitor voltage range, depending on whether it is being entered due to voltage drop or voltage rise. As shown in Figure 5-5, the MCU operates at {8 MHz, 2.2 V} when the capacitor voltage in the range 2.2 V-2.7 V in the case it is entered via the higher changepoint at 2.7 V, but it operates at {1 MHz, 1.8 V} in the opposite case. Table 5.1 summarizes the scaling decisions.

The D^{2}VFS driver is also in charge of *sequencing* of transitions across changepoints. When scaling up, the voltage regulator is reconfigured first to deliver a higher voltage before increasing the frequency. When scaling down, frequency is reduced first, then voltage is decreased.

D^{2}VFS derives its name—discrete dynamic voltage and frequency scaling—from its voltage regulation behavior that mimics a *discrete* step function, as shown in Fig-

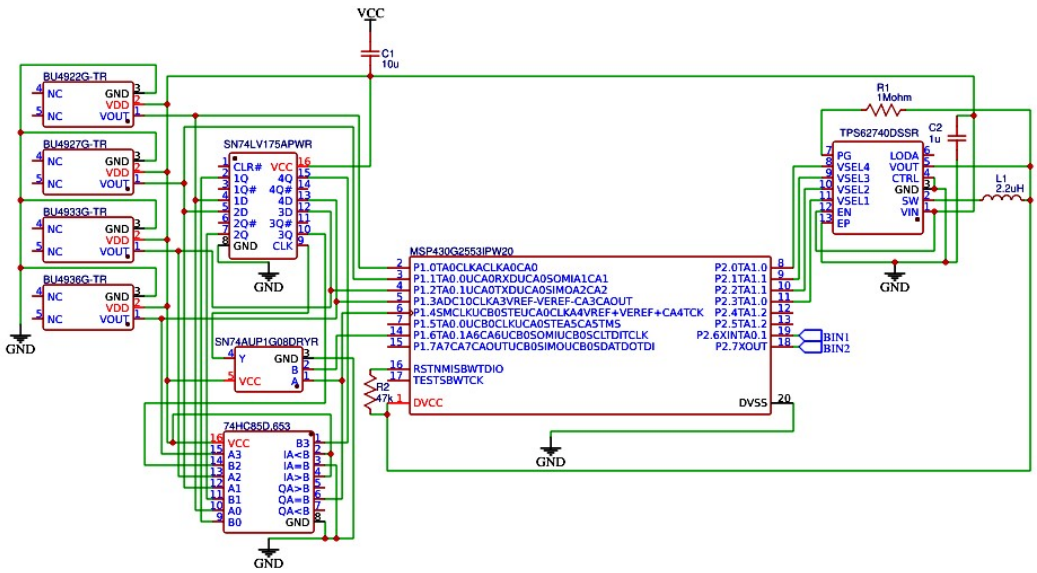


Figure 5-6: D^{2}VFS schematic implementation in EasyEDA.

ure 5-5. A conventional frequency scaling approach may allow the voltage to traverse the entire supply range of a given frequency.

5.3.2 Implementation

We design hardware support for D^{2}VFS in EasyEDA. Figure 5-6 shows the schematics of our implementation. The capacitor voltage is intercepted by BU49xx-series voltage detectors [134]. Every detector generates a 1 if the capacitor voltage is above their detection level, 0 otherwise. The output from the detectors is used as an input to a trigger circuit, which fires an interrupt whenever it detects a change in the input, namely, a changepoint is reached.

The interrupt triggers the execution of D^{2}VFS driver, which scans the state of voltage detectors at GPIOs. This is needed to determine the direction the changepoint is reached and to take corresponding scaling decisions. We change the MCU operational zone by reconfiguring the clock registers and voltage regulators, which are available at GPIOs.

Implementation of D^{2}VFS is enabled by highly efficient voltage detection and regulation ICs. Below we provide important details regarding these amid justifying our choices.

Detector	Detection Voltage			Circuit Current
	Min	Typ.	Max	μA
BU4922	2.19	2.2	2.22	0.26
BU4927	2.67	2.7	2.73	0.29
BU4933	3.27	3.3	3.33	0.34
BU4936	3.56	3.6	3.64	0.35

Table 5.2: Salient features of D^{2}VFS voltage detectors.

Voltage detectors and regulators. We use BU49xx series ROHM semiconductor low-voltage standard CMOS voltage detectors [134]. These are known for high-accuracy ($\pm$1%) and ultra-low current consumption. Their detection range is 0.9-4.8 V at 0.1 V steps, and they can operate in the temperature range -40 °C to 125 °C. Table 5.2 shows the salient features of detector ICs we use in our schematic.

We use TI's TPS62740x step down converters for low power applications [69]. Their input voltage range is 2.2-5.5 V and they can regulate up to 16 output voltages between 1.8-3.3 V with a step size of 100 mV. If the input voltage falls below 2.2 V, the converter enters no ripple (or bypass) mode where it stops regulation and the output is directly connected to the input voltage. Because of that, the MCU obtains unregulated supply voltage for the range 2.2 to 1.8 V, i.e., the lowest operational zone of D^{2}VFS. The component offers 90% efficiency for up to 300 mA output current, which is way beyond the maximum current consumption of MSP430 MCUs, and draws 360 nA quiescent current.

Interrupts. D^{2}VFS needs to generate an interrupt whenever a changepoint is detected. The interrupt logic compares the previous state of detectors with the current one and fires an interrupt when they differ. We use TI's SN54LV175A Quadruple D-TYPE flipflops [113] to store the previous detector state and Nexperia's 74HC85 4-bit magnitude comparators [114]. Both satisfy our supply voltage and ultra-low power requirements. Additionally, we employ a SN74AUP1G08 2-input positive-AND gate [68], with MCU clock and comparator output at its input to clock flipflops. This is to ensure that we do not miss any change in detectors' state if they occur at the

same time as the clock pulse.

Altogether, these components increase the energy consumption of the MCU between 0.56% to 11.5% depending on the capacitor size and operational zone, while offering substantial scaling benefits.

We thoroughly validate our schematic by generating all possible voltage inputs to measure the outputs of both voltage detectors and regulators as well as the power dissipation of the corresponding ICs. We use these measurements to emulate and benchmark the performance of D^{2}VFS.

5.4 Benchmark Evaluation

We evaluate the performance of D^{2}VFS using four key performance metrics to study the various trade-offs involved:

- The *cycles per active epoch* is the number of clock cycles achieved in a single active epoch, whose duration is determined by power consumption, capacitor size, and shutdown voltage threshold. This figure represents a measure of the amount of computation enabled in an active epoch.
- The size of the *smallest energy buffer* is the minimum energy to complete a workload. If too small, a system may be unable to make progress and complete checkpoints, causing non-termination. Using capacitors, however, smaller sizes reach the operating voltage sooner and enable smaller device footprints.
- The *number of checkpoints* is the total number of checkpoints to complete a workload. The more are necessary, the more the system subtracts energy from useful computations. Increasing clock cycles in an active epoch allows the application to progress further on the same charge.
- The *completion time* is the time to complete a workload, excluding deployment-dependent recharge times. D^{2}VFS gradually de-accelerates the MCU and introduces a run-time overhead. On the other hand, due to availability of more compute cycles, a larger portion of workload can be completed within a single active epoch.

5.4.1 Settings

We describe the benchmark applications, the hardware and software we use for experiments, the metrics we compute, and the power profiles we test.

Benchmarks. We consider three benchmarks widely employed in intermittent computing [11, 79, 130, 152]: *i)* a Fast Fourier Transform (FFT) implementation, *ii)* RSA cryptography, and *iii)* Dijkstra spanning tree algorithm. FFT is representative of signal processing functionality in embedded sensing. RSA is an example of security support on modern embedded systems. Dijkstra's spanning tree algorithm is often found in embedded network stacks [76].

These benchmarks offer a variety of memory access patterns and processing load. The FFT implementation has moderate processing requirements; RSA demands great MCU resources; Dijkstra's algorithm only handles integer data and has the lightest processing demands. This diversity allows us to generalize our conclusions. The implementations are from public code repositories [107].

Systems and platforms. We measure D^{2}VFS performance using established system support for intermittent computing, thoroughly discussed in previous chapters; namely Hibernus [11] and MementOS [130].

We consider as baselines the execution of the aforementioned benchmarks along with these support systems on a *statically configured* MSP430G2553 running at 1 MHz, 8 MHz, 12 MHz, and 16 MHz [64]. To make our analysis of MementOS independent of specific energy traces, we manually sweep the possible parameter settings at steps of 0.2 V and use the best performing one [130].

We must note, however, that the choice of underlying system support for evaluating D^{2}VFS is not critical. We choose these systems because of their widespread use for benchmarking intermittently-computing systems [3, 18, 32, 101]. The ability of D^{2}VFS to place the MCU in the most efficient operational zone should, in principle, equally benefit any system support, whether based on checkpointing [18] or abstractions with transactional semantics [32, 101, 104].

5.4.2 Power profile and measurements

We use a foundational power profile found in existing literature [16, 79, 130, 152] that provides fine-grained control over executions and facilitates interpreting results. The device boots with the full capacitor and computes until the capacitor is empty again. In the mean time, the environment provides no additional energy. Once the capacitor is empty, the environment provides new energy until the capacitor is full again and computation resumes.

This profile is also representative of a staple class of intermittently-powered applications, namely, those based on wireless energy transfer [17,23]. With this technology, devices are quickly charged with a burst of wirelessly-transmitted energy until they boot. Next, the application runs until the capacitor is empty again. The device rests dormant until another burst of wireless energy comes in.

We trace the executions on a MSP430G2553 Launchpad interfaced with an FRAM chip for persistent storage. For this purpose, the Launchpad offers a range of hooks, enabling fine-grained measurements. For example, it allows us to practically implement frequency scaling and ascertain the time taken and power consumed by every operation at different frequencies amid accounting for the D^2VFS overhead. We instrument our benchmarks to collect these measurements and simulate energy draw from the capacitor during execution. Due to their extensive usage in electronics, the accuracy of capacitor models for their charge-discharge behavior and voltage drop between the plates is well established and undisputed [62]. The results are obtained from 100 application iterations.

5.4.3 Results → Computation per Active Epoch

We compare the number of clock cycles enabled by D^2VFS, which represents the amount of computation available in an active epoch, with the statically configured factory-calibrated frequency settings. We use variable capacitor sizes, thus different energy storage capacities, and record the clock cycles achieved in a single charge of the capacitor.

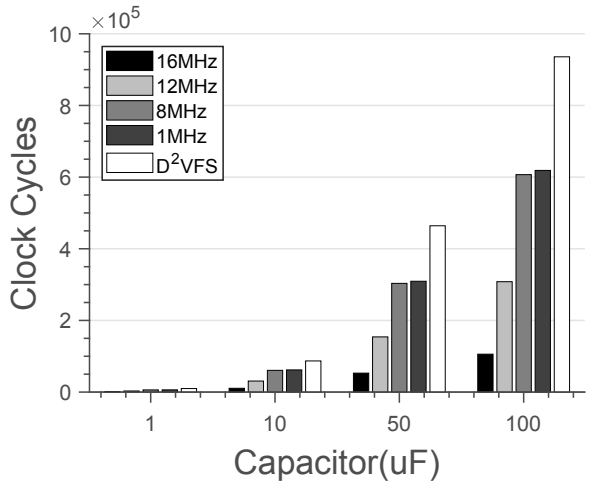


Figure 5-7: Cycles per active epoch. *D^2VFS retains its benefits over static frequency settings despite the implementation overhead due to additional hardware and the D^2VFS driver.*

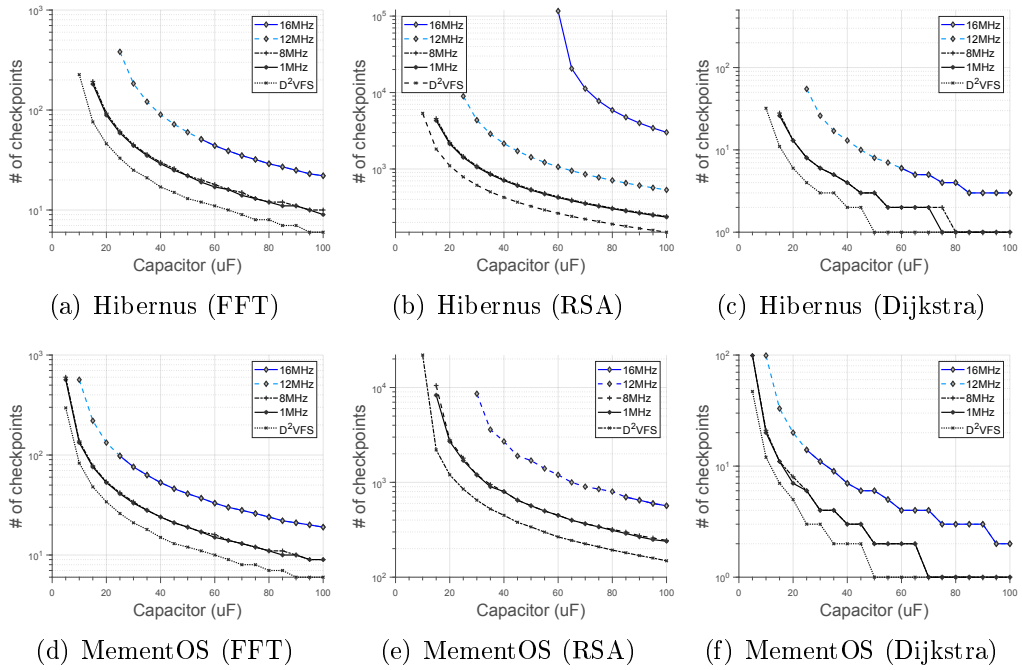
Figure 5-7 shows the results. These are similar to the first order approximation shown in Section 5.2 to scrutinize D^{2}VFS. Nonetheless, we observe that the margin of improvement is only slightly reduced compared to Figure 5-2, validating the energy efficiency of the concrete D^{2}VFS implementation. An appropriate selection of scaling granularity and our choice of hardware components ensures that the energy overhead of D^{2}VFS components is kept low, and that the advantages of dynamic voltage and frequency scaling are retained.

Worth noticing is also that the margin of improvement increases with the size of capacitor. This is due to constant power dissipation of D^{2}VFS components and static number of changepoints in an active epoch irrespective of capacitor sizes. Thus, the percentage energy overhead of D^{2}VFS decreases when the capacitor size increases.

5.4.4 Results → Checkpoints and Energy Buffers

The results in the number of checkpoints and in the size of the smallest energy buffer are intimately intertwined.

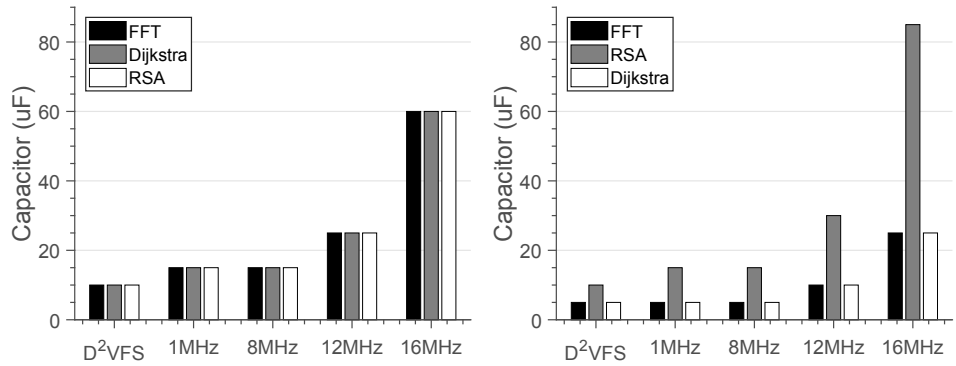
Figure 5-8 depicts the reduction in the number of checkpoints against variable capacitor sizes. A portion of these charts only shows the performance of the D^{2}VFS-based execution, as the ones with static frequency settings are unable to complete the



(a) Hibernus (FFT) (b) Hibernus (RSA) (c) Hibernus (Dijkstra)

(d) MementOS (FFT) (e) MementOS (RSA) (f) MementOS (Dijkstra)

Figure 5-8: Number of checkpoints necessary against varying capacitor sizes. *D^2VFS allows the system to progress farther on a single charge, reducing the number of checkpoints needed to complete a workload.*



(a) Hibernus (b) MementOS

Figure 5-9: Smallest capacitor. *A D^2VFS-equipped system completes the workload with smaller capacitors. This is due an increase in available clock cycles in a single active epoch.*

workload with too small capacitors. When a comparison is possible, the improvements for D^{2}VFS are substantial and apply across benchmarks and capacitor sizes.

The number of checkpoints is maximum at 16 MHz. There, the MCU only operates in a narrow interval of supply voltage, resulting in fewer clock cycles in an active epoch compared to other static frequency settings. Therefore, the execution occurs in small

bursts separated by checkpoints.

Figure 5-9 reports the minimum size of the capacitor required to complete the given workloads. A D^{2}VFS-equipped system constantly succeeds with smaller capacitors. With Hibernus, D^{2}VFS allows one to use a capacitor up to one-sixth of the one required with a static frequency setting. Since checkpoints in MementOS are more energy efficient, it generally achieves smaller capacitor sizes compared to Hibernus. However, the heuristics employed by MementOS[2] for inserting checkpoint calls are not always productive and may lead to an increase in peak energy demand despite energy efficient checkpoint operations. For example, in one particular instance of RSA, the calls to checkpoint fell so far apart that the energy needed to guarantee successful execution exceeds that of Hibernus, requiring an even bigger capacitor size, as shown in Figure 5-9(b).

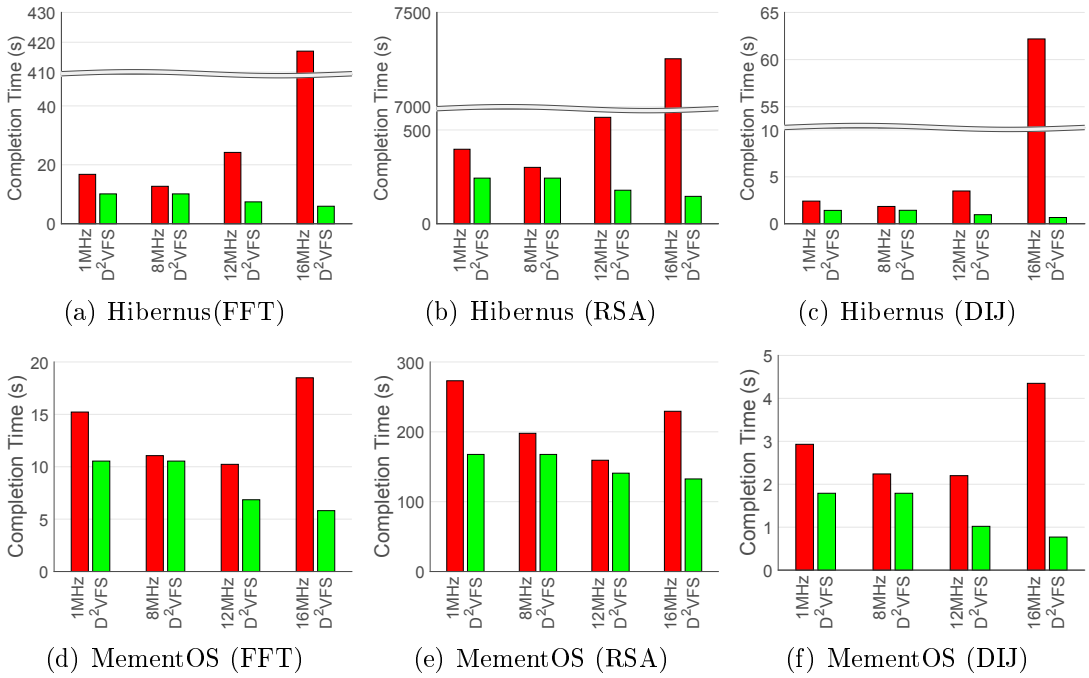
These results are directly enabled by the ability of D^{2}VFS to increase the number of clock cycles, thus allowing systems to perform more computation in a single active epoch. Applications thus progress farther on a single charge, while reducing the number of checkpoints en route to completion. Similarly, the smallest amount of energy the system needs to have available at once to move from one checkpoint to the next without any power failure in between, reduces as well. Smaller capacitors mean reaching operating voltage faster and smaller device footprints.

These results demonstrate that D^{2}VFS allows intermittently-computing systems to reduce the total *time* invested in checkpoint operations because of a reduction in their number. This leads to shorter completion times for given workloads, as we investigate next.

5.4.5 Results → Completion Time

D^{2}VFS gradually de-accelerates the processor to exploit the maximum range of the MCU supply voltage. On the one hand, this increases the instruction execution time compared to higher frequencies (e.g., 16 MHz). On the other hand, checkpointing

[2]MementOS suggests two strategies for inserting checkpoint calls: (1) after every loop, or (2) after return from every function call [130].



(a) Hibernus(FFT) (b) Hibernus (RSA) (c) Hibernus (DIJ)

(d) MementOS (FFT) (e) MementOS (RSA) (f) MementOS (DIJ)

Figure 5-10: Completion time. *The run-time overhead due to D²VFS is overturn by increasing the number of clock cycles available, and therefore the amount of computation possible in a single active epoch, and thus reducing the number of checkpoints. The combined effect shortens completion times.*

operations are delayed and more useful progress is made before a checkpoint is eventually required. We explore the relative contribution of either aspect to the total completion time for a given workload.

Figure 5-10 reports the results. We execute these experiments using the smallest common capacitor needed to guarantee completion with a given static frequency setting and D²VFS, as these are the preferred settings for an intermittently-computing device. We can see that the reduction in speed is not only compensated, but actually overturned by the ability of D²VFS to complete larger portions of the workload in each setting before initiating checkpoints. This allows D²VFS to complete the workload much earlier, increasing the system's responsiveness.

Comparing across static frequency settings is not possible in these experiments because the respective smallest capacitor sizes ensuring completion of their workload are different, as shown in Figure 5-9. With the change in capacitor size, the factors that impact completion time such as the amount of energy storage, the number of available clock cycles in a single active epoch and, more importantly, the location and

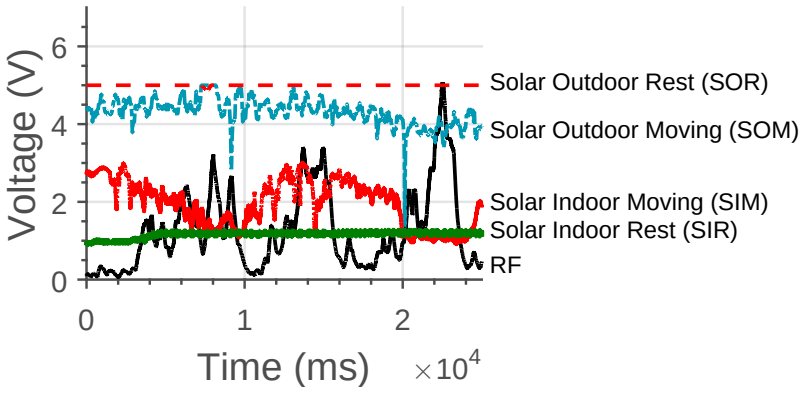


Figure 5-11: Voltage traces used for evaluation.

the number of checkpoints change drastically. Similarly, the reduced energy overhead in MementOS due to smaller size of checkpoints causes it to produce a different outcome using static frequency settings compared to Hibernus.

Regardless, D^{2}VFS consistently and substantially outperforms all static frequency settings across all benchmarks, and its benefits are not affected by these factors.

5.5 Real World Evaluation

The benchmark evaluation of D^{2}VFS uses a synthetic power profile that does not take into account the recharge times of the capacitor. In real world settings, the recharge times may differ depending upon the frequency settings. A capacitor retains large amounts of residual energy at higher frequencies, since these only operate in a narrow interval of supply voltage, but drains off considerably at lower frequencies. Conversely, the charging rate of capacitor is faster at the start but then tapers off as the capacitor acquires additional charge at a slower rate.

We thus investigate the impact of these conflicting factors using real-world power traces and confirm whether the results of Section 5.4 carry over to real world settings. We build the same activity recognition (AR) application often seen in the literature [21, 32, 101, 104, 130], using the same source code [49]. The rest of the setup is as for Hibernus in Section 5.4.

We focus on completion time using the smallest capacitor that ensures completion

of the AR application across D^{2}VFS and the baselines we consider. Based on the results of Section 5.4, completion time is indeed the one metric where all the involved trade-offs manifest.

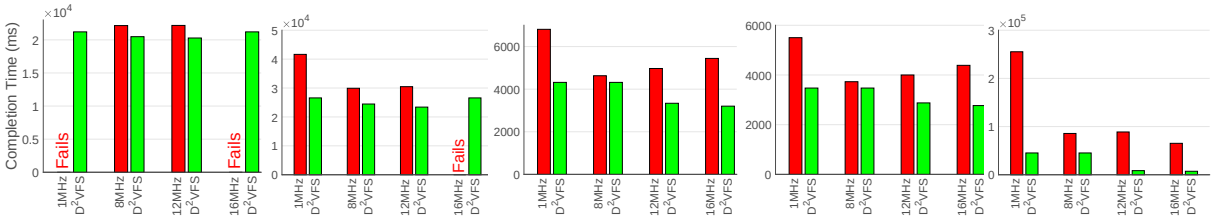
Power traces. We consider five power traces, obtained from diverse energy sources and in different settings. One of the traces is the RF trace from MementOS [88, 130]. The black curve in Figure 5-11 shows an excerpt, plotting the instantaneous voltage reading at the energy harvester over time.

We collect four additional traces using a mono-crystalline solar panel [147] and an Arduino Nano [9] to measure the voltage output across a 30 kΩ load, roughly equivalent to the resistance of the MSP430G2553 in active mode [64]. Using this setup, we experiment with different scenarios. We attach the device to the wrist of a student to simulate a fitness tracker. The student roams in the university campus for outdoor measurements (SOM), and in research lab for indoor measurements (SIM). Alternatively, we keep the device on the ground right outside the lab for outdoor measurements (SOR), and at desk level in our research lab for the indoor measurements (SIR).

Figure 5-11 demonstrates the extreme variability and considerable differences among the power traces we consider.

Results. Figure 5-12 shows the completion times across all traces. D^{2}VFS consistently performs better, regardless of the power trace and static frequency setting it is compared with. Another notable observation is the failure of MCU to complete the workload when running at 1 MHz with the RF trace and at 16 MHz with both the RF and SIR traces. Both the *lowest* and *highest* frequencies get penalized. The former happens because of the highest energy per clock cycle ratio as well as by a considerably drained off capacitor at the end of each active epoch; whereas the latter is to endure the largest number of costly checkpoints.

The intermediate frequencies, that are, 8 MHz and 12 MHz, perform better comparatively, but D^{2}VFS reaps the highest benefit from all possible frequency settings; it makes the system operate in the most efficient configuration given a certain capac-



(a) RF (b) SIR (c) SOM (d) SOR (e) SIM

Figure 5-12: Performance of the AR app running on Hibernus. *Performance gains with D^2VFS are observed across diverse power traces obtained from different energy sources. The average improvement compared to static frequency settings ranges from 30% to 300% across all traces.*

itor voltage, while also reducing the checkpointing overhead. Based on these results, the performance and trade-offs we discuss in Section 5.4 are thus confirmed with a concrete application and diverse power traces.

5.6 Related Work

A large body of work investigates DVFS in mainstream computing. However, to scale down our discussion on related works, here we focus on DVFS in embedded systems. We broadly divide related literature in three categories: general purpose embedded systems, wireless sensor networks, and intermittently-computing devices.

Embedded systems. Salehi et al. [136] present an adaptive voltage and frequency scaling technique that rapidly tracks the workload changes to meet soft real-time deadlines. Their work demonstrates considerable power savings and fewer frequency updates compared to DVFS systems based on fixed update intervals. HyPowMan [15] considers the problem of power consumption minimization for periodic real-time tasks that are scheduled over multiprocessor platforms with dynamic power management (DPM) and DVFS capabilities. This technique takes a set of well-known existing DPM and DVFS policies, each performing well for a given set of conditions, and adapts at run-time to the best-performing policy for any given workload.

Huang et al. [63] apply DVFS to mixed-criticality systems, and show that DVFS can be used to help critical tasks meet deadlines by speeding up the processor when

they are bound to miss the deadline. Liu et al. [98] employ DVFS to optimize system thermal profile, to prevent run-time thermal emergencies, and to minimize cooling costs. They present a framework for system designers to determine a proper thermal solution and provide a lower bound on the minimum temperature achievable by their DVFS technique. RT-DVFS [122] targets embedded operating systems, such as in mobile phones and camcorders. It modifies the OS's real-time scheduler and task management service to provide significant energy savings while maintaining real-time deadline guarantees. Generalized Shared Recovery (GSHR) [157] efficiently uses DVFS techniques to achieve a given reliability goal for real-time embedded applications.

These works provide foundational knowledge on applying DVFS in embedded systems, yet their design goals are very different, and their techniques are not directly applicable to intermittently-computing devices.

Wireless sensor networks. Kulau et al. [85–87] thoroughly analyze the effects of undervolting for a typical wireless sensor node both in theory and practice. They show that a wireless sensor network can still work reliably, even if the voltage recommendations are violated, because there is a correlation between temperature and error-proneness at the same voltage level, and that ideal voltage levels depend on environmental conditions. Powell et al. [125] design DVFS hardware to meet battery life and form factor expectations of body area sensor networks. Similar to this are the works on developing DVFS techniques in distributed microsensor networks [112] and in sensing applications with deadlines [7].

Many of these works are similar to ours in spirit, as they all aim to conserve energy, yet these approaches consider battery-powered devices with *finite* energy supplies, and tend to accept performance penalties to increase life time. On the contrary, we deal with intermittent but unbounded energy supplies where the goal is to increase the amount of work done in an active epoch that, in turn, improves a number of other key performance metrics. The techniques we use are rather aggressive compared to the ones employed in wireless sensor networks in scaling both voltage and frequency.

Intermittently-computing devices. EA-DVFS [97] is a high-level simulation-

based study that highlights the benefits of DVFS in achieving real-time operation on battery-less devices. As the corresponding hardware architecture and implementation is not available, this approach cannot be used as baseline in our work. Noise-aware DVFS sequence optimization techniques are proposed to reduce noise, i.e., extra current that accompanies the clock speed transition, in energy-harvesting devices [103]. This work is complementary to our efforts and, if integrated with them, we expect a further reduction in the energy overhead.

Lin et al. [96] model a framework for concurrent task scheduling and dynamic voltage and frequency scaling in real-time embedded systems with energy harvesting. They develop a global controller that performs optimal operating point tracking for the PV panel, state-of-charge management for the supercapacitor, and energy-harvesting aware real-time task scheduling with DVFS. Li et al. [93] also provide early insights into the benefits of jointly scaling workload, voltage, and frequency in multi-core sensor networks powered by energy harvesting.

These works provide useful early-stage insights on employing DVFS in energy harvesting devices. However, D^2VFS is the first concrete implementation of any such technique, along with a detailed evaluation that precisely highlights the benefits of employing DVFS in intermittently-computing devices.

5.7 Summary

The peculiar provisioning patterns of ambient energy harvesting, together with the features of modern low-power MCUs, create an opportunity to improve the energy efficiency of intermittently-computing devices by dynamically adjusting supply voltage and frequency settings.

We seized to the opportunity and presented D^2VFS, a hardware and software co-design that seeks to reap maximum benefits from these patterns and features by reducing the overhead imposed by additional hardware components and software drivers. To make up for the lack of dedicated hardware on low-power MCUs, we employ an external low-power DVFS engine to identify voltage changepoints and trigger

the execution of a software driver that scales both supply voltage and frequency settings. The performance improvements of D^2VFS are notable. For example, the number of available clock cycles in a single active epoch is increased by 40-900% compared with static frequency settings. The ultimate impact is on workload completion times, which are reduced by up to 300%, as shown in real world settings with diverse power traces.

Chapter 6

Discussion and Conclusion

In this dissertation, we propose techniques for reducing size of checkpoint, its triggering mechanism, and increasing the computational progress made by transiently powered devices in a given energy budget. The basic idea is to maximize the amount of energy available for program execution, while using it efficiently, to reduce the completion time of an application and energy spent on checkpointing. This requires relevant system support at three different levels: First, we developed a **run-time** technique to track changes in the application state. We keep an in-memory data structure allowing us to not only reduce NVM accesses but also enabled faster computation of checkpoint differentials. Second, we propose a **compile-time** tool to predict energy consumption of intermittent programs; allowing better placement of trigger calls in the program. This required energy profiling of the MCUs and incorporating variation in energy consumption with variation in voltage. Our tool provides accurate prediction of the energy consumption which helps TPC in avoiding redundant checkpoints. Finally, we present a **hardware** design to configure transiently powered devices at the most energy-efficient configuration, requiring least energy per-clock-cycle while executing program instruction at a faster rate.

All the three approaches can be integrated in a system stack to reap benefits provided by each approach. For example, our evaluation for DICE, on both copy-all and copy-used approaches show that the proposed techniques are independent of the underlying hardware technology. One key achievement is that our prototype

implementations outperform the state-of-the-art in checkpointing strategies, triggering mechanisms, and hardware support for faster execution of programs. Besides thoroughly evaluating the proposed approaches on widely used applications in intermittent computing domain, the data analysis presented in this dissertation argues about the applicability of the legacy volatile systems in transiently powered systems.

The rest of this chapter is structured as follows. Section 6.1 summarizes the major contributions of this dissertation by revisiting the key concepts of checkpointing strategies, triggering mechanisms, and hardware support for faster program execution. In Section 6.2, we shed light on the limitations and Section 6.3 highlights the future directions in the intermittent-computing domain.

6.1 Summary

In this section, we summarize our main contributions by focusing on the key concepts discussed in the preceding chapters.

6.1.1 Differential Checkpointing

Existing approaches for checkpointing application state for transiently powered devices either use a copy-all approach to copy the entire main memory onto NVM or they copy the used memory regions to retain application state across reboots. With the aim of reducing run-time overhead, these approaches use a coarse grained analysis to identify the change in application, resulting in high NVM accesses. Some techniques which are able to find checkpoint differentials incur a high computational overhead due to NVM reads thus losing the benefit gained by checkpoint size reduction. Overall, these approaches waste energy in performing checkpointing, leaving very little for application execution causing high latency and reduced throughput of the application.

Our observation show that there are very few memory locations which get modified during application execution. Further analysis revealed that only few program statements are responsible for those changes in the memory. However, tracking those

Techniques Compared	Checkpoint size	NVM accesses	Differential Checkpoints
Copy-all	very high	very high	✗
Copy-used	high	high	✗
DICE	low	low	✓

Table 6.1: DICE Summary: DICE uses a differential checkpointing technique which helps it in reducing checkpoint size. It uses an in-memory data to record checkpoint differentials which reduces NVM accesses.

locations through naive method can incur a significant computational overhead thus nullifying the benefit. Additionally, employing any new hardware for tracking such statements adds to the energy consumption of the device.

We propose a software-only approach to track changes in the main memory at two levels; global and local level. At global level, DICE uses a pre-compiler to instrument the source code to add special function calls, **`record()`**, in the program before each state modifying statement. We keep an in-memory data structure, modification record, which saves the location and size of the change in main memory. We represent the entire main memory with a bit array, in the modification record, with a goal to reduce the burden on small memory sizes of these resource constrained devices. At the local level, we introduce a stack tracking technique that helps us avoid the computational overhead of tracking the short lived local variables.

Based on these techniques, we enabled checkpoint size reduction for both proactive and reactive techniques enabling them to set lower threshold for checkpointing allowing them to perform more computations in the given energy budget. Our evaluation demonstrates that checkpoint size reported by our techniques is, by orders of magnitude, less than what was reported by existing techniques. This implies reduction in the checkpointing energy thus allowing the system to make more progress on the charge. Our approach unlocks devices which can execute long running workloads on even smaller capacitor sizes; enabling smaller device footprints.

Our major findings with regard to DICE's performance are summarized in Table 6.1.

6.1.2 Modeling Dynamic Energy Consumption

Triggering mechanisms, used in existing literature, either rely on the user/programmer or employ constant energy models for accurate placement of trigger calls. These models pessimistically assume each program instruction to be executed at the constant maximum voltage. Although, this ensures correct program execution under intermittent energy supply, a significant amount of energy in the energy budget gets wasted which can be used for program execution.

We propose a compile-time tool to predict energy consumption of intermittent program which can automate the process of energy prediction. It models the energy consumption of underlying hardware after carefully considering the variation of clock frequency and power consumption with the variation in input voltage. A primary goal of EPIC is to provide accurate energy consumption estimates of the intermittent program. For this purpose, we evaluate our model in three different use-cases. Our results show that EPIC outperforms state-of-the-art solutions in the literature. For example, our energy model allows HarvOS to make more precise placement of trigger calls which helps the device avoid redundant checkpoints. When integrated with CleanCut, our model removes false positive reported by the tool previously.

While we show the effectiveness of our model on MCUs of MSP430G2 family, the proposed approach for modeling energy consumption is more general one and is not tied to any specific MCU. Given an energy profile of any MCU, our tool can predict energy consumption of the program accordingly.

Table 6.2 summarizes EPIC's evaluation results.

Techniques Compared	**Improvement**	
	# of checkpoints	**Speedup**
HarvOS	2x	3.5x
MSPsim	4.5x	7x

Table 6.2: EPIC's summary: Precise modeling of clock and current behavior enables EPIC to significantly reduce the number of checkpoint required for completing intermittent programs. This reduction translates into gain in speedup by orders of magnitude.

6.1.3 D^{2}VFS

Non-volatile hardware support for removing checkpoints force the TPC to run the MCU on fixed slower frequencies in order to ensure longer operation on the same charge. However, such configuration lead to inefficient energy consumption due to high energy and access latency requirements of the non-volatile hardware. We propose a dynamic voltage and frequency scaling technique on top of a volatile system to argue that, it is possible have an energy-efficient execution of program on a volatile system by performing run-time adaptation to the changing energy conditions.

Dynamic voltage and frequency scaling technique has been widely used for energy conservation in the main-stream computing. It allows the MCU to dynamically react to the changing energy requirements of the MCU i.e., scale up when the application demands high performance or scale down when the application is requires energy conservation. However, the primary focus of existing literature, in the intermittent computing domain, is to reduce the energy spent on checkpointing and its tracking. With highly variable and unpredictable energy supply, TPC need to react dynamically to the ever changing energy conditions. Unfortunately, none of the existing state-of-the-art solutions enable them in achieving this goal.

To overcome this limitation, we propose a hardware-software co-design to perform discrete dynamic voltage and frequency scaling technique, D^{2}VFS, that performs voltage and frequency scaling at discrete steps thus configuring the system at the best configuration in a given voltage range. It defines the set of best configurations applicable in different operational zones in the voltage range. Each operational zone

Techniques Compared	**Improvement**		
	Smallest Capacitor	**Active Epoch Cycles**	**Completion Time**
16MHz	6x	9x	3x
1MHz	50%	40%	30%

Table 6.3: D^{2}VFS summary: Our approach dynamically adapts to the changing energy condition and enables TPC to execute more number of clock cycles in the given energy budget; resulting reduced completion time for applications.

start at a particular voltage point, the changepoint. D^{2}VFS detects them with the help of voltage detectors. At each changepoint, the system either moves to higher or a lower operational zone as per the incoming energy condition. D^{2}VFS configures the MCU to best frequency while regulating the voltage at the lower end of the operational range of the configured frequency. This allows minimum energy consumption per clock cycle at any given configuration. The basic idea is to run the system as fast as possible at a particular voltage to enable maximum computational progress.

Our prototype implementation and evaluation shows superior performance of our approach over common configuration techniques used in the intermittent computing literature. Table 6.3 summarizes the comparative evaluation of D^{2}VFSwith static configuration.

6.2 Limitations

In this section, we summarize our key findings and pinpoint the limitations of the approaches presented in this dissertation.

6.2.1 Differential Checkpointing

Differential checkpointing techniques are designed to be deployed on devices facing high intermittence in the application execution; a common case for transiently-powered systems. However, the computational overhead of these approaches overcomes the benefit brought by checkpoint size reduction when the energy in the environment is abundant. The energy buffer gets charged continuously as the program executes; allowing the device to make more progress on the single charge thus increasing the distance between two checkpoints. Consequently, changes in the application state get accumulated resulting in higher checkpoint size and computational overhead.

6.2.2 Modeling Dynamic Energy Consumption

The empirical evidence we provide, for EPIC's viability, is focused on MSP430-class MCUs. Despite their popularity especially in TPCs, they still represent a specific instance in a potentially vast landscape.

We maintain, however, that our work does have a foundational nature. The contribution we provide is ultimately more general than embodied in concrete systems. The measurement methodology we employed to derive the empirical model is applicable to other MCUs. Once an energy model is derived for other MCUs, the design of EPIC and MSPsim++ remains the same.

6.2.3 Fast Program Execution

Our design of D^2VFS focuses only on running the configuring the processor to best operational zone at any given voltage thus allowing energy-efficient program-execution on the processor. However, peripheral devices have different configurations than that of the processor. Therefore, if a program needs to communicate with the peripheral device, the processor must switch back to the minimum voltage level required by the peripheral device to operate and should not switch back until the peripheral operation completes.

6.3 Future Work

This dissertation focuses on exploring the potential of making program execution faster and energy-efficient in the transiently powered computers. In the following we briefly highlight the possible future directions for this work.

6.3.1 Approximate Computing

With the ever increasing number of edge devices, the amount of data gathered by these devices has increased tremendously thus demanding on-device processing to save energy and network bandwidth. These devices employ complex machine learning

algorithms to perform analysis on the data. With their resource-constrained nature, machine learning algorithms have to be executed on the edge in an energy-efficient manner to successfully complete the analysis in the given energy budget of these devices. However, machine learning algorithms perform large number of computations which, either are repetitive, or have little-to-no-effect on the end result. Performing such computations not only wastes energy but it also increases the response time for an application. Therefore, there is a need to approximate the output of such computations to avoid spending energy and save time; demanding language-,compiler and run-time support for these devices.

6.3.2 Secure and Reliable checkpointing

Federated learning algorithms are trending in the edge computing domain where a global machine learning model is learned from the information gathered by an edge device. This learned model is then disseminated to all edge devices connected to the cloud. While this approach allows an edge device to learn from the experiences of other devices in the network, it is prone to model poisoning attacks as a malicious device can degrade the performance of model e.g., causing mispredictions for a particular class. Therefore, techniques are required to detect such malicious users at the edge and cloud level.

Bibliography

[1] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, 2004.

[2] Saad Ahmed, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. The betrayal of constant power x time: Finding the missing joules of transiently-powered computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019.

[3] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019.

[4] Muhammad Hamad Alizai, Qasim Raza, Yasra Chandio, Affan A. Syed, and Tariq M. Jadoon. Simulating intermittently powered embedded networks. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, 2016.

[5] James Allen, Matthew Forshaw, and Nigel Thomas. Towards an extensible and scalable energy harvesting wireless sensor network simulation framework. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017.

[6] Patricia Anacleto, PM Mendes, E Gultepe, and DH Gracias. 3d small antenna for energy harvesting applications on implantable micro-devices. In *Antennas and Propagation Conference (LAPC)*, 2012.

[7] Ryan Antonio, Rafael Costa, Aldrin Ison, Wesley Lim, Robert Pajado, Deanne Roque, Ruelle Yutuc, Chris Densing, Maria Theresa de Leon, Marc Rosales, and Louis Alarcon. Implementation of dynamic voltage frequency scaling on a processor for wireless sensing applications. In *TENCON*, 2017.

[8] Faycal Ait Aouda, Kevin Marquet, and Guillaume Salagnac. Incremental checkpointing of program state to nvram for transiently-powered systems. In *Recon-*

figurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on, 2014.

[9] ARDUINO. Nano. `https://store.arduino.cc/usa/arduino-nano`, 2018. Accessed: 11-04-2020.

[10] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[11] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 2015.

[12] Medusa-M2233 UHF RFID battery-free device. Farsens. `http://www.farsens.com/en/products/medusa-m2233/`, 2019. Accessed: 11-04-2020.

[13] David Benedetti, Chiara Petrioli, and Dora Spenza. Greencastalia: An energy-harvesting-enabled framework for the castalia simulator. In *Proceedings of the 1st International Workshop on Energy Neutral Sensing Systems*, 2013.

[14] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Sytare: a lightweight kernel for nvram-based transiently-powered systems. *IEEE Transactions on Computers*, 2018.

[15] Muhammad Khurram Bhatti, Cécile Belleudy, and Michel Auguin. Power management in real time embedded systems through online and adaptive interplay of dpm and dvfs policies. *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, 2010.

[16] Naveed Bhatti and Luca Mottola. Efficient state retention for transiently-powered embedded sensing. In *International Conference on Embedded Wireless Systems and Networks*, 2016.

[17] Naveed Anwar Bhatti, Muhammad Hamad Alizai, Affan A Syed, and Luca Mottola. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions on Sensor Networks (TOSN)*, 2016.

[18] Naveed Anwar Bhatti and Luca Mottola. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2017.

[19] Naveed Anwar Bhatti, Affan Ahmed Syed, and Muhammad Hamad Alizai. Sensors with Lasers: Building a WSN Power Grid. In *Proc. 13th Int. Symp. Information Processing in Sensor Networks*, 2014.

[20] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[21] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019.

[22] Florian Brandner, Benoit Boissinot, Alain Darte, Benoît Dupont De Dinechin, and Fabrice Rastello. Computing Liveness Sets for SSA-Form Programs. Technical report, INRIA, 2011.

[23] Michael Buettner, Benjamin Greenstein, and David Wetherall. Dewdrop: An energy-aware runtime for computational RFID. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, 2011.

[24] Binlei Cai and Keqiu Li. Slo-aware colocation: Harvesting transient resources from latency-critical services. *Journal of Systems Architecture*, 2019.

[25] Giuseppe Mario Calvagna, Giuseppe Torrisi, Clea Giuffrida, and Salvatore Patanè. Pacemaker, implantable cardioverter defibrillator, crt, crt-d, psychological difficulties and quality of life. *International journal of cardiology*, 2014.

[26] Luís Vitório Cargnini, Lionel Torres, Raphael Martins Brum, Sophiane Senni, and Gilles Sassatelli. Embedded memory hierarchy exploration based on magnetic ram. In *2013 IEEE Faible Tension Faible Consommation*, 2013.

[27] Andrea Castagnetti, Alain Pegatoquet, Cécile Belleudy, and Michel Auguin. A framework for modeling and simulating energy harvesting WSN nodes with efficient power management policies. *EURASIP J. Emb. Sys.*, 2012.

[28] Geoffrey Werner Challen, Jason Waterman, and Matt Welsh. Idea: Integrated distributed energy awareness for wireless sensor networks. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010.

[29] Hsung-Pin Chang, Yen-Ting Liu, and Shang-Sheng Yang. Surviving sensor node failures by mmu-less incremental checkpointing. *Journal of Systems and Software*, 2014.

[30] Ming-Tsung Chiu and Yi-Ping You. Clpkm: A checkpoint-based preemptive multitasking framework for opencl kernels. *Journal of Systems Architecture*, 2019.

[31] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. *SIGOPS Oper. Syst. Rev.*, 2016.

[32] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[33] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, 2018.

[34] Nikolaus Correll, Prabal Dutta, Richard Han, and Kristofer S. J. Pister. New directions: Wireless robotic materials. *ArXiv*, 2017.

[35] CSIRO. Global initiative for honey bee health. `t.ly/wZRbv`, 2019. Accessed: 11-04-2020.

[36] Riccardo Dall'Ora, Usman Raza, Davide Brunelli, and Gian Pietro Picco. Senseh: From simulation to deployment of energy harvesting wireless sensor networks. In *IEEE 39th Conference on Local Computer Networks, Edmonton, AB, Canada, 8-11 September, 2014 - Workshop Proceedings*, 2014.

[37] Denis C Daly, Patrick P Mercier, Manish Bhardwaj, Alice L Stone, Zane N Aldworth, Thomas L Daniel, Joel Voldman, John G Hildebrand, and Anantha P Chandrakasan. A pulsed uwb receiver soc for insect motion control. *IEEE Journal of solid-state circuits*, 2010.

[38] Amine Didioui, Carolynn Bernier, Dominique Morche, and Olivier Sentieys. Harvwsnet: A co-simulation framework for energy harvesting wireless sensor networks. In *International Conference on Computing, Networking and Communications, ICNC 2013, San Diego, CA, USA, January 28-31, 2013*, 2013.

[39] Bernhard Egger, Younghyun Cho, Changyeon Jo, Eunbyung Park, and Jaejin Lee. Efficient checkpointing of live virtual machines. *IEEE Trans. Computers*, 2016.

[40] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Osterlind, and Thiemo 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.

[41] H.A Nguyen et al. Sensor node lifetime: An experimental study. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, 2011.

[42] J. Yang et al. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007.

[43] Kurt B Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. libhashckpt: hash-based incremental checkpointing using gpu's. In *European MPI Users' Group Meeting*, 2011.

[44] M. Muztaba Fuad and Michael J. Oudshoorn. Transformation of existing programs into autonomic and self-healing entities. In *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), 26-29 March 2007, Tucson, Arizona, USA*, 2007.

[45] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. Realistic simulation for tiny batteryless sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, 2016.

[46] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[47] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: A portable testbed for the batteryless iot. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019.

[48] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[49] Abstract Research Group. Benchmark applications. `www.github.com/CMUAbstract/releases\#benchmark-applications`, 2018. Accessed: 11-04-2020.

[50] Berkin Güler and Öznur Özkasap. Efficient checkpointing mechanisms for primary-backup replication on the cloud. *Concurrency and Computation: Practice and Experience*, 2018.

[51] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*, 2001.

[52] Joaquín Gutiérrez, Juan Francisco Villa-Medina, Alejandra Nieto-Garibay, and Miguel Ángel Porta-Gándara. Automated irrigation system using a wireless sensor network and gprs module. *IEEE transactions on instrumentation and measurement*, 2014.

[53] Mohamadhadi Habibzadeh, Moeen Hassanalieragh, Akihiro Ishikawa, Tolga Soyata, and Gaurav Sharma. Hybrid solar-wind energy harvesting for embedded applications: Supercapacitor-based system architectures and design tradeoffs. *IEEE Circuits and Systems Magazine*, 2017.

[54] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014.

[55] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, 2015.

[56] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *SenSys*, 2017.

[57] Josiah Hester and Jacob Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017.

[58] Josiah Hester, Kevin Storer, Lanny Sitanayah, and Jacob Sorber. Towards a language and runtime for intermi ently-powered devices. *sleep*, 2016.

[59] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017.

[60] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P Burleson, and Jacob Sorber. Persistent clocks for batteryless sensing devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 2016.

[61] Matthew Hicks. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[62] Paul Horowitz and Winfield Hill. *The art of electronics*. Cambridge Univ. Press, 1989.

[63] Pengcheng Huang, Pratyush Kumar, Georgia Giannopoulou, and Lothar Thiele. Energy efficient dvfs scheduling for mixed-criticality systems. In *Proceedings of the 14th International Conference on Embedded Software*, 2014.

[64] Texas Instruments. Data sheet. `http://www.ti.com/lit/ds/symlink/msp430g2553.pdf`, 2013. Accessed: 11-04-2020.

[65] Texas Instruments. Msp430 solar energy harvesting development tool. `http://www.ti.com/tool/EZ430-RF2500-SEH`, 2013. Accessed: 11-04-2020.

[66] Texas Instruments. Msp430x2xx family user guide. http://www.ti.com/lit/pdf/slau144, 2013. Accessed: 11-04-2020.

[67] Texas Instruments. Msp430fr573x mixed-signal microcontrollers. http://www.ti.com/lit/ds/symlink/msp430fr5739.pdf, 2014. Accessed: 11-04-2020.

[68] Texas Instruments. Data sheet. http://www.ti.com/lit/ds/symlink/sn74aup1g08.pdf, 2016. Accessed: 11-04-2020.

[69] Texas Instruments. Voltage regulator. http://www.ti.com/lit/ds/slvsb02b/slvsb02b.pdf, 2016. Accessed: 11-04-2020.

[70] Texas Instruments. Getting Started with the MSP430 LaunchPad. https://goo.gl/6ueTEC, 2018. Accessed: 11-04-2020.

[71] Texas Instruments. Manual. http://www.ti.com/lit/an/slaa336a/slaa336a.pdf, 2018. Accessed: 11-04-2020.

[72] Texas Instruments. Power-management integrated chip (pmic). https://goo.gl/45psWK, 2018. Accessed: 11-04-2020.

[73] Texas Instruments. Ti e2e coummunity. https://goo.gl/XxrhN3, 2018. Accessed: 11-04-2020.

[74] Texas Instruments. Ti e2e coummunity. https://goo.gl/dPbNkJ, 2018. Accessed: 11-04-2020.

[75] Texas Instruments. Ulp meets energy harvesting. https://bit.ly/2GkBkOg, 2019. Accessed: 11-04-2020.

[76] Oana Iova, Pietro Picco, Timofei Istomin, and Csaba Kiraly. Rpl: The routing standard for the internet of things... or is it? *IEEE Communications Magazine*, 2016.

[77] Hassan Iqbal, Muhammad Hamad Alizai, Ihsan Ayyub Qazi, Olaf Landsiedel, and Zartash Afzal Uzmi. Scylla: Interleaving multiple iot stacks on a single radio. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018.

[78] Vikram Iyer, Rajalakshmi Nandakumar, Anran Wang, Sawyer B. Fuller, and Shyamnath Gollakota. Living iot: A flying wireless platform on live insects. In *International Conference on Mobile Computing and Networking (MobiCom)*, 2019.

[79] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. Quick recall: A hw/sw approach for computing across power cycles in transiently powered computers. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2015.

[80] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, 2014.

[81] Hrishikesh Jayakumar, Arnab Raha, Jacob R Stevens, and Vijay Raghunathan. Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 2017.

[82] S. Khanna, S. C. Bartling, M. Clinton, S. Summerfelt, J. A. Rodriguez, and H. P. McAdams. An fram-based nonvolatile logic mcu soc exhibiting 100retention at $rmVDD = 0$ v achieving zero leakage with lt; 400-ns wakeup time for ulp applications. *IEEE Journal of Solid-State Circuits*, 2014.

[83] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, 1986.

[84] P. Koopman. *Better Embedded System Software*. CMU Press, 2010.

[85] Ulf Kulau, Felix Büsching, and Lars Wolf. Undervolting in wsns: Theory and practice. *Internet of Things Journal, IEEE*, 2015.

[86] Ulf Kulau, Felix Büsching, and Lars Wolf. Idealvolting: Reliable undervolting on wireless sensor nodes. *ACM Trans. Sen. Netw.*, 2016.

[87] Ulf Kulau, Stephan Rottmann, Sebastian Schildt, Johannes Balen, and Lars Wolf. Undervolting in real world wsn applications: A long-term study. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2016.

[88] PERSIST Lab. Rf trace. `https://github.com/PERSISTLab/BatterylessSim/tree/master/traces`, 2018. Accessed: 11-04-2020.

[89] Olaf Landsiedel, Hamad Alizai, and Klaus Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, 2008.

[90] Eunji Lee, Seunghoon Yoo, Jee-Eun Jang, and Hyokyung Bahn. Shortcut-jfs: A write efficient journaling file system for phase change memory. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[91] Juchang Lee, Kihong Kim, and Sang Kyun Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proceedings 17th International Conference on Data Engineering*, 2001.

[92] Fuyang Li, Keni Qiu, Mengying Zhao, Jingtong Hu, Yongpan Liu, Yong Guan, and Chun Jason Xue. Checkpointing-aware loop tiling for energy harvesting powered nonvolatile processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[93] Xiangyu Li. Dynamic voltage-frequency and workload joint scaling power management for energy harvesting multi-core wsn node soc. *Sensors*, 2017.

[94] Libelium. Waspmote. `http://www.libelium.com/products/waspmote/`, 2017. Accessed: 11-04-2020.

[95] Kris Lin, Jennifer Yu, Jason Hsu, Sadaf Zahedi, David Lee, Jonathan Friedman, Aman Kansal, Vijay Raghunathan, and Mani Srivastava. Heliomote: enabling long-lived sensor networks through solar energy harvesting. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, 2005.

[96] Xue Lin, Yanzhi Wang, Siyu Yue, Naehyuck Chang, and Massoud Pedram. A framework of concurrent task scheduling and dynamic voltage and frequency scaling in real-time embedded systems with energy harvesting. *Proceedings of the International Symposium on Low Power Electronics and Design*, 2013.

[97] Shaobo Liu, Qinru Qiu, and Qing Wu. Energy aware dynamic voltage and frequency selection for real-time systems with energy harvesting. In *Proceedings of the conference on Design, automation and test in Europe*, 2008.

[98] Yongpan Liu, Huazhong Yang, Robert P. Dick, Hui Wang, and Li Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *Proceedings of the 8th International Symposium on Quality Electronic Design*, 2007.

[99] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. In *LIPIcs-Leibniz International Proceedings in Informatics*, 2017.

[100] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 2015.

[101] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[102] Giedrius Lukosevicius, Alberto Rodriguez Arreola, and Alexander Weddell. Using sleep states to maximize the active time of transient computing systems. In *ENSsys (with SenSys)*, 2017.

[103] Shaoheng Luo, Cheng Zhuo, and Houle Gan. Noise-aware dvfs transition sequence optimization for battery-powered iot devices. In *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[104] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages*, 2017.

[105] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018.

[106] R. Margolies et al. Demo: An adaptive testbed of energy harvesting active networked tags (EnHANTs) prototypes. In *IEEE INFOCOM'13*, 2013.

[107] mbed. Iot os. `goo.gl/u918jX`, 2017. Accessed: 11-04-2020.

[108] Scott Meninger, Jose Oscar Mur-Miranda, Rajeevan Amirtharajah, Anantha Chandrakasan, and Jeffrey H Lang. Vibration-to-electric energy conversion. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2001.

[109] Geoff V. Merrett, Neil M. White, Nick R. Harris, and Bashir M. Al-Hashimi. Energy-aware simulation for wireless sensor networks. In *Proceedings of the Sixth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON 2009, June 22-26, 2009, Rome, Italy*, 2009.

[110] Kresimir Mihic, Ajay Mani, Manjunath Rajashekhar, and Philip Levis. Mstore: Enabling storage-centric sensornet research. In *IPSN*, 2007.

[111] Pieter De Mil, Bart Jooris, Lieven Tytgat, Ruben Catteeuw, Ingrid Moerman, Piet Demeester, and Ad Kamerman. Design and implementation of a generic energy-harvesting framework applied to the evaluation of a large-scale electronic shelf-labeling wireless sensor network. *EURASIP J. Wireless Comm. and Networking*, 2010.

[112] Rex Min, Travis Furrer, and Anantha Chandrakasan. Dynamic voltage scaling techniques for distributed microsensor networks. In *Proceedings of the IEEE Computer Society Annual Workshop on VLSI (WVLSI'00)*, 2000.

[113] Mouser. Data sheet. `http://www.ti.com/lit/ds/symlink/sn74lv175a.pdf`, 2018. Accessed: 11-04-2020.

[114] Mouser. Data sheet. `https://eu.mouser.com/datasheet/2/916/74HC_HCT85-1597760.pdf`, 2018. Accessed: 11-04-2020.

[115] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, 2005.

[116] NASA. International space station. `t.ly/Jgmby`, 2018. Accessed: 11-04-2020.

[117] Robert HB Netzer and Mark H Weaver. Optimal tracing and incremental re-execution for debugging long-running programs. In *PLDI*, 1994.

[118] Kevin J Nowka, Gary D Carpenter, Eric W MacDonald, Hung C Ngo, Bishop C Brock, Koji I Ishii, Tuyet Y Nguyen, and Jeffrey L Burns. A 32-bit powerpc system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling. *IEEE Journal of Solid-State Circuits*, 2002.

[119] Expansion of STM32 Nucleo boards. Data sheet: X-nucleo-nfc02a1. `www.st.com`, 2017. Accessed: 11-04-2020.

[120] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2011.

[121] Terence Parr. *The Definitive ANTLR 4 Reference*. `goo.gl/RR1s`, 2013.

[122] Padmanabhan Pillai and Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[123] Dilushi Piumwardane, Carlos Pérez-Penichet, Christian Rohner, and Thiemo Voigt. Backscatter communication for wireless robotic materials. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2019.

[124] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, 1995.

[125] Harry C. Powell, Adam T. Barth, and John Lach. Dynamic voltage-frequency scaling in body area sensor networks using cots components. In *Proceedings of the Fourth International Conference on Body Area Networks*, 2009.

[126] Shashank Priya, Chih-Ta Chen, Darren Fye, and Jeff Zahnd. Piezoelectric windmill: a novel solution to remote sensing. *Japanese journal of applied physics*, 2004.

[127] Xiongpai Qin, Yanqin Xiao, Wei Cao, and Shan Wang. A parallel recovery scheme for update intensive main memory database systems. In *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 509–516, 2008.

[128] Daler Rakhmatov and Sarma Vrudhula. Energy management for battery-powered embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2003.

[129] Brian Randell, Pete Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 1978.

[130] Benjamin Ransford. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2011.

[131] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014.

[132] Alberto Rodriguez, Domenico Balsamo, Zhenhua Luo, Steve P Beeby, Geoff V Merrett, and Alex S Weddell. Intermittently-powered energy harvesting step counter for fitness tracking. In *2017 IEEE Sensors Applications Symposium (SAS)*, 2017.

[133] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff Merrett, and Alex Weddell. Restop: Retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 2018.

[134] ROHM. Data sheet. `https://www.rohm.com/datasheet/BU4910F/bu48xxg-e`, 2015. Accessed: 11-04-2020.

[135] Noboru Sakimura, Yukihide Tsuji, Ryusuke Nebashi, Hiroaki Honjo, Ayuka Morioka, Kunihiko Ishihara, Keizo Kinoshita, Shunsuke Fukami, Sadahiko Miura, Naoki Kasai, et al. 10.5 a 90nm 20mhz fully nonvolatile microcontroller for standby-power-critical applications. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, 2014.

[136] M. E. Salehi, M. Samadi, M. Najibi, A. Afzali-Kusha, M. Pedram, and S. M. Fakhraie. Dynamic voltage and frequency scheduling for embedded processors considering power/performance tradeoffs. *IEEE Trans. Very Large Scale Integr. Syst.*, 2011.

[137] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement*, 2008.

[138] Antonio Sánchez, Salvador Climent, Sara Blanc, Juan Vicente Capella, and Ignacio Piqueras. Wsn with energy-harvesting: Modeling and simulation based on a practical architecture using real radiation levels. In *Proceedings of the 6th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, 2011.

[139] Michael D Scott, Bernhard E Boser, and Kristofer SJ Pister. An ultralow-energy adc for smart dust. *IEEE Journal of Solid-State Circuits*, 2003.

[140] Dennis G Severance and Guy M Lohman. Differential files: their application to the maintenance of large databases. *ACM Transactions on Database Systems (TODS)*, 1976.

[141] Jalpa Shah and Biswajit Mishra. Iot enabled environmental monitoring system for smart cities. In *Internet of Things and Applications (IOTA), International Conference on*, 2016.

[142] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. Llbt: An llvm-based static binary translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2012.

[143] Yiwen Shen, Xiang Meng, Qixiang Cheng, Sébastien Rumley, Nathan Abrams, Alexander Gazman, Evgeny Manzhosov, Madeleine Strom Glick, and Keren Bergman. Silicon photonics for extreme scale systems. *Journal of Lightwave Technology*, 2019.

[144] Victor Shnayder, Mark Hempstead, Bor-Rong Chen, and Matt Welsh. PowerTOSSIM: Efficient Power Simulation for TinyOS Applications. In *ACM SenSys*, 2004.

[145] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Energy-efficient design of battery-powered embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2001.

[146] Rebecca Smith and Scott Rixner. Surviving peripheral failures in embedded systems. In *USENIX Annual Technical Conference*, 2015.

[147] IXYS SolarMD. Slmd481h08l. `http://ixapps.ixys.com/`, 2018. Accessed: 11-04-2020.

[148] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnan Narayanan. Nonvolatile processors: Why is it trending? In *2017 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2017.

[149] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[150] Florin Sultan, Thu Nguyen, and Liviu Iftode. Scalable fault-tolerant distributed shared memory. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.

[151] Cristiano Tapparello, Hoda Ayatollahi, and Wendi Heinzelman. Energy harvesting framework for network simulator 3 (ns-3). In *Proceedings of the 2Nd International Workshop on Energy Neutral Sensing Systems*, 2014.

[152] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[153] Ramakrishnan Venkitaraman and Gopal Gupta. Static program analysis of embedded executable assembly code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '04, 2004.

[154] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *ESSCIRC (ESSCIRC), 2012 Proceedings of the*, 2012.

[155] Mimi Xie, Mengying Zhao, Chen Pan, Jingtong Hu, Yongpan Liu, and Chun Jason Xue. Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor. In *Proceedings of the 52nd Annual Design Automation Conference*, 2015.

[156] Kasim Sinan Yildirim, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2018.

[157] Baoxian Zhao, Hakan Aydin, and Dakai Zhu. Generalized reliability-oriented energy management for real-time embedded applications. In *Proceedings of the 48th Design Automation Conference*, 2011.