A Survey and Experimental Analysis of Checkpointing Techniques for Energy Harvesting Devices

Priyanka Singla[a,*], Smruti R. Sarangi[a]

[a] *School of Information Technology, IIT Delhi, New Delhi, India 110016*

Abstract

With the advent of ultra-low-power embedded processors, energy harvesting devices (EHDs) are becoming exceedingly prevalent. These devices are highly portable, self-sustainable, and once deployed, they can run for an extremely long time. They can thus be installed at hard-to-reach locations. Despite the benefits, it is challenging to use these devices as they rely on sporadic and variable sources of ambient energy, and are equipped with very small memories. The intermittent nature of the ambient energy leads to a loss of device state. Such repeated failures might cause non-termination of the programs executing on these devices. To achieve termination, we need to use state retention techniques that guarantee the programs' forward progress.

Checkpointing is the most common state retention technique. However, performing checkpointing arbitrarily can lead to inefficient and incorrect execution of the program. Thus, several approaches have been proposed to perform checkpointing intelligently. In this paper, we present a comprehensive survey of these checkpointing techniques. We also performed an extensive evaluation of 13 state-of-the-art approaches and showed detailed time and energy figures for these approaches. Our comparison provides dual benefits: (i) it tells the reader which classes of checkpointing approaches are the best, (ii) it shows the sensitivity of performance with respect to various external factors such as the nature of the energy source and the energy storage capacity.

This survey will help researchers quickly understand the complexities and issues involved in creating checkpointing schemes. It will further enable them to efficiently design their programs by choosing the best checkpointing mechanism according to their requirements.

Keywords: Energy Harvesting Devices, Correctness, Efficiency, Instrumentation, Checkpointing, Nonvolatile Memory.

1. INTRODUCTION

Energy Harvesting Devices (EHDs) are emerging as an important class of IoT (Internet of Things) devices. Unlike the traditional IoT devices that rely on stable sources of

[*] Corresponding author

Email addresses: `priyanka@cse.iitd.ac.in` (Priyanka Singla), `srsarangi@cse.iitd.ac.in` (Smruti R. Sarangi)

power such as batteries, these devices extract energy from the ambient sources: solar, RF signals, thermal, and vibrations. The interest in such devices is primarily due to the feasibility of their deployment in remote and hard-to-reach locations, e.g., arctic oil pipelines, under-sea structures, inside caves, and hazardous environments such as within volcanoes or nuclear reactors. The EHD market has already reached 500 million USD and is projected to grow 10% per year [1].

Insufficiency of Battery-powered IoT Devices: It is important to understand the issues related to traditional battery-equipped IoT devices. The batteries generally have a large form factor that makes the device less portable. Even though batteries have a high energy density, their limited lifetime results in a low power density [2]. Furthermore, when the batteries deplete, we need to replace them. Replacing batteries is hard and expensive as it requires human, or in rare cases, robotic intervention. Moreover, the disposal of billions of batteries every year is dangerous to the environment. They contain toxic substances such as lead, nickel, and mercury, which are very expensive to recycle [3]. Finally, recycling is not an entirely friendly process as it pollutes the environment and degrades the quality of human life.

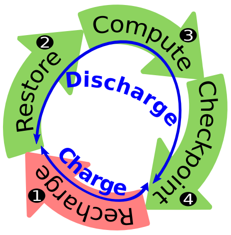
Thus, to mitigate the periodic replacement issue, several approaches propose to use self-sustainable capacitor-based energy harvesting devices. EHDs can harvest power in the range of microwatts to a few milliwatts, typically required by IoT devices such as a wireless sensor node [4]. Though these EHDs might also employ toxic or rare earth metals, these poisonous elements are tightly encapsulated in compounds such as cadmium telluride and cadmium selenide. Hence, the toxic contents are highly unlikely to be released while the device is in use or upon its disposal [5].

Intermittent Computing in EHDs: Ambient sources are highly sporadic and unpredictable. To smooth out the incoming power variations and provide a stable power supply to the EHD's load, an energy storage unit (a rechargeable battery or a supercapacitor) is used. However, at times of insufficient incoming power supply, the device may fail due to energy depletion. Thus, large energy storage is used so that sufficient energy can be harvested and stored. These EHDs are called energy-neutral systems [6], and they formally follow the equations mentioned below:

$$\int_0^T P_h(t)dt = \int_0^T P_c(t)dt \tag{1}$$

$$V_{CC}(t) \geq V_{min}, \forall t \tag{2}$$

where $P_h(t)$ and $P_c(t)$ represent the instantaneous harvested and consumed power at time t, T denotes the duration for which the device is run, V_{min} is the voltage below which the device stops operating, and V_{CC} denotes the device's instantaneous voltage. If the second equation does not hold, the device is no longer energy-neutral, and it fails to operate correctly. However, if a system can function correctly despite a failure, it is called an intermittent or transient system [7]. Such systems are equipped with nonvolatile memory to backup (also known as a checkpoint) the executing program's state when the device's energy runs out. Subsequently, when sufficient energy to execute a considerable amount of the program is harvested and accumulated, the checkpoint is restored, and the execution is resumed. In this paper, we focus on the more interesting intermittent aspect of the EHDs. An intermittent EHD repeatedly executes in charge-discharge cycles, and each cycle comprises four phases: ❶ recharge, ❷ restore, ❸ compute, and ❹ checkpoint (Figure 1(a)).



(a)

Topics	Existing surveys
Energy sources and energy storage technologies	[8]
Single source harvesting	[9, 10]
Multi-source harvesting	[11]
Energy conservation schemes in WSNs	[12]
Harvesters' conversion efficiency	[13]
Aspects in wireless communication	[14]
State retention in EHDs	This survey, [15, 16]

(b)

Figure 1: **(a)** Charge-discharge cycle in an EHD **(b)** Existing surveys in the context of EHDs

Significance of Checkpointing:

Checkpointing is an age-old concept that has been used for fault/failure tolerance in several systems, e.g., database/transactional systems, file systems, and other high-reliability systems. The key idea behind a checkpoint is to freeze/halt the system and store the relevant program (or system) state in secondary nonvolatile memory (generally a disk). This stored state is later used for rollbacks in case of failures. Thus, several approaches have been used in the past to checkpoint these high-end systems [17, 18]. The primary goal of these techniques is to reduce the system's recovery time.

However, it is important to discuss why these existing techniques do not apply to EHDs. Two reasons are responsible for that:

- *Type and frequency of failures*:Previous works assume that failures can arise due to several reasons within a system/program, such as a transactional failure (in databases), OS failure, disk failure, or power failure, and are entirely unpredictable. In contrast, the works on checkpointing in energy harvesting devices consider failures due to the intermittent and fluctuating nature of the ambient energy source, i.e., the cause of failure is external. Thus, depending upon the characteristic of the environment, the available device's input power can be predicted for some sources. Furthermore, unlike the failures in existing systems, which are rare events that occur occasionally, ***the power failures in EHDs are very frequent***. The impact of failures can be reduced by overzealous checkpointing, however, this comes at the cost of a higher runtime overhead of the program.

- *Small scale nature of the EHDs*: As mentioned previously, EHDs are highly resource-constrained devices with ***limited memory storage and energy***. This is unlike high-end systems used in the previous works where resource limitations are not an issue.

- *Size of the checkpoint*: Energy harvesting devices perform checkpointing at the code level, i.e., they checkpoint certain variables or the program stack. In contrast, existing systems perform checkpointing extrinsically (full state of the application or the entire memory). Furthermore, the size of the checkpoint in conventional systems is extremely large. For example, a checkpoint in databases includes flushing all the dirty pages (which include large tables) and the log files. Several high-end systems backup several such databases; this further increases the checkpoint size. The checkpoint's size is proportional to the energy consumption.

Hence, existing techniques are not a viable option for energy harvesting devices. Therefore, new techniques are required that emphasize minimizing the data to be checkpointed. Thus, a lot of research has been going on to reduce the amount of data to be checkpointed. Furthermore, we cannot use traditional mechanisms to decide when to take a checkpoint. If many checkpoints are taken during the program's execution, a large amount of energy and time will be consumed, which could have otherwise been used to execute more program instructions. In contrast, very few checkpoints would result in many failed executions and non-termination in the worst case. Since energy is not an issue in high-end systems, they do not have any such issues with respect to non-termination.

Considering the aforementioned points, we need to have energy-aware checkpointing techniques with extremely low overheads.

Apart from efficiency, if a checkpoint is trivially performed, it can result in issues related to correctness, program termination, and data timeliness (discussed later in §3.3). In this survey, we shall discuss various approaches for efficient and correct checkpointing.

1.1. Motivation for this Survey

There is a lot of commercial interest in devising sensor-equipped EHDs due to their extreme energy efficiency, small size, and an almost infinite lifetime. These devices are being used in numerous applications spanning various fields such as biomedical appliances, smart homes, smart cities, and structural or environmental monitoring. Academia is also actively involved, and researchers are proposing novel methods to implement these applications, using their technical expertise from domains of AI [19], wireless networks [14], and security [20].

Checkpointing is an integral part of intermittent systems, irrespective of the application areas. With technological advancements such as (i) the transition of nonvolatile memories from traditional flash-based SSDs to contemporary energy-efficient and fast FRAMs, MRAMs, and STT-MRAMs and (ii) bespoke processors for energy harvesting applications, checkpointing techniques are increasingly becoming more sophisticated. Thus, we view this survey as well-timed because it summarizes most of the research done in checkpointing in energy harvesting systems in the last 10 years. Furthermore, we present a thorough comparison and an extensive evaluation of various state-of-the-art checkpointing approaches. Designers and researchers in this space can use this survey to understand, appreciate, and design state-of-the-art checkpointing systems.

1.2. Scope

Though there exists numerous literature on traditional IoT systems, very few survey papers thoroughly cover the EHD systems. Figure 1(b) lists the existing surveys on EHDs, and these works cover topics such as ambient energy sources, their conversion efficiencies, and energy storage technologies. They also extensively study the energy conservation schemes and communication aspects from the perspective of wireless sensor networks. The salient feature of these papers is that they primarily focus on the physical (device level) aspects of harvesting energy and storing it.

In comparison, our survey is at a much higher level – it is at the architecture and software level. The initial part of the paper provides an overview of the architecture of EHDs, various state retention mechanisms, and approaches for correct program execution in these devices. The latter part is dedicated to a detailed discussion of novel checkpointing approaches. Some recent surveys have in parallel worked on state retention in energy harvesting-based systems [15, 16]. However, despite the same basic idea of state retention, these surveys differ considerably from our work. The study by Qiu et al. [15] considers only EHDs using NVPs (non-volatile processors) and NVAs (non-volatile accelerators) that have circuit and architecture level support for intermittent computing. Considering such hardware, the authors present a very abbreviated overview of software techniques (both checkpointing and non-checkpointing based) that guarantee successful program execution despite the failures. Our work, in contrast, covers prior work exhaustively and presents a detailed comparison of diverse techniques used in all types of EHDs and under various energy sources.

The survey by Umesh et al. [16] is the most related in the sense that it discusses intermittent computing in energy harvesting systems. However, its primary focus is not checkpointing, and second, the taxonomy is totally different. Our paper distinguishes itself on several counts. First, our taxonomy is driven by functionality and is far more implementation-driven as compared to Umesh et al.'s work. Second, we have invested a tremendous amount of effort in implementing schemes from each class of proposals and have thoroughly compared them along several different axes. We showed detailed time and energy characteristics of different approaches. We discussed their relative strengths and weaknesses in great detail and made a set of concrete recommendations. To the best of our knowledge, such a rigorous experimental study has not been done before. We have also evaluated the sensitivity with respect to different capacitor sizes and ambient energy profiles. Finally, we have shown Kiviat plots that compare the approaches on a host of different metrics and can help the system designer decide which solution to choose.

In summary, the comparison provides us with dual benefits: (i) it tells the reader which classes of checkpointing approaches are the best, (ii) it shows the sensitivity of performance with respect to various external factors such as the nature of the energy source and the energy storage capacity. This thorough characterization serves as a guide for practitioners regarding which design to choose based on runtime conditions. This study is also very useful for practicing researchers because it provides actual numbers, trends, and comparative results – all implemented on a standard, baseline platform. To the best of our knowledge, this is a **novel contribution**. Neither does the paper by Umesh et al. nor does any other paper performs simulation and comparative studies quite unlike what we have done in this work.

1.3. Organization of the Paper

The organization of the paper is summarized in Figure 2. §2 describes the architecture and execution of energy harvesting systems. This section also highlights the importance of checkpointing compared to some newly proposed alternatives for intermittent execution. §3 provides an overview of checkpointing and discusses the correctness and efficiency aspects of checkpointing-based intermittent systems. §4 presents different approaches for statically instrumenting a program. §5, then discusses various approaches to choose the instrumented locations where the checkpoint should be taken. A detailed experimental analysis comparing various checkpointing approaches is presented in §6. Finally, we conclude the survey in §7.

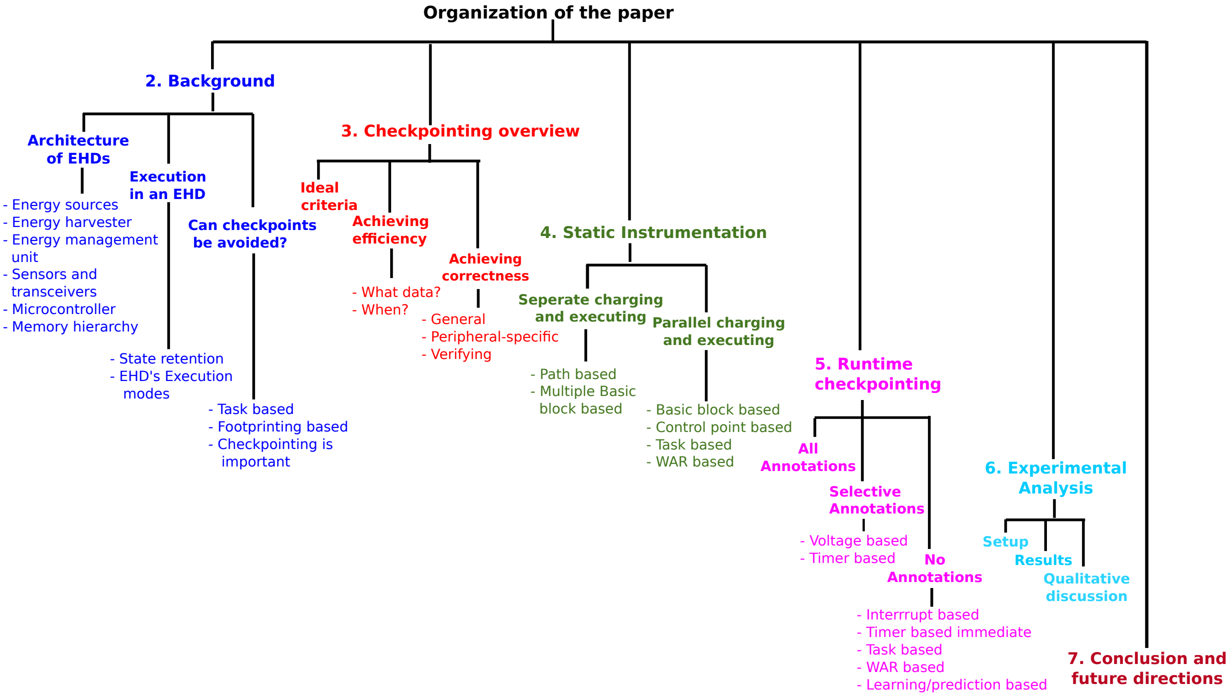


Figure 2: Organization of the paper

2. BACKGROUND

This section discusses the architecture and working of energy harvesting devices.

2.1. Architecture of EHDs

Figure 3 presents a simplified block diagram showing the various components of an EHD. These components are described below.

2.1.1. Energy Sources

The ambient sources include solar, vibrational, thermal, radio frequency (RF), and wind power. These sources differ in their power densities and power profile patterns [21]. E.g., the maximum power density of the solar source is $15\mathrm{mW\,cm^{-2}}$, whereas the vibrational source (from buildings) has only $330\mathrm{\mu W\,cm^{-3}}$. Similarly, the solar power changes very slowly (at the granularity of hours), while the vibrational power varies at the granularity of milliseconds. Table 1 summarizes the power densities of different energy sources, along with the parameters that can be configured to obtain a desired amount of energy.

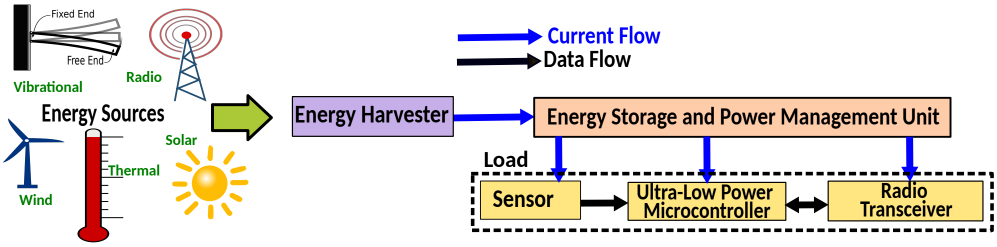


Figure 3: Architecture of EHDs

Energy Source	Harvester	Typical Power Density (μW cm^{-2})	Typical Variation Granularity	Configurable Parameters
Solar	solar panel (photovoltaic cells) [24]	14,000 (μW cm^{-3} - Direct sun) [21] 15 (Indoor lights) [23]	constant in hours [22]	area, orientation angle, number of solar cells, Artificial: number and power of bulbs, distance from bulb [24, 21]
Vibration	electromagnetic converter [23]	145 (μW cm^{-3}) [23]	periodic in milliseconds [25]	amplitude and frequency of mechanical stimulus, vibration acceleration, device's dimensions, magnetic field strength, coil's length [23, 21]
Vibration	electrostatic converter [23]	50 (μW cm^{-3}) [23]	periodic in milliseconds [25]	vibration acceleration, mass and size of the device, length of the conductors and the distance between them [23, 21]
Vibration	piezoelectric converter [23]	330 (μW cm^{-3}) [23]	periodic in milliseconds [25]	piezoelectric material, resonant frequency of source, peak acceleration, mass and size of device, number of sources [25]
Shoe Inserts	piezoelectric converter [26]	330 (μW cm^{-3}) [21]	linear in milliseconds, periodic in seconds [26]	walking/running speed [26]
Thermal	thermoelectric generators [24]	20-60 [24]	fluctuating in seconds or hours; application dependent	temperature difference, material properties (Seebeck coefficients) [24]
RF	antenna and a converter [24]	50,000 (directed) [23] 0.012 (ambient) [23]	fluctuating in seconds [27]	source power, distance from source antenna gain, number of sources [24]
Wind	wind turbines, rotors [24]	28,500 [24]	fluctuating in seconds [23]	height (chord length), wind speed, number and length of blades [24]

Table 1: Characteristics of various ambient energy sources

2.1.2. Energy Harvester

The incoming energy from various sources is converted into electrical energy by the corresponding energy harvester (Table 1). Each harvester has an energy conversion circuit, and the output energy of the harvester depends upon its *conversion efficiency*. E.g., the RF harvester has a receiving antenna, a combination matching network/bandpass filter, a rectifying circuit, and a low-pass filter [13]. Each of these components loses some amount of energy to the ambient and is further constrained by limits imposed by physics. E.g., there are fundamental thermodynamic limits to the process of converting heat into electrical energy. We cannot design a more efficient system than the Carnot engine [28]. Thus, the efficiency of these converters is generally far less than 100%. The converter efficiency also depends upon the EHD's load. E.g., to achieve maximum output for a given power intensity in a solar harvester, the solar panel should be loaded to its *Maximum Power Point (MPP)* [29], which is defined as the point on the V-I curve where the power output is maximum.

2.1.3. Energy Storage and Power Management Unit

The EHDs have an energy storage unit for storing the harvested energy and a management unit that controls the device's charging and discharging process. Such a unit provides the desired voltage to the load [30]. The energy storage unit can either be a capacitor or a battery, each having different characteristics. Batteries have a high energy density (151-532 mW h cm^{-3}) compared to capacitors (3.8-6.4 mW h cm^{-3}) [2]. However, batteries have a limited lifetime resulting in their low power density. Furthermore,

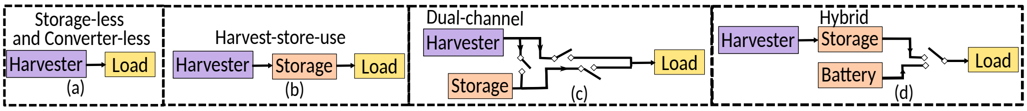


Figure 4: Different energy storage architectures

capacitors can tolerate 100-1000 times more charge-discharge cycles than a battery [24].

The storage element's size has to be carefully chosen. A large size would result in a larger charging time. In contrast, a small storage unit can be charged quickly; however, the small amount of stored energy will also be consumed rapidly. The size can be chosen considering the programs to be executed on the device. The energy consumption of the programs (for different inputs) can be analyzed, and a capacitor size that balances the charging time and failure rate should be chosen [31]. An alternative way is to use multiple capacitors, and the required number of capacitors can be dynamically reconfigured according to the program's requirement [32, 33, 34].

Apart from the capacitor-based devices, certain energy sources obviate the need for an energy storage unit. E.g., the solar source provides continuous and high power that can be directly harvested and used. In contrast, some sources are feeble, and we need some secondary storage such as a battery to keep the device working when there is no harvested energy. Thus, EHD systems can have multiple architectures described as follows (Figure 4).

(1) *Storage-less and Converter-less Architecture* [8](Figure 4(a)): In this, the energy source is directly connected to the device's load. The architecture's efficiency depends upon the instantaneous ambient energy, i.e., if the energy is insufficient to power the load, the system's efficiency drops to 0%. Thus, high power density sources (e.g., a solar source during the day or mechanical power based systems such as walking or pedaling) are best suited for this architecture.

(2) *Harvest-store-use Architecture* [35](Figure 4(b)): This uses a storage element (mostly a capacitor) to store the harvested energy. This architecture has three advantages: (i) the storage element smoothes out the variations in the incoming power, (ii) devices can operate for sometime even when there is no incoming power, (iii) the ambient source and the load can operate at different voltages by using voltage regulators/converters such as Zener diode regulators, linear voltage regulators such as a low-dropout regulator (LDO) [36], charge pumps (switched capacitor converters), and inductor-based switching converters (buck-boost converter or step-up/step-down converter) [37]. However, these converters consume power; thus, decreasing the system's efficiency.

(3) *Dual-channel Architecture* [33](Figure 4(c)): This consists of two physical channels: a direct power supply channel and another channel that transfers the energy via an energy storage unit. This architecture is beneficial when the energy storage unit is full, but energy can still be harvested. Thus, we can power the load without consuming the stored energy.

(4) *Hybrid Architecture* [30]: Also known as a "double storage energy harvesting sys-

tem" [30], this architecture supports both a capacitor (as the primary storage) and a battery (as the secondary storage); however, only one component is active at a time (Figure 4(d)). This architecture is useful for applications that need the device to be *always on* during execution. The capacitor is used when the ambient energy is high, while the battery is used in case of insufficient ambient energy.

2.1.4. Sensors and Transceivers

Sensors sense and collect the ambient parameters such as temperature and pressure, which are either processed locally or are transmitted to other devices using a transceiver. The sensing frequency can be adapted according to the energy available in the device. Transceivers form one of the most power-intensive components of an EHD. The transceivers usually exchange data at rates from 100kb/s to a few MB/s [38].

2.1.5. Ultra-Low Power MicroController Unit (ULP MCU)

Microcontrollers execute the programs and control the functionality of all the components (e.g., sensors/transceivers) in an EHD. A large number of low power processors have been developed for embedded devices such as MSP430, ARMv6-M, and TigerMIPS [27]. Most of these microcontrollers have an inorder pipeline architecture and support multiple low power modes with fast wake-up times. Researchers have proposed to augment these microcontrollers with support for execution with sporadic energy sources. Researchers have also looked at real designs or have focused on simulation studies. For example, Mementos [7] extended *mspsim* [39], an MSP430 based simulator, by incorporating models of energy harvesting and energy consumption. Similarly, Thumbulator [40] is a cycle-accurate ARMv6-M instruction set simulator with EHD features.

2.1.6. Memory Hierarchy in EHDs

EHDs typically use SRAM arrays for storing the volatile data, and nonvolatile memories such as Flash, FRAM, or nvSRAM – where each SRAM cell is connected to a nonvolatile element, forming a bit-to-bit connection so that the data in SRAM cells can be directly transferred to the nonvolatile element upon a power failure [41].

Nonvolatile Memories (NVMs): EHDs initially used Flash memory [7], but due to its enormous energy-time overheads in write/erase operations, it is now being replaced by other NVM technologies such as ferroelectric RAM (FRAM), magnetoresistive RAM (MRAM), Spin-transfer torque RAM (STT-RAM), and Phase Change Memory (PCM). FRAM is the most popular [40]; as of 2020 it is commercially available.

2.2. Execution of the EHD

Having discussed the various components, we now discuss the execution of a typical EHD that follows the 'harvest-store-use' architecture. As shown in Figure 5(a), the device's energy storage unit initially has no charge in it. It is then charged with the harvested energy. Once a sufficient amount of energy is accumulated, the device starts executing. As time progresses, the storage unit discharges, and eventually, the device shuts down when the stored energy gets depleted. It is then recharged when ambient energy is available, and the device reboots and starts executing again.

Program's State: It is expected that the EHD will often run out of energy. It needs to enter a quiescent state if it has no energy left. Furthermore, because of leakage currents, it will not be able to maintain the data in its SRAM arrays. This means that when it wakes up again, it needs to restart from a fresh state. To ensure the forward progress of the program running on the EHD, it is necessary to retain the program's state before the EHD becomes quiescent and restore it when the EHD has enough energy to resume execution. The program's state is composed of the memory state (which includes the runtime stack, heap memory, and static data memory), register state (i.e., the CPU register file), peripheral state (both on and off-chip), and the program counter. The program counter could be a part of the CPU register file as in the case of the ARM architecture or can be a separate entity as in the x86 architecture. So, in general, we can consider the program counter a separate component of the application state.

2.2.1. State Retention

The program's state is retained by storing it in some data storage element where it will remain intact with *low* or *no power*. Figure 5(b) describes different mechanisms for state retention. State retention can be done either by running an SRAM array in ultra-low power (ULP) mode or by using NV memory. State retention can be of two types: (i) *in-place* – the data remains at its original location, and (ii) *out-of-place* – the data is moved out from SRAM to a state retaining memory.

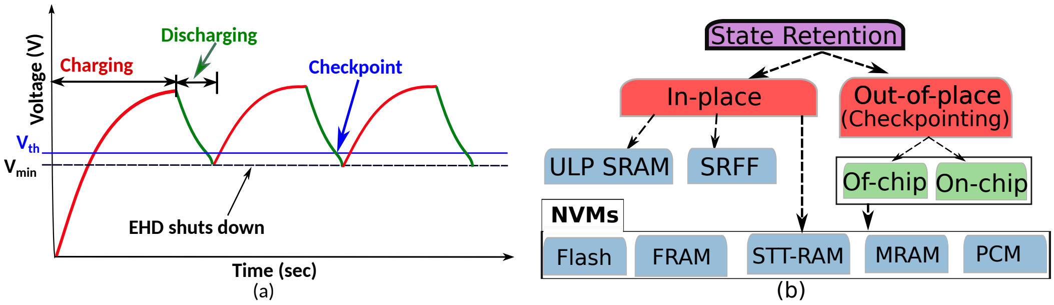


Figure 5: (a) Capacitor's voltage profile (adapted from [32]), (b) State retention mechanisms.

In-place State Retention: Jayakumar et al. [42] observed that the minimum voltage required to retain the contents of an SRAM array is around 10× lower than the minimum operating voltage of a typical MCU. So, they introduced an extremely low-power mode that retains all the data, making the SRAM array virtually nonvolatile. Particularly, the authors observed that with a voltage as low as $220mV$, SRAM cells could retain data for infinite time [42]. Apart from SRAM, the registers can be replaced by State Retentive Flip-Flops (SRFFs) powered by auxiliary supplies such as a tiny battery. A battery of a size of $50\mu Ah$ can store 10 kilobits for over two years [43]. Furthermore, Williams et al. [44] observed that most of the intermittent systems have very short power-off times ($< 1s$), and even after a power loss, the remaining charge in the device is sufficient to maintain the data in the SRAM array. Thus, in some cases where the power-off times are low, SRAM memory can be safely used for state retention. Alternatively, the SRAM can be replaced by NVM, which can sustain power failures but have high access energy and time.

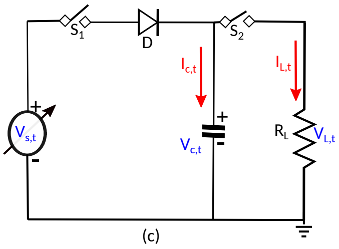
Out-of-place State Retention: Also known as *checkpointing*, this method backs up the data on an off-chip or an on-chip NVM. Off-chip checkpointing stores the backup on a remote external nonvolatile memory, accessing which is both energy and time consuming. In contrast, on-chip checkpointing gets the proximity benefits because the NVM is integrated with the processor's CMOS circuits. However, using an on-chip NVM is still not very efficient as the data in the processor's flip-flops has to be sequentially copied to/from the centralized NVM [45]. A solution is to use nonvolatile processors (NVPs), which have nonvolatile flip-flops (NVFFs). Each NVFF is attached to a standard volatile flip-flop, enabling a parallel bit-to-bit transfer, which results in energy and time efficiency [22, 46]. However, this achievement is at an increased area cost and leakage energy due to the NVFFs.

The backup can be taken either in hardware (using DMA [47, 48]) or in software (using memcpy), the former being more efficient.

2.2.2. EHDs' Charging and Execution Modes

The discussion until now assumed that charging and execution does not happen simultaneously. However, it is possible to charge the device while executing a program. The two scenarios are described below with a simplified circuit diagram (Figure 6) of an energy harvesting system with 'dual-channel architecture'. For simplicity, we have not shown the voltage regulators. The circuit has a variable voltage source with $V_{s,t}$ as its instantaneous voltage that corresponds to the harvested power. The diode(D) prevents the back-flow of charge to the harvester [49]. R_L denotes the resistance of the load (MCU, sensors, and transceivers) and $V_{L,t}$ is the voltage across it at time t. The other notations have their usual meanings.

Depending on the closed switches (S_1,S_2), the circuit operates in different modes. The circuit operates in 'separate charge-execution mode' if only one switch is closed at a time; else, if both the switches are simultaneously closed, the device is said to be in 'parallel charge-execution mode.' We describe the two modes as follows:



(a)

Notation	Description
$S1$, $S2$	Switches for connecting the circuit
$V_{s,t}$	Voltage corresponding to the power harvested at time t
$V_{c,t}$	Voltage across the capacitor at time t
D	Diode to prevent back-flow of charge to harvester
$V_{L,t}$	Voltage across the load at time t
R_L	Resistance of the load (MCU, sensors and transceivers)

(b)

Figure 6: **(a)** Simplified circuit diagram of an EHD **(b)** Description of the various circuit components

(1) Separate charge-execution mode: The switches are used in a controlled manner such that only one circuit is active at a time. The capacitor always drives the load, and the voltage across the load is $V_{L,t} = V_{c,t} - I_{L,t} * R_s$. A voltage regulator can be used to obtain constant voltage across the load.

(2) Parallel charge-execution mode: If $V_{s,t} > V_{c,t}$, then $V_{s,t}$ would simultaneously charge the capacitor and drive the load. Once the capacitor gets fully charged, the capacitor acts as an open circuit, and the entire harvested power can be used to drive the load. Finally, if $V_{s,t} \leq V_{c,t}$, the load would be driven by the capacitor alone as the potential across the diode will be negative, disconnecting $V_{s,t}$ from the system.

2.3. Can Checkpoints be Avoided?

Checkpointing is the most common state retention technique in EHDs. Along with its benefits, checkpointing has its issues. Due to these issues, several approaches have been proposed to eliminate checkpointing. In this subsection, we present these alternative approaches and discuss why it is more important to warrant a dedicated survey despite the issues in checkpointing.

Limitations of checkpointing: We discuss the limitations from four perspectives: overheads, correctness, feasibility, and termination.

(i) *Checkpointing overheads*: A significant issue with checkpointing is that it requires suspension of the executing program [50], thus, increasing the program's total running time. This overhead increases with the size of the checkpoint – the larger the checkpoint, the longer the program's suspension.

Along with the suspension time, checkpoints have energy, storage, and time overheads, which again are proportional to the size of the program state [51]. Furthermore, checkpoints deprive the program of the EHD's scarce resources - storage and energy.

(ii) *Correctness issues due to checkpointing*: Checkpoints, if not taken cautiously, can lead to various correctness issues, e.g., memory inconsistency [51, 50](Please refer to §3.3 for more details).

(iii) *Feasibility of a checkpoint*: Taking a checkpoint is not always feasible, e.g., when the total energy of executing a code segment between two checkpoints and taking the checkpoint is more than the energy stored in the device's capacitor. In such cases, we need to spend some effort to reduce the distance between the consecutive checkpoints.

Another scenario where checkpointing might not be possible is in systems with I/O peripherals. Sometimes, in these systems, the complete internal states of the peripherals are not accessible [52] making it infeasible to checkpoint successfully.

(iv) *Termination issues*: Checkpointing only ensures re-execution of the program upon a failure but does not guarantee its termination [53].

To summarize, checkpointing is expensive, and at times it is infeasible. Due to this, several alternative approaches have been proposed to eliminate checkpointing. Most of these approaches have been proposed for task-based systems and systems with peripherals; they can be classified as (i) task-based and (ii) footprint-based approaches.

In summary, checkpointing is expensive, and at times it is infeasible. Due to this, several alternative approaches have been proposed to eliminate checkpointing. Most of these approaches have been proposed for task-based systems and systems with peripherals and can be classified as (i) task-based and (ii) footprint-based approaches.

2.3.1. Task-based Approaches

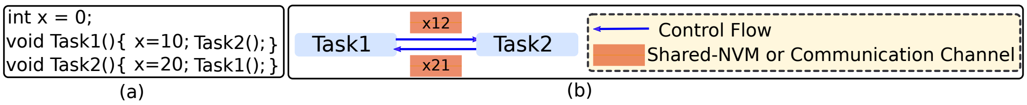
A task-based system is a system in which a program has been rewritten by dividing it into a sequence of semantically meaningful short tasks [51, 54, 55]. These tasks are idempotent (i.e.,can be executed any number of times without any change in the output), execute atomically, and each task performs some meaningful computation. These systems have two types of variables: (i) task-local – accessed only within a particular task and (ii) shared variables. Since the tasks execute atomically (§4.2.3), the task-local variables need not be stored in the NVM as other tasks do not require them. However, we need to store the shared variables in the NVM and handle them carefully.

One way to handle the shared variables is to update them in the NVM itself; however, this might result in memory inconsistency (§3.3.1). An alternative solution is data privatization [55], i.e., the task-local copies of the shared variables are created and stored in the task's volatile memory. All the updates by the task are now performed on this private copy. Upon completion, the task *atomically* commits the updates to the NV shared variables. Concurrent updates to an NV shared variable can be done (i) by adopting the concurrency control mechanisms while *using a single NV copy* or (ii) by *using multiple NV copies.*

(i) **Using a Single NV Copy** When multiple tasks try to commit simultaneously, the well-known approaches, such as optimistic concurrency control and pessimistic lock-based concurrency control, can be followed. In the former approach, the committing task ensures that the variable's value it had read (from the NVM) at the beginning of the execution has not been modified in between by any other task; else, the task aborts itself. Chen et al. [50] follow this approach. The authors further extended this approach to heterogeneous multicore systems [56].

This approach can also handle peripheral interrupts that arrive during a task's execution, e.g., sensor events. These interrupts might update a shared data structure; thus, the executing task might lose its atomicity. One way to maintain atomicity is to first finish the task and then handle the interrupt (sense and process the input). However, it is possible that when the task finishes, the ambient condition changes, and the sensed value gets changed. As followed in COATI [57], an alternative is to immediately handle the time-critical part of the interrupt (such as sensing the input) and defer the input's processing till the interrupted task finishes. The sensed value is stored and processed in a local buffer rather than the shared memory. The final value is committed to the shared memory upon interrupt's completion. In case the interrupt conflicts with the task's updates, the interrupt's updates are discarded.

This approach suffers from frequent task failures in a highly concurrent scenario. This issue can be resolved by lock-based concurrency. Each shared NV variable is associated with a lock. Tasks acquire the lock before updating a variable and release it once they finish. This eliminates the conflicts but degrades performance as only one task can acquire the lock at a time [58].



```
int x = 0;
void Task1(){ x=10; Task2(); }
void Task2(){ x=20; Task1(); }
```

Figure 7: (a) Code for two communicating tasks, (b) task model

(ii) **Using Multiple NV Copies** A solution to overcome the above approaches' issues, as used by Chain [54] is to have multiple NV copies of the shared variable, one copy for each interacting task pair. The shared copies are directional, i.e., if two tasks ($Task1$ and $Task2$) are communicating via a shared variable x (see Figure 7(a)), we maintain two NV copies – one for storing the updates from $Task1$ to $Task2$ ($x12$), and the other for storing from $Task2$ to $Task1$ ($x21$) (Figure 7(b)). Since each task writes to its NVM copy of the shared variable, this approach does not suffer from any concurrent update issue. Though this approach avoids expensive checkpointing, it accesses NVM upon every shared variable read/write, which might negatively impact the performance.

2.3.2. Footprint-based Approaches

These approaches neither checkpoint the application context nor partition the application into atomic tasks. Instead, they track a minimum amount of data, called a *footprint*. These approaches are used in peripheral-based applications. The footprint acts as the "progress indicator" of a peripheral operation. As the peripheral executes, the peripheral specific footprints are tracked in parallel and redirected to the non-volatile memory; thus, obviating the need for checkpointing.

These approaches require a deterministic and sequential application workflow. Moreover, they work best when used on peripheral-based operations, where the minimal amount of information related to the progress of a peripheral operation can be easily identified. Kang et al. [59] used them to speed up deep neural networks. The authors considered a DNN accelerator that progresses by performing the convolution operation on multiple layers. Since the accelerator has a very small amount of internal memory, it cannot store the entire DNN model or a layer. So, the model is stored in the device, and the convolution operation is divided into smaller suboperations. When a suboperation is to be performed, the corresponding data is fetched by the accelerator (from the device) and the computation's result is returned to the device. The results are then redirected to the device's NVM. Thus, the convolution operation is performed accumulatively at the granularity of suboperations. For tracking the accelerator's progress (i.e., its footprint), we only need to track the number of completed layers and the number of suboperations for the current layer.

In this approach, we relaxed the atomicity constraint from the operation-level granularity to the suboperation-level. The energy requirement of the suboperations is very small and can be easily managed with a small energy buffer. Mendis et al. [60] use this idea of relaxing the atomicity constraint while updating an e-display. Instead of updating the entire display atomically, the updates are performed incrementally across the power failures. As mentioned before, an issue with the footprinting approach is that it is not general and can only be applied to peripherals that perform sequential execution. E.g., the DNN inferencing is sequential; thus, the number of completed suboperations and DNN layer count are sufficient for a footprint.

2.3.3. Checkpointing is More Important

Table 2 summarizes the differences between the three approaches.

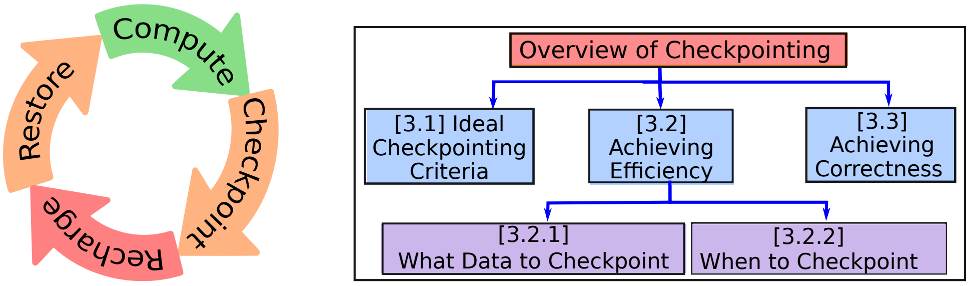
We would like to emphasize that despite its issues and various alternative approaches, checkpointing is still the most widely used approach. The primary reason is that it is a general approach and mostly applicable to all kinds of programs, unlike its alternatives. Furthermore, if we carefully observe the limitations of checkpointing, we notice that these issues are problematic if we perform checkpointing naively. However, the issues can be fixed by performing checkpointing intelligently to minimize its overheads, ensure correctness, and guarantee the program's termination. This is why a vast literature of checkpointing techniques exists: trying to reduce the checkpoint size, the number of

	Checkpointing	**Task-based methods**	**Footprint-based methods**
Can the approach be used when the internal state of the peripheral is not accessible?	No	Yes (by cautious task creation)	Yes
Applicable to any general program?	Yes	No*	No**
Heavily dependent on the program's energy estimate?	Not always	Yes	No
Programmer effort involved	Low	Very high	Low
Recovery time	Depends on the checkpoint size	Nil	Very low
*: Applicable only if the application program can be divided into tasks **: Applicable only if the application is peripheral-based and its workflow is sequential			

Table 2: Comparison of the three approaches

checkpoints and ensure the program's correctness [51, 54, 7, 61, 62, 63, 63, 25, 27]. Thus, it is important to discuss those intelligent works.

3. OVERVIEW OF CHECKPOINTING


Compute
Checkpoint
Recharge
Restore
Overview of Checkpointing
[3.1] Ideal Checkpointing Criteria
[3.2] Achieving Efficiency
[3.3] Achieving Correctness
[3.2.1] What Data to Checkpoint
[3.2.2] When to Checkpoint

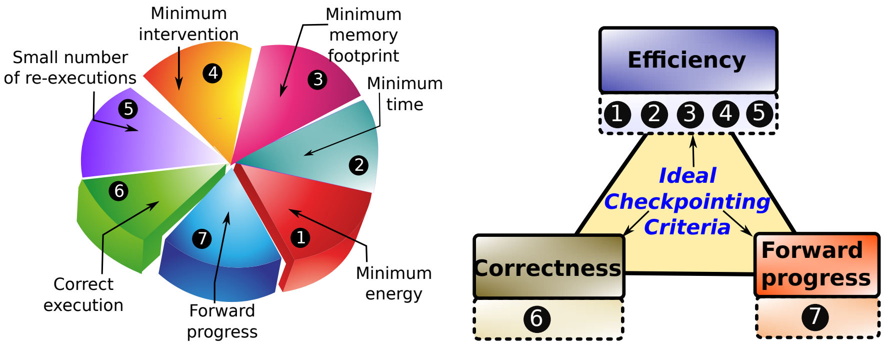
(a) (b)

Figure 8: (a) EHD's execution cycle and (b) General overview of the checkpointing procedure

Checkpointing is the most common state retention technique in EHDs and forms one of the four phases of an EHD's execution (Figure 8(a)). It is the most tractable phase and enables us to increase the device's efficiency by controlling a checkpoint's size and location. This section describes an ideal checkpointing criterion that makes the device maximally efficient and correct. Then we discuss two control aspects – (i) what data to checkpoint, and (ii) when to take a checkpoint – that helps in checkpointing efficiently. Finally, we review the various correctness issues involved in checkpointing and discuss corresponding solutions. Figure 8(b) summarizes this overview of the checkpointing procedure.

3.1. Ideal Checkpointing Criteria

The primary purpose of a checkpoint is to ensure the forward progress of an executing program, despite the EHD's intermittency. With forward progress being a necessary requirement, researchers have also emphasized other aspects of checkpointing [55, 40, 51, 64]. We combine these aspects into the criteria for ideal checkpointing (Figure 9(a)). An ideal checkpointing approach should ❶ consume minimum energy, ❷ take minimum time, ❸ consume a minimum amount of memory, ❹ involve minimum programmer intervention, ❺ result in a small number of re-executions, ❻ ensure correct program execution, and ❼



(a)

(b)

Figure 9: (a) Ideal checkpointing criteria and (b) Classified criteria for ideal checkpointing

lead to the forward progress of the program, i.e., the program should eventually terminate. We further classify the aspects into three requirements: (i) efficiency, (ii) correct execution, and (iii) forward progress (Figure 9(b)). Please note that all three criteria are important. Among these, forward progress and correctness are necessary criteria that should be followed by all the checkpointing approaches, as they avoid non-termination and incorrect execution, respectively. These two criteria are non-negotiable; they are mandatory. However, if a program is unable to terminate, it is because of some issue in checkpointing. So, we can consider non-termination (or forward progress) also as a correctness issue.

Efficiency, in contrast, is desirable. Ideally, an efficient checkpointing approach should (i) consume a very little amount of energy, (ii) take minimum time, (iii) have a small memory footprint, (iv) have minimal programmer intervention, and (v) have very infrequent re-executions; however, it is not often possible to satisfy all these requirements simultaneously. Thus, we need to prioritize these requirements as per the application. Out of these, time and energy are the most important metrics that are generally quantified in the EHD literature.

Next, we discuss the efficiency and correctness (including non-termination) requirements in the following subsections.

3.2. Achieving Efficiency

Satisfying the five efficiency-related criteria depends upon two questions: (i) ***what*** data should be checkpointed and (ii) ***when*** should the checkpointing be done during the program's execution.

3.2.1. What Data to Checkpoint

We need to checkpoint the entire program's state (§2.2), i.e., the memory state, register state, peripheral state, and the program counter. The memory state comprises the runtime stack, heap, bss region, and the initialized global variables (as shown in Figure 10(a)). This state is stored in the EHD's volatile memory (SRAM), while the NVM stores the program code and other constants. These allocations can be changed depending upon the programmer's requirement. E.g., Jayakumar et al. [63] propose to map different sections of a program to FRAM or SRAM such that the program's total energy consumption is minimized along with maintaining reliability. While checkpointing, the program's register state, peripheral subsystem's configuration registers (GPIOs), and the program counter are first pushed on to the stack, and then the entire memory state is checkpointed. Now, we discuss checkpointing of the program's memory state.

The most straightforward approach is to checkpoint the entire volatile memory (SRAM) that contains the program's memory state (Figure 10(a,b)). Though simple, this approach has $\mathcal{O}(n)$ time complexity (where n is the SRAM size), and also leads to the checkpointing of unnecessary data. So, a better approach is to checkpoint only the allocated volatile memory, i.e., memory state. This requires tracking the stack, heap, data, and bss regions. Considering that the heap, data, and bss regions occupy continuous memory (Figure 10(c)), tracking the memory state can be done using only two pointers (top of stack and heap). Though the amount of tracking is tiny, we are unnecessarily checkpointing the entire heap. The heap memory may be fragmented, providing us an opportunity to reduce the checkpoint size. This opportunity can be seized by tracking only the allocated heap fragments (Figure 10(d)); thus, checkpointing only the allocated SRAM locations.

The allocated heap fragments can be tracked by splitting the heap into fixed-size blocks. For each block, we maintain whether it contains the allocated memory. The tracking overhead depends upon the block size. A larger block size indicates a smaller number of blocks, thus, smaller tracking overhead. However, a large block might still have a lot of unallocated space that would be unnecessarily checkpointed. A small block size, in contrast, would increase the amount of tracking to be done. Thus, the block size should be carefully decided considering the overheads.

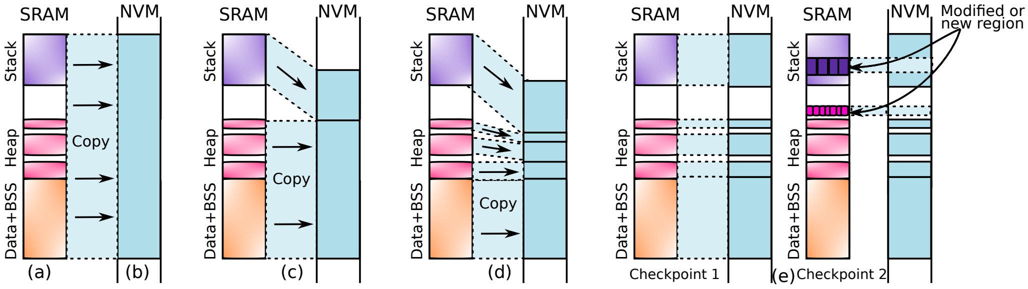


Figure 10: Different checkpointing granularities (adapted from [65]). Checkpointing the (a,b) Entire SRAM, (c) Program's memory state, (d) Allocated SRAM locations, and (e) Copy-if-change.

Tracking the heap fragments reduces the checkpoint's size considerably; however, we are still checkpointing a lot of unnecessary data. Ideally, we should checkpoint only the data modified after the previous checkpoint (Figure 10(e)). The first checkpoint comprises all the allocated regions, while the next checkpoint copies only the modified regions, i.e., copy some data only if it is changed (copy-if-change) [65]. There are several approaches to track the modified SRAM regions:

(i) Word-by-word Comparison [65]: The SRAM is divided into blocks, and each block's contents are compared to the corresponding block stored in the NVM. If the contents differ, the volatile block is checkpointed. This approach requires reading all the NVM

blocks from the previous checkpoint, i.e., equal to the SRAM size. Please note that, due to the read-intensive nature of this approach, Flash memory is appropriate for it. With Flash memory, reads are much cheaper than writes; thus, we can read and compare the blocks and then write only the modified data. In contrast, byte-addressable NVMs such as FRAM, where reads cost roughly the same as writes, could have directly written the entire SRAM. Thus, FRAM is not suitable for word-by-word comparison. Furthermore, performing block reads in Flash is cheaper than reading a block in byte-addressable NVMS (FRAM). We experimentally verified that performing a word-by-word comparison on $2KB$ of FRAM would take around 125µs, while it would roughly take roughly $50\mu s$ to read Flash memory.
(ii) Hash Comparison [66]: Another approach is to compare the hash values of the blocks. If the hash value differs, the block is copied to NVM along with the new hash value. Though this approach avoids expensive NVM block reads, it is compute-intensive. Due to this, we need to analyze its computation overhead. The summation of the compute and NVM overheads determines the total overheads of this approach. We ran the SHA-1 hash algorithm on our simulator to compute the energy and time overheads for computing the hash of 256 bytes of data. The experiment showed that computing one hash value consumed on average around 38.6µJ of energy and took around 1.7ms. Considering that FRAM writes take only 125ns for 2 bytes and FRAM writes are as fast as FRAM reads, the total time for reading the 20-byte hash digest for SHA-1 hash would only be 1.25µs. Hence, the hash computation overhead is prohibitive and is thus not the best choice for EHD-based systems.
(iii) Tracking Changes in Volatile Memory [67]: Both the aforementioned techniques perform the expensive NVM accesses for comparison. A better technique is to track the differences in the volatile memory itself. Ahmed et al. [67] track the program's global and local data. A precompiler inserts a specialized function where it detects a potential change in a global variable. Upon execution, this function updates an in-memory data structure - a bit array, used to track the modified addresses. Each bit in the array corresponds to a byte in the main memory and stores if the corresponding byte address is modified or not. The bit array has a slight overhead in the main memory (at max 12.5%) and can be updated in constant time, i.e., $\mathcal{O}(1)$ time. Unlike the global variables, the local variables can be easily tracked by monitoring the growth and shrinking of the stack. This approach appears best as there are no NVM accesses involved to determine the changed data. However, if a program frequently updates memory, the tracking overhead might eventually surpass the benefits.

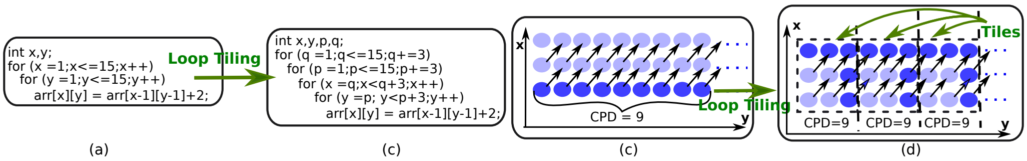
From the above discussion, we conclude the granularity at which checkpointing should be done depends on both the checkpoint size and tracking overhead. The tracking overhead depends on the data layout in memory and the program's read/write patterns. E.g., if the program data is allocated continuously and has no fragmentation, then tracking the heap blocks would be overkill. Furthermore, if the program modifies all the allocated data, it is not wise to adopt the copy-if-change.

Apart from the software approaches, we can track the changed data by using special hardware such as MPU (Memory Protection Unit) [68] or custom circuits that spies the address signal of SRAM [69].

This idea of checkpointing only the *changed data* has been used in hardware and software systems [70, 71, 72, 41, 73, 74]. Authors in [70, 71] apply this idea to FPGAs and propose to include only the changed registers in a checkpoint. Furthermore, rather than considering all the changed registers, they checkpoint only those registers that would be used in the future.

Li et al. [41] and Liu et al. [73] use the copy-if-change technique for software systems with nvSRAM based caches and checkpoint only the modified cache blocks that would be used later. An issue with using nvSRAM is that its nonvolatile part is used only at the time of checkpointing; hence it is underutilized. Furthermore, it represents a large area overhead. Xie et al. [74] address this issue by proposing a hybrid cache architecture comprising volatile SRAM and nonvolatile STTRAM blocks. During a checkpoint, the dirty SRAM blocks are copied to clean STTRAM blocks. Since the checkpoint is stored in the STTRAM blocks, it can be directly accessed after a power failure, resulting in instant execution resumption.

As seen above, the copy-if-change idea is mostly used along with another important concept - to checkpoint only the data required in the future. Both of these ideas can significantly reduce the checkpoint size. The latter concept is very beneficial in a program with nested loops [75]. Figures 11(a) and (c) present an example code and its pictorial representation, respectively. In the pictorial representation, one row of circles represents the instructions executed in one iteration. Within each row, a checkpoint is inserted after executing a fixed number of instructions. This number is called the *'checkpoint distance' (CPD)* and is either provided by the user or is equal to the *maximum* number of instructions that can be executed in one full charge-discharge cycle. In our running example, it has a value of nine (indicated by dark blue circles in Figure 11(c)). Since the output of each of these nine instructions will be used in the next iteration, we need to checkpoint all of them. Now consider another version of the same code and correspondingly new pictorial representation (Figures 11(b) and (d)). Figure 11(d) shows that the execution progresses in tiles, and only the output of boundary instructions is required to be checkpointed. Thus, despite using a CPD of 9, the checkpoint's size has decreased from 9 to 6. The key idea of loop tiling is to leverage data locality and limit the lifetime of most of the data to one tile so that the checkpoint's size is reduced.



```
int x,y;
for (x =1;x<=15;x++)
  for (y =1;y<=15;y++)
     arr[x][y] = arr[x-1][y-1]+2;
```

```
int x,y,p,q;
for (q =1;q<=15;q+=3)
   for (p =1;p<=15;p+=3)
      for (x =q;x<q+3;x++)
         for (y =p; y<p+3;y++)
            arr[x][y] = arr[x-1][y-1]+2;
```

Figure 11: Reducing the checkpoint size by loop tiling (adapted from [75])

Along with reducing the checkpoint's size, we can also reduce the amount of restored data [72]. Instead of restoring the entire checkpoint at once, the required data is restored when needed; thus, reducing the program's execution time. Apart from reducing the checkpoint/restore size, several proposals [61, 40, 76] eliminate using SRAM, and thus, they need to checkpoint only the CPU and configuration registers. However, these systems suffer from high energy and latency during normal program execution, as all the accesses are sent to the nonvolatile memory.

Atomic Checkpointing The ambient power profile's intermittent nature can result in the device's failure during checkpointing, resulting in an incomplete checkpoint. So, a mechanism is required where our program is in a safe state despite failures. This can be achieved by *double buffering*, where we maintain two checkpoints (the previous and the current), such that at least one of them is valid at all times. If the current checkpointing process fails, we can still restore the previous checkpoint. When checkpointing the entire memory, we can discard the older checkpoint. However, in the case of copy-if-change checkpointing, discarding an old checkpoint might lead to data loss, as the previous and the current checkpoints might contain disjoint memory regions. As Winkel et al. [77] proposed, we need to store all the checkpoints until a failure happens, and subsequent to a failure, we restore the partial contents of the latest checkpoint for each memory region.

3.2.2. When to Checkpoint

Checkpoints should be taken such that the executing program's termination can be guaranteed. The checkpoints should not be very distant as it would increase the chances of failure. Similarly, the checkpoints should also not be taken very frequently. It would increase the number of checkpoints, resulting in substantial energy and time overheads (violating criteria ❶ and ❷– minimum energy and time). Thus, considering the trade-off, the program can be statically analyzed to determine the checkpointing locations.

Apart from static analysis, we can choose the checkpointing locations dynamically during the program's execution. One approach is to track the device's runtime energy, and take a checkpoint just before the energy is about to deplete (also known as just-in-time (JIT) checkpointing). Several other dynamic approaches exist, which we discuss in §5. We can also use a hybrid approach, where we statically choose the checkpointing locations and dynamically decide if a checkpoint should be taken at a particular location. §4 and §5, respectively, discuss various approaches for statically annotating and runtime checkpointing.

Summary:

- What data to checkpoint and when should a checkpoint be taken are the two primary parameters in determining the efficiency of a checkpointing approach.
- Copy-if-change, along with the idea of checkpointing only the data required in the future results in the minimum checkpoint size.
- The decision of when to checkpoint can either be taken statically or dynamically.

3.3. Achieving Correctness

An intermittent device should execute a program correctly, i.e., the program's final output despite failures should be the same as its failure-free execution. Though checkpointing helps achieve the forward progress of intermittent programs, it can result in an incorrect execution if not done carefully. Different kinds of correctness issues are possible in an EHD. Some issues can arise only in peripheral-equipped EHDs. Thus, we broadly classify the correctness issues as – (i) general correctness issues and (ii) peripheral-specific correctness issues (summarized in Figure 12).

3.3.1. General Correctness Issues

Taking a checkpoint at the beginning of an instruction sequence makes that sequence idempotent, i.e., the sequence can be executed any number of times without any change

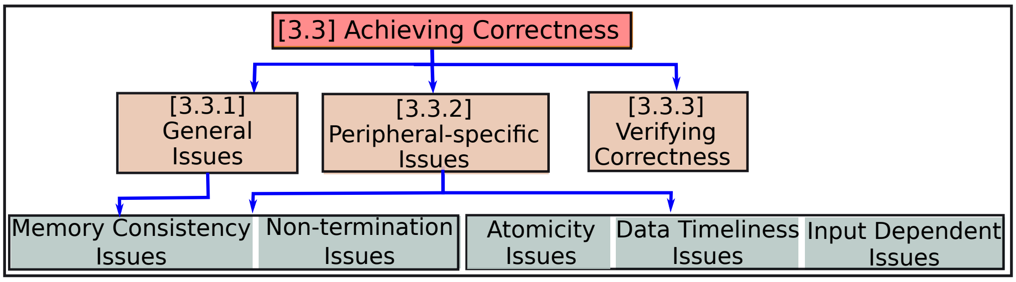

[3.3] Achieving Correctness
[3.3.1] General Issues
[3.3.2] Peripheral-specific Issues
[3.3.3] Verifying Correctness
Memory Consistency Issues
Non-termination Issues
Atomicity Issues
Data Timeliness Issues
Input Dependent Issues

Figure 12: Overview of different approaches for achieving correctness in an EHD

in the output. The checkpoint stores the program's volatile state that is restored upon a restart, nullifying the impact of the instructions executed in the previous run. However, the idempotency does not hold if a program contains both volatile and nonvolatile variables, as we might have memory consistency issues (described below).

Memory Consistency Issues: After a checkpoint, an executing program might contain modified volatile and NV variables. Upon a reboot, the volatile updates are lost, while the nonvolatile updates persist, leading to inconsistency. Figure 13 presents a scenario of an *incorrect* program execution, where the programmer defines two NV variables – an array '*buf*', and its length '*len*' (Figure 13(a)). The device reboots after updating *len* but not *buf*. The execution resumes from the last checkpoint. Since *len* was an NV variable, it was not included in the checkpoint. Thus, upon re-execution, *len* gets incremented once again (Figure 13(b)), resulting in the value '*a*' being written at the second index skipping the first (Figure 13(c)), which is incorrect. The reason behind this incorrect behavior is that we modified 'len' after reading its value in the previous execution. In particular, the inconsistency is due to the write-after-read (WAR) dependency between the load and store instructions (marked as 'A' and 'B', respectively, in Figure 13(c)) accessing the NV memory. Thus, we need carefully handle the WAR dependencies to remove this inconsistency.

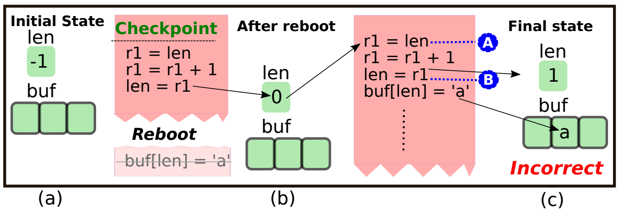


Figure 13: Memory inconsistency due to a failure (adapted from [51]). Here the variables 'len' and 'buf' are nonvolatile variables and 'r1' is a volatile register variable.

Handling the WAR Dependencies

Figure 14(a) shows a sequence of six instructions with three WAR dependencies. The instructions update two nonvolatile variables x and y. Consider that based on some technique, checkpoints have been inserted at locations 'A' and 'B' in the figure. Now we discuss various approaches to handle WAR dependencies. For each approach, we present the steps that will be followed ‘during the execution’ and ‘upon a failure.’

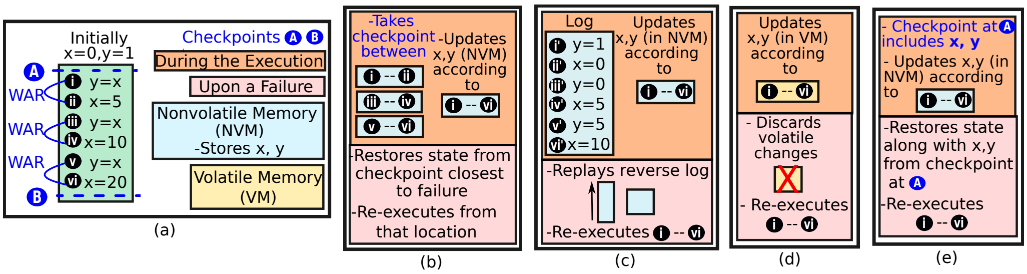

Initially
x=0,y=1
Checkpoints A B
During the Execution
Upon a Failure
Nonvolatile Memory (NVM) -Stores x, y
Volatile Memory (VM)
A
WAR
WAR
WAR
B
i y=x
ii x=5
iii y=x
iv x=10
v y=x
vi x=20
(a)
-Takes checkpoint between
i -- ii
iii -- iv
v -- vi
-Updates x,y (NVM) according to
i -- vi
-Restores state from checkpoint closest to failure
-Re-executes from that location
(b)
Log
i' y=1
ii' x=0
iii' y=0
iv' x=5
v' y=5
vi' x=10
Updates x,y (in NVM) according to
i -- vi
-Replays reverse log
-Re-executes i -- vi
(c)
Updates x,y (in VM) according to
i -- vi
- Discards volatile changes
- Re-executes
i -- vi
(d)
- Checkpoint at A includes x, y
- Updates x,y (in NVM) according to
i -- vi
-Restores state along with x,y from checkpoint at A
-Re-executes
i -- vi
(e)

Figure 14: Approaches to handle WAR dependencies (a) Code with 3 WAR dependencies, (b) Inserting a checkpoint between WAR instructions, (c) Undo logging, (d) Redo logging, and (e) Checkpointing the NV Updates

(1) *Insert a Checkpoint Between the WAR Instructions [40, 70]* This approach inserts a checkpoint between the instructions involved in a WAR dependency. As shown in Figure 14(b), three checkpoints are added between instructions (i-ii), (iii-iv), and (v-vi).
During the execution: Along with taking checkpoints, the variables are updated in the nonvolatile memory as per the instructions.
Upon a failure: The program’s state from the checkpoint closest to the failure is restored, and the execution is resumed.
This approach has a considerable overhead for programs with many WAR dependencies since a checkpoint is inserted for each WAR dependency.

(2) *Undo Logging [78, 64, 55, 52, 47]* Checkpointing on every WAR dependency can be avoided by undo logging. The current value of an NV variable is logged whenever a store instruction updates it.
During the execution: It maintains the logs in the persistent NVM (Figure 14(c)). These logs are discarded when the execution safely reaches the next checkpoint location.
Upon a failure: The logged values are replayed in the reverse order to roll back the performed updates. Then, all the instructions are re-executed when the device reboots.
Though this approach avoids writing the entire checkpoint to the NVM, it accesses the NVM twice for each store instruction: to log the current value and to update the variable. Subsequently, during restore, energy is spent in replaying the reverse log and then re-executing the instructions.

(3) *Redo Logging [62, 79]* We delay the NVM writes that are involved in WAR dependency.
During the execution: We maintain the updated values in a fixed-size volatile redo log (Figure 14(d)). Once the log is full or the next checkpoint location is reached, we dump the log to the NVM. This way, we avoid accessing the NVM on every WAR dependency, unlike in the undo logging.
Upon a failure: The volatile log is discarded, and all the instructions since the last checkpoint are re-executed upon reboot.
The approach’s performance depends upon the log size. A large log size can lead to non-termination as the ambient energy might exhaust before the log becomes full.

Maintaining a log in the above approaches is expensive as the log either needs to be copied to the NVM upon successful completion (in redo logs) or rolled back upon a failure (in undo logs). Hoseinghorban et al. [80] propose to keep two copies of the NV data variables (using two NVMs), of which one copy would be correct at all times. Each data variable has a timestamp – indicating the variable's latest update time and a location – pointing to the NVM containing the latest updated copy. It also maintains a timestamp of the latest checkpoint. In case of a failure, if a data's timestamp is larger than the checkpoint's timestamp, it indicates that the data has been updated after the checkpoint and thus might be inconsistent. So, the latest copy of the data is ignored, and the older copy is reinstated.

(4) *Checkpoint the NV Updates [51]* Another approach is to checkpoint the NV data along with the volatile data. After restoring the checkpoint, the inconsistent NV updates would be overwritten.
During the execution: After checkpointing the nonvolatile variables, the execution proceeds normally; we keep on updating the nonvolatile variables in NVM (Figure 14(e)).
Upon a failure: We restore the values of the nonvolatile variables from the checkpoint.
Tracking the WAR Dependencies
Having explained the ways to handle the WAR dependencies, let us discuss various approaches to track WAR dependencies in a program. Tracking can be done either statically or at runtime.
(1) *Static Tracking* A typical way to determine the WAR dependencies is to perform a *def-use* analysis on the NV variables, which primarily involves a backward traversal of the program's control-flow graph (CFG). For each *def* (variable definition), if the corresponding variable has already been read in the previous instructions, it indicates a WAR dependency. The *def-use* analysis is also used in performing live variable analysis that determines the range of code in which a variable may be used without being re-written. Thus, when a variable is re-written, it indicates a WAR dependency. This approach has been used in Ratchet [40], Chinchilla [64], and Alpaca [55].

Please note that WAR dependencies are not always harmful. E.g., suppose a code segment between two consecutive checkpoints contains a RAW dependency followed by a WAR to the same memory location (write-read-write sequence). In that case, even if the program fails after the second write, it is harmless. Upon re-execution, the read operation will read the value re-written by the first write.
(2) *Dynamic Tracking* Static tracking is conservative and may report many WAR dependencies that might never occur during execution; we can dynamically track only the actual dependencies visible during the program's execution [62]. We maintain two buffers – a read and a write buffer, to track the memory addresses of the variables being read or written. As the program executes, we find the load and store variables' memory addresses for each instruction. For a particular address, if the *first* access is a load, then the address is added to the read buffer, else to the write buffer. As the subsequent instructions arrive, the involved load/store addresses are checked. For the load address, if it is already in one of the buffers, it is ignored; else, it is added to the read buffer. For the store address, if it is already in the write buffer, it is ignored. However, if the address is in the read buffer, it indicates a WAR dependency.
Apart from the memory consistency issue that occurs due to the uncheckpointed NV variables, we can have other correctness issues in an EHD [81]. These issues occur when

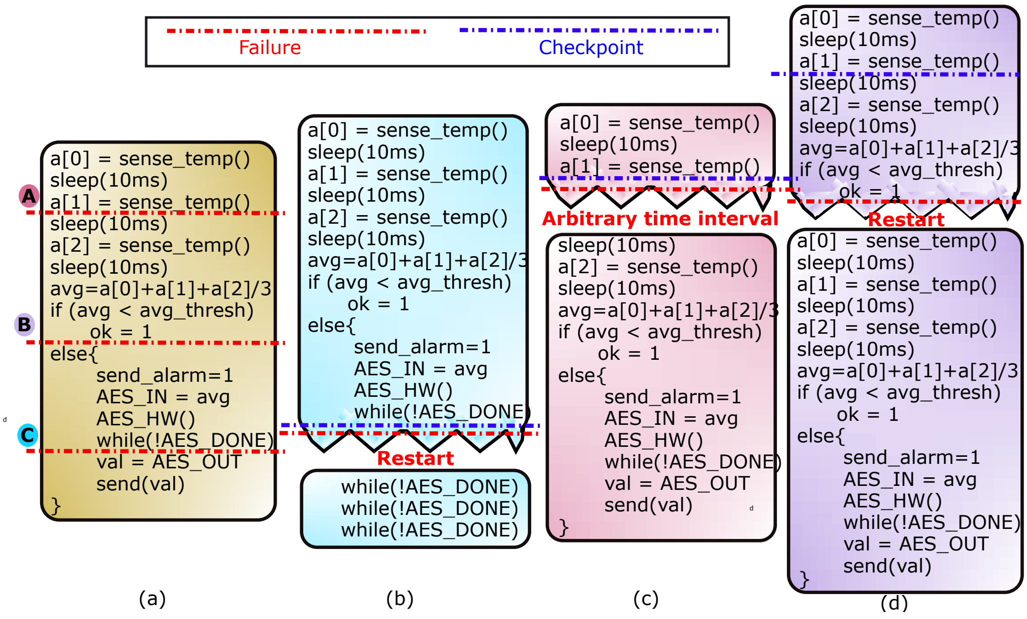
the stack and heap are stored in NVM and are not checkpointed. E.g., suppose the heap is allocated in NVM. It is possible to have a scenario where a checkpoint is taken after heap allocation, and a failure occurs after the heap deallocation. Upon restoration, we would execute from the last checkpoint and might try to deallocate an unallocated region, thus performing an incorrect execution.

Non-termination Issues: There does not exist much literature on this, as most of the checkpointing approaches in general systems ensure termination of the program. However, this issue is prominent in EHDs with peripherals. So, we would discuss this in the next subsection.

Summary:

- Memory consistency issues occur because of the WAR dependency between the load and store instructions accessing the NV memory.
- The WAR dependency can be handled in several ways: (i) inserting a checkpoint between the WAR instructions, (ii) undo logging, (iii) redo logging, and (iv) checkpointing the NV updates.

3.3.2. Peripheral-Specific Correctness Issues



(a)

```
a[0] = sense_temp()
sleep(10ms)
a[1] = sense_temp()
sleep(10ms)
a[2] = sense_temp()
sleep(10ms)
avg=a[0]+a[1]+a[2]/3
if (avg < avg_thresh)
     ok = 1
else{
     send_alarm=1
     AES_IN = avg
     AES_HW()
     while(!AES_DONE)
     val = AES_OUT
     send(val)
}
```

(b)

```
a[0] = sense_temp()
sleep(10ms)
a[1] = sense_temp()
sleep(10ms)
a[2] = sense_temp()
sleep(10ms)
avg=a[0]+a[1]+a[2]/3
if (avg < avg_thresh)
     ok = 1
else{
     send_alarm=1
     AES_IN = avg
     AES_HW()
     while(!AES_DONE)
```

```
while(!AES_DONE)
while(!AES_DONE)
while(!AES_DONE)
```

(c)

```
a[0] = sense_temp()
sleep(10ms)
a[1] = sense_temp()
```

```
sleep(10ms)
a[2] = sense_temp()
sleep(10ms)
avg=a[0]+a[1]+a[2]/3
if (avg < avg_thresh)
     ok = 1
else{
     send_alarm=1
     AES_IN = avg
     AES_HW()
     while(!AES_DONE)
     val = AES_OUT
     send(val)
}
```

(d)

```
a[0] = sense_temp()
sleep(10ms)
a[1] = sense_temp()
sleep(10ms)
a[2] = sense_temp()
sleep(10ms)
avg=a[0]+a[1]+a[2]/3
if (avg < avg_thresh)
     ok = 1
```

```
a[0] = sense_temp()
sleep(10ms)
a[1] = sense_temp()
sleep(10ms)
a[2] = sense_temp()
sleep(10ms)
avg=a[0]+a[1]+a[2]/3
if (avg < avg_thresh)
     ok = 1
else{
     send_alarm=1
     AES_IN = avg
     AES_HW()
     while(!AES_DONE)
     val = AES_OUT
     send(val)
}
```

Figure 15: Peripheral-specific correctness issues (a) Temperature sensor code without any failure, (b) Non-termination issues (adapted from [52]), (c) Data timeliness issues, and (d) Input dependent issues.

The above correctness issues were for a general EHD (i.e., with or without peripherals). Now, we discuss issues that can arise in a peripheral-equipped EHD. Figure 15(a) shows an example code for a temperature sensor-equipped EHD. The device takes three readings at an interval of $10ms$, and their average (avg) is compared to a threshold value (avg_thresh). If the average is less than the threshold, the *ok* variable is set; else, the *send_alarm* variable is set, and then the *avg* is encrypted and sent to a base station. The encryption is performed using a hardware AES encryption module. Thus, the device is equipped with two peripherals - a hardware AES encryption module and a temperature sensor.

This program can fail anytime in an intermittent environment. We consider three failure scenarios (indicated by Ⓐ, Ⓑ, and Ⓒ in Figure 15(a)) and discuss the corresponding issues.

Non-termination Issues: This issue can arise due to the AES encryption module [52]. Assume that the hardware module has an input buffer (AES_IN), an output buffer (AES_OUT), and a flag (AES_DONE) stored in NVM. The EHD's software populates AES_IN with the *avg* sensed temperature, runs the AES_HW(), waits for the hardware encryption to complete (i.e., AES_DONE), and then collects the output from AES_OUT and stores it in *val*.

Consider a scenario (Figure 15(b)), where the device checkpoints while busy waiting for the AES to finish, and it fails during this busy waiting. The device restarts and continues to busy wait. However, the *while* loop will never terminate as the H/W module lost its configuration and never sets the AES_DONE flag. One solution is to disable the checkpoints during peripherals' execution and maintain an undo log to roll back in case of a failure. Since this approach disables checkpoints, it will not work when the peripheral operation's energy requirement is more than that supported by the device's energy storage unit. To handle this, we can statically estimate the operation's energy requirement and breaks the complete operation into multiple smaller functions [52].

An alternative solution is to save the peripheral's state, which can be done in nonvolatile IO wherein the nonvolatile ferroelectric flip-flops are attached to the volatile I/O modules [82]. This approach, however, has hardware overheads due to the NV flip-flops. This overhead can be eliminated by checkpointing the state of the peripheral along with the microcontroller's state.

Another approach is to maintain a redo log of the instructions issued to the peripherals [83, 84, 85]. The log can be maintained either in SRAM (which is checkpointed to NVM) or in NVM. Upon a failure, the log can be replayed to retrieve the peripheral's state. An issue with this approach is that it does not support asynchronous peripheral operations as the microcontroller is unaware of the ongoing peripheral operations. Karma [84] provides a solution to this by modeling the peripheral's behavior as a state machine. Each peripheral is characterized by its state that is updated when an instruction corresponding to it is issued. The state of the peripheral is stored along with the log.

Atomicity Issues: Some applications require to execute a part of the code atomically. E.g., an image sensor should sense the complete image. If it powers off in-between, it might not resume at the same point as the target image may have changed. HomeRun [46] proposes to encapsulate such a code segment in an *atomic section* that executes as an uninterruptible unit.

Data Timeliness Issues: Figure 15(c) shows a program that executes incorrectly due to time. The program senses the temperature and takes a checkpoint before the energy gets depleted. The device restarts after an arbitrarily long time and continues the execution by sensing the third value. Due to the long delay, previously sensed values might no longer be valid as the temperature might have changed, resulting in an incorrect average temperature.

The temporal aspect of the data can be maintained by explicitly propagating the wall-clock time in the program and binding the data to the time when it was collected. In this direction, Mayfly [86] proposes a graph-based programming language developed on the top of Embedded-C, which allows specifying timing constraints on sensor data. It also provides a runtime scheduler that maintains time across the power failures using the Remanence Timekeeper [87] – an external capacitor or real-time-clock (RTC) with its dedicated energy store. The RTC is used to decide if a particular datum should be considered or discarded based on the time elapsed. TICS [88] further extends this approach to support legacy software with pointers and recursion that were not allowed in task-based programs supported by Mayfly [86].

Input Dependent Issues: In general, a program's output depends upon its inputs, i.e., different input values might lead to different control flows. Re-executing a program after a failure might result in a path different from the program's failure-free execution. Figure 15(d) shows such a scenario in which the *avg* value in the program's first execution is less than the threshold, thus setting the '*ok*' variable. However, upon re-execution, the code resumes from the checkpointed location and senses again. The new sensed value might lead to an *avg* value greater than the threshold, resulting in setting the '*send_alarm*' variable. Thus, at the end of the program, both the '*ok*' and '*send_alarm*' variables are set, which is never possible in a continuous execution.

A solution is to perform static taint analysis on the program to determine the potential input-dependent issues [89]. A dynamic taint tracking is then performed to detect the issues at runtime.

Summary: Issues in peripheral equipped EHDs arise primarily in two scenarios: (i) The state of the peripheral is inaccessible, (ii) Peripherals such as sensors are unable to capture the entire data or the sensed data changes after a reboot, thus, changing the previously followed control flow or invalidating the previously sensed data.

3.3.3. Verifying Correctness

We now briefly discuss ways to test a program's correctness. Ekho[90] proposes a tool to record the ambient energy traces, which can be later replayed to test the program's behavior. Colin et al. [91] propose EDB – a H/W-S/W debugger that performs runtime monitoring of the program on a particular target platform without affecting the device's energy level. Like Ekho, Eriksson et al. [92] propose a testbed for multiple EHDs to record and replay the energy traces. CleanCut [53] presents a tool to test a program's termination based on EHD's energy capacity.

These approaches are not generic and require either the energy traces, the device's information, or its energy storage unit's capacity. A more beneficial approach is to prove the program's correctness using a theoretical model. Dahiya et al. [93] present a model for intermittence that exhaustively captures all the possible failure scenarios. They also provide a bisimulation-based tool to verify the program's correctness automatically.

However, they do not consider the issues due to peripherals. Berthou et al. [94] present an axiomatic model for systems that checkpoint based on the device's voltage. Their model ensures the entire system's correctness, including the peripherals. On similar lines, Surbatovitch et al. [95] formalize intermittent systems' correctness properties and provide invariants to prove correctness. They also handle input-dependent issues due to peripherals.

4. STATIC INSTRUMENTATION

As discussed in §3.2.2, most of the checkpointing approaches comprise two steps: (i) static instrumentation (offline code annotation) and (ii) runtime checkpointing. Static instrumentation is discussed in this section, while the next section describes the runtime checkpointing.

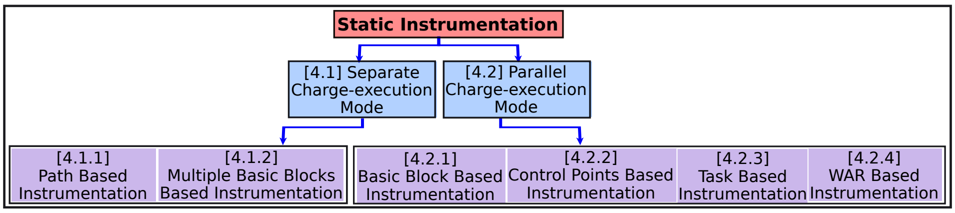

Static Instrumentation
[4.1] Separate Charge-execution Mode
[4.2] Parallel Charge-execution Mode
[4.1.1] Path Based Instrumentation
[4.1.2] Multiple Basic Blocks Based Instrumentation
[4.2.1] Basic Block Based Instrumentation
[4.2.2] Control Points Based Instrumentation
[4.2.3] Task Based Instrumentation
[4.2.4] WAR Based Instrumentation

Figure 16: Overview of static instrumentation techniques

Static instrumentation is primarily done by analyzing the program's control-flow graph (CFG). An EHD can execute in two modes (§2.2.2): (i) 'separate charge-execution mode' and (ii) 'parallel charge-execution mode.' In the former mode, the device loops in fully charging and discharging phases. In contrast, in the latter mode, the device can start charging without fully discharging and vice-versa. This mode can take advantage of the ambient energy's variable nature. If much ambient energy becomes available during execution, it's wise to start charging the device. Similarly, when the available energy becomes extremely low, we can execute the program with the partially charged device. So, in the former mode, we can statically determine the device's energy at different program locations, while in the latter mode, it is not possible as the device can randomly charge/discharge. We classify the various instrumentation schemes based on these two modes (Figure 16).

4.1. Separate Charge-execution Mode

4.1.1. Path Based Instrumentation

This is a straightforward approach, where the program's control-flow graph (CFG) is used to determine all possible execution paths. Given a target EHD and a program, we compute each instruction's energy requirement, considering that the device is executed at its maximum voltage and frequency. We also compute the memory usage and the energy requirement for checkpointing after each instruction. Next, all the possible execution paths are analyzed. We assume that the device is initially fully charged. During the path analysis, we find the instruction after which a checkpoint can be safely inserted, i.e., the instructions and the checkpoint can be executed before the device fails. This process is repeated until all possible execution paths in the program are covered.

Instructions' Energy Requirement: An instruction's energy requirement depends upon the device's memory system. For systems with multiple memory levels (e.g., with caches), an instruction's energy would depend on its hit/miss probability at each level. Thus, the program under consideration is profiled to determine the expected number of cache hits/misses and the average energy required to execute a memory instruction [75, 96]. Rather than simply profiling the program, we can also change the cache replacement policy to maximize the hit probability [74].

Instead of repeatedly computing the number of instructions that can be safely executed based on the device's available energy, we can insert checkpoints periodically after the 'checkpointing distance' (CPD) (Please see §3.2.1 for the definition of CPD). Mirhoseini et al. [97, 98] use this approach in ASICs. They consider the finite-state machine (FSM) generated during the high-level synthesis. The FSM states are equivalent to the instructions in a software program. The execution of FSM is input-dependent, and it results in a sequence of states. For a particular execution sequence, the checkpoint locations are determined based on the CPD. However, instead of trivially inserting the checkpoints at these locations, an optimization problem is formulated that minimizes the total energy to finish the execution, subject to the CPD. Please note that this approach applies to a particular state execution sequence. If an FSM contains a loop, then depending upon the input, the loop can be executed any number of times, resulting in different execution sequences. Authors propose to handle such cases by placing a checkpoint at the loop's back-edge, thus making the execution independent of the number of loop iterations.

Path based instrumentation guarantees forward progress by analyzing all the paths. However, the number of paths increases exponentially with an increase in the program's complexity. Furthermore, analyzing independent paths might lead to unnecessary instrumentation. Figure 17(a) shows a scenario where multiple *switch statement* paths result in instrumenting consecutive instructions.

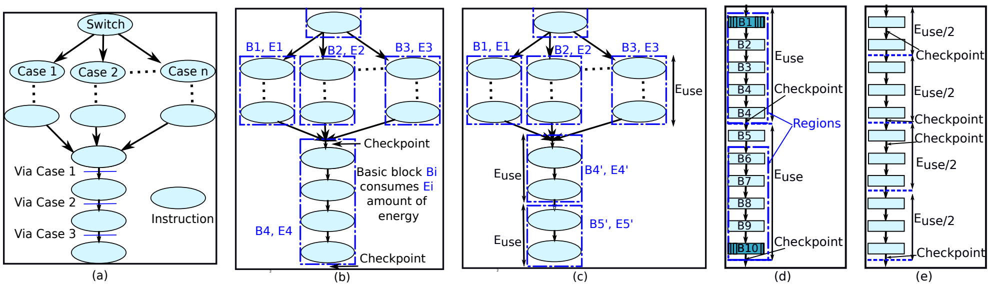


Figure 17: (a) Path based instrumentation, (b) Multiple Basic blocks based instrumentation, (c) Splitting a basic block if its energy requirement is more than the device's available energy, (d) Encapsulating multiple basic blocks as sub-graphs, and (e) Choosing optimal checkpointing locations within a sub-graph.

4.1.2. Multiple Basic Blocks Based Instrumentation

A better solution is to analyze the CFG at the granularity of *basic blocks*. Since we do not know the path that the execution would follow, we use the worst-case value (i.e., the maximum energy required) for a checkpoint (and restore) at any point during the program's execution. Given a fully charged device, we compute the useful energy (E_{use}: used to execute the program) by subtracting the checkpoint and restore energies. We then analyze the CFG and insert a checkpoint after that basic block where the worst-case energy to execute the next basic block is less than the device's available energy. E.g., in Figure 17(b), if the energy consumption of the individual blocks ($B1$, $B2$, and $B3$) is less than E_{use}, the instrumentation is done depending upon the energy required by $B4$. If $max(E1, E2, E3) + E4 \leq E_{use}$, the checkpoint is inserted after $B4$; else, it is added before $B4$.

In this example, we assumed that the energy of all the individual blocks is less than E_{use}. However, if any basic block requires more energy, we need to split the block (Figure 17(c) - block $B4$ is split into $B4'$ and $B5'$). Thus, depending upon E_{use} and blocks' energy requirements, two consecutive checkpoints can have any number of blocks in between. This set of basic blocks forms a *sub-graph*, and a checkpoint is inserted at the end of every sub-graph. Figure 17(d) considers an extremely simple CFG and inserts the checkpoints after blocks $B4$ and $B10$ (last block of their sub-graphs).

The above approach is solely based on E_{use} criteria and inserts checkpoints at the end of every sub-graph. We can improve this approach based on the fact that checkpointing energy is directly proportional to the program state's size. Since a CFG is not generally linear as shown in Figure 17(d) but has a complex structure due to branches and functions, the memory consumption within a sub-graph varies. We can use memory consumption as another criterion to improve energy efficiency. Instead of inserting a checkpoint at the end of a sub-graph, we analyze the instantaneous memory consumption after each basic block (within a sub-graph) and insert a checkpoint where the memory consumption is minimum, thus, reducing the checkpoint size. An issue with this improvement is that we can have a situation where the instantaneous memory is minimum in the beginning for one subgraph, while at the end for the next subgraph (e.g., after blocks B1 and B10 in Figure 17(d)). This would make the maximum distance between two consecutive checkpoints equal to $2{*}E_{use}$, violating our E_{use} criteria. Thus, we reduce the sub-graphs' size to $E_{use}/2$, so that the two consecutive checkpoints would be E_{use} energy apart in the worst-case (Figure 17(e)) [99]. Shoemaker et al. [100] apply this idea of minimum instantaneous memory for programs with dynamic memory allocation and propose to place the checkpoints immediately after free() instead of malloc().

Rather than trivially creating sub-graphs based on basic blocks, we can create them based on the program's semantics [78]. Particularly, the sub-graphs are created such that each sub-graph is reasonably independent of others (i.e., has very few connecting edges to other sub-graphs) and implements a distinct piece of logic. E.g., one sub-graph can correspond to a nested loop, while the other can be a set of nested function calls.

The above approaches compute the energy requirement of each instruction (or basic block or region), implicitly assuming that the EHD runs at constant voltage and frequency. However, as the capacitor discharges during execution, its voltage falls, affecting the clock speed and power consumption. Thus, depending upon the EHD's instantaneous voltage, the energy consumed by the instructions differs. So, an energy model considering the impact of varying voltage should be considered while computing the instructions' energy requirement. In this direction, Ahmed et al. [101] propose EPIC – an automated tool that provides accurate compile-time energy information.

4.2. Parallel Charge-execute Mode

As described in §2.2.2, in this mode, a device can start charging without discharging completely or vice-versa depending upon the ambient energy. If the ambient energy becomes very low while charging, the device begins to discharge. Similarly, if much ambient energy becomes available, it will charge the capacitor and power the load simultaneously. Due to this incomplete charging-discharging, the device's available energy fluctuates at runtime instead of following a particular pattern as in the 'separate charge-execution mode'. Hence, it is not feasible to estimate the device's available energy statically and perform instrumentation. Therefore, a separate set of techniques is needed to cater to devices in this space, which is agnostic to the device's available energy.

4.2.1. Basic Block Based Instrumentation

The simplest approach is to insert checkpoints after each basic block. This approach, however, pessimistically inserts an excessively large number of checkpoints. Maeng et al. [64] follow this approach; however, the authors propose to adopt a runtime mechanism to reduce the number of actual checkpoints (discussed in §5). Please note that unlike the separate charge-execution mode, we cannot estimate the energy available in the device and hence not merge multiple basic blocks.

4.2.2. Control Points Based Instrumentation

Another approach is to insert checkpoints at the *control points* in a program. Mementos [7] inserts checkpoints after each function call or at the loop's back-edge. If an iteration's energy requirement is more than the device's capacity, the loop should be split. Splitting reduces the number of instructions to be executed before taking a checkpoint. A smaller number of instructions also reduce the program state, and thus the checkpoint size.

4.2.3. Task Based Instrumentation

Inserting checkpoints based on the basic blocks [64] or control points [7] results in an excessive number of checkpoints, increasing the program's total execution overhead. This issue can be avoided by rewriting the program such that a long program is divided into a sequence of *semantically meaningful* short tasks connected by checkpoints [51, 54, 55]. These tasks execute atomically, and each task performs some meaningful computation. E.g., a monolithic program that performs sensing and then computes some statistics on the sensor data can be divided into two tasks: sampling() and compute_statistics(). These tasks are created solely based on the logic and without considering the device's available energy. However, to guarantee termination, the tasks should be created such that any task's maximum energy consumption is less than the device's storage unit's capacity. A significant issue with this approach is that it is difficult to make even a small change to the program as it can make the previously created tasks unusable. Furthermore, the entire onus of efficient task creation is on the programmer.

4.2.4. WAR Based Instrumentation

As discussed in §3.3.1, we can also have WAR based instrumentation, where a checkpoint is inserted between the load and store instructions accessing the same memory location. Several proposals [64, 40] use this approach.

Summary: In separate charge-execution mode, the energy of the device can be accurately estimated statically. Thus, we can optimally instrument a program. However, in parallel-charge execution mode, accurate estimation is not possible. So, in this case, the instrumentation is generally done pessimistically, and a runtime mechanism is used to reduce the number of checkpoints.

5. RUNTIME CHECKPOINTING

Having discussed the program annotation techniques, let us now discuss the strategies to efficiently take checkpoints at these annotated locations. These strategies are summarized in Figure 18.

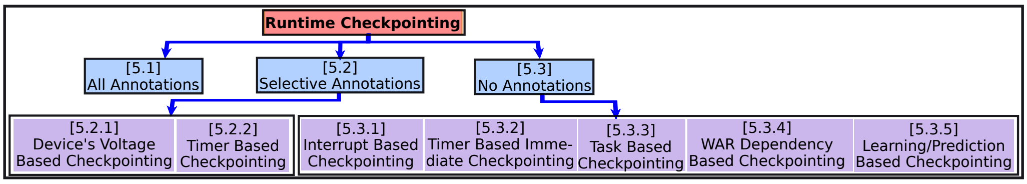

Runtime Checkpointing
[5.1] All Annotations
[5.2] Selective Annotations
[5.3] No Annotations
[5.2.1] Device's Voltage Based Checkpointing
[5.2.2] Timer Based Checkpointing
[5.3.1] Interrupt Based Checkpointing
[5.3.2] Timer Based Immediate Checkpointing
[5.3.3] Task Based Checkpointing
[5.3.4] WAR Dependency Based Checkpointing
[5.3.5] Learning/Prediction Based Checkpointing

Figure 18: Overview of the existing runtime checkpointing approaches

5.1. All Annotations

The most straightforward approach, as followed by Dino [51], is to take checkpoints at all the annotated locations. However, most of the annotation techniques [64, 7] conservatively annotate the program with a large number of checkpoints to avoid non-termination. Therefore, this approach of checkpointing at all the annotations would be quite expensive.

5.2. Selective Annotations

As discussed above, it is not wise to take checkpoints at all the annotations. So, different approaches have been proposed to decide if a checkpoint should be taken at a particular instrumented location.

5.2.1. Device's Voltage Based Checkpointing

The device's instantaneous energy is checked at each annotated location. The device's voltage is used as a measure of the device's energy since it can be easily measured using an analog-to-digital converter (ADC). Based on this voltage and the energy requirement of the program's instructions, we determine if we can successfully reach the next instrumented location and take a checkpoint there. A checkpoint is taken only if it is not possible to reach the next instrumented location. Li et al. [96] and Xie et al. [70] follow this approach.

This approach has two drawbacks. Firstly, for every instrumented program, we need to have a lookup table of the energy requirement of the instructions between any two consecutive annotations, resulting in memory overheads. A solution is to use a *threshold* voltage. If the device's voltage is below the threshold value, only then a checkpoint is taken. The threshold is determined based on the program's runtime behavior after repeatedly executing the program with different ambient energy traces [7]. Secondly, the ADC consumes energy and can result in a 40% loss of the system's energy [62]. There are two ways to reduce the overhead of ADC: (i) use an advanced ADC that has a much lower overhead than a typical ADC [49], (ii) reduce the ADC's use by employing a timer. Mementos [7] uses the latter approach. It sets the timer to an empirically determined value, and the voltage at an annotated location is checked only if the timer expires.

5.2.2. Timer Based Checkpointing

Instead of using both the timer and an ADC, a timer alone can also determine when to take a checkpoint. However, the timer needs to be adaptive to capture the ambient energy's variations. The timer is initially set to a random expiration time. When the timer expires during the program's execution, the checkpoint is taken at the next annotated location. If the device fails before the timer expires, a reduced timer value is used in the next execution. In contrast, if the timer does not expire until the subsequent annotation, the timer's value is increased. Thus, the timer value is tuned according to the available ambient energy. The time value can be changed using any heuristic, e.g., Chinchilla [64] either doubles or halves the timer value. Please note that this approach requires that the program be over-provisioned with checkpoint annotations, as distant annotations would increase program failure chances.

Please note that this approach differs from the previous 'device's voltage based checkpointing. Here, a timer expiration means that we should take a checkpoint, while in the previous approach, we check the device's voltage when the timer expires.

5.3. No Annotations

Though the annotations added during static instrumentation guide the process of runtime checkpointing, static instrumentation has its overheads. The program's CFG needs to be analyzed, and then the checkpoints are inserted either manually or by the compiler. The annotations increase the code size. Furthermore, static annotation might insert an excessive number of checkpoints. So, several researchers have proposed to checkpoint directly at runtime without any prior instrumentation. This provides flexibility as we are not constrained by the annotated locations and can dynamically checkpoint, adapting to the ambient profile. Now, we present various such approaches.

5.3.1. Interrupt Based Checkpointing

Followed by several works [61, 49, 76, 102, 36, 77], this approach monitors the device's runtime voltage similar to 'device's voltage based checkpointing,'. However, this approach performs interrupt-based monitoring, as there are no annotations. Whenever the device's voltage goes below a threshold voltage (V_H), an interrupt is generated, and a checkpoint is taken. After checkpointing, the device *hibernates*, i.e., it halts the execution and enters a low power mode. The threshold voltage, V_H, is chosen such that at this voltage, the device has energy only to take a checkpoint, i.e., there is no more energy left in the device to make any progress. This is also called just-in-time (JIT) checkpointing, and unlike various annotation based approaches where the device continues to execute after a checkpoint without hibernating. In this, the device takes a checkpoint just before its energy is about to deplete. The device wakes up and restores its state only when the voltage rises beyond a restore threshold voltage (V_R, where $V_R > V_H$). The threshold values can either be *fixed* [61, 49] or *adaptive* [36, 72]. Adaptive thresholds are computed based on the device's capacitance, its power consumption, and the dynamics of the harvested energy.

The thresholds are generally decided conservatively considering the worst-case scenario, i.e., the amount of energy required to backup the entire SRAM [51, 49, 36]. However, certain approaches follow copy-if-change checkpointing (see §3.2.1), wherein only the modified data is checkpointed. Thus, the checkpoint's size and its energy requirement would vary at different program locations and would be less than the worst-case energy. Thus, it is possible that when the interrupt occurs, the device has sufficient energy to execute some of the future instructions before checkpointing. Song et al. [103] use this idea to find and execute the additional instructions before checkpointing.

It is possible that after entering into hibernation, sufficient energy gets harvested (in parallel charge-execution mode), without failing and the device's voltage rises beyond V_R without any device failure. In such a scenario, we cannot change the thresholds as the device did not fail. Thus, based on the previously set thresholds, an interrupt will be generated whenever $V < V_H$, leading to unnecessary checkpointing. This issue can be avoided by tracking the device's state after it enters hibernation [102]. While hibernating, the device can either fail or safely resume execution. This state information can be used in the subsequent execution to decide if a checkpoint should be taken or not before hibernating. Particularly, if no failure happened, the checkpoint can be avoided hoping sufficient energy would be harvested during hibernation.

Another approach for reducing the number of checkpoints is to have an additional threshold (V_S), such that $V_S > V_H$ [48]. When the device's voltage (V) is between V_S and V_H, the device goes into sleep mode without taking any checkpoint, thus consuming less power. The idea is that when the device notices its dropping voltage, it decides to conserve energy rather than spending on the program's execution. The energy between the voltages V_H and V_S, earlier used in the execution, is now used to retain the program's state. A checkpoint is taken when $V < V_H$.

The efficiency of these approaches relies on the chosen thresholds. If the thresholds are high, they would be reached soon, and the device would stop executing despite the energy being available.

Rather than checkpointing/sleeping when the voltage falls below a threshold, Fan et al. [104] propose to take the checkpointing decision dynamically by using a Q-table that is generated online based on the capacitor's instantaneous energy and the amount of data to be checkpointed.

5.3.2. Timer Based Immediate Checkpointing

As in 'timer based checkpointing (selective annotations),' this approach removes the need for voltage tracking; however, it uses an un-instrumented program and checkpoints whenever the timer expires. The timer value is adaptive [47]. The value is halved if the device fails before the timer expires. More than one checkpoint before the device's failure indicates that sufficient energy is being harvested, and the timer value is doubled. Instead of simply halving/doubling the timer value, iCheck [105] devises a mathematical model to compute the timer value based on the ambient source's power traces and program's overheads of checkpointing, recovery, and execution.

5.3.3. Task Based Checkpointing

This is similar to the 'task based annotation' in static instrumentation. However, in contrast to static splitting, the tasks are split dynamically based on the harvested energy. QuarkOS [106] and Coala [107] adopt this approach where they adjust the task's size dynamically by merging and splitting the tasks, thus, adapting to the energy harvesting conditions. QuarkOS [106] presents an OS that automatically divides every communication, sensing, and computing task into tiny fragments. E.g., in a communication task, the data to be communicated is fragmented into data bits, while in an image sensing task, the individual pixels form the fragments. Sleep is introduced between these fragments to allow the device to recharge. Thus, systems with a tiny harvesting rate and a small energy storage unit can run any program. Coala [107] determines the number of tasks based on the recent history of tasks, i.e., depending upon the number of tasks executed successfully in the previous charge-discharge cycle.

5.3.4. WAR Dependency Based Checkpointing

Various systems, such as Clank [62], track the WAR dependencies at runtime and take a checkpoint whenever a WAR dependency is encountered. This approach is ineffective if the WAR dependencies in a program are so far apart that the device's energy depletes before reaching the instructions involved in WAR dependency.

5.3.5. Learning/Prediction Based Checkpointing

Another way is to learn the checkpointing decisions offline for a particular ambient profile and use them online during the program's execution. Ghodsi et al. [27] adopt this approach, where they model the checkpointing problem as a Markov decision process. Given a program and an ambient profile, they use Q-learning (a standard reinforcement learning algorithm) to compute the optimal checkpointing policy. The state used in the algorithm is a tuple of the – program counter, program counter at the time of the last checkpoint, and device's instantaneous energy. At each state, two actions are possible: (i) take a checkpoint or (ii) execute. There is a time cost involved for each state-action pair. For the checkpoint action, the cost is equivalent to the checkpointing cost, while for the execute action, the cost depends upon if a failure occurred or not. In case of a failure, the incurred cost is equivalent to the time spent in the lost progress. These costs are learned offline and stored in memory. As the program executes, the action with minimum cost is chosen for a particular state. This approach has two issues: (i) the space overhead for storing the learned decisions, (ii) the actual ambient power profile might vary from the learnt profile; thus, the checkpoint decisions need to be adapted to these variations. Another similar approach is to use an optimization formulation of the checkpointing problem. Singla et al. [25] formulated a Quadratically Constrained Linear Program (QCLP) that provides the optimal solution to the checkpointing problem. However, it takes an exponential time to find a solution, and the solution is for a particular ambient profile. So, the authors proposed another approach that predicts the ambient energy and takes checkpoint decisions accordingly. If a large amount of incoming energy is predicted, the checkpoint is avoided, else it is taken. These approaches do not require an annotated program but take checkpoint decisions periodically after a fixed number of instructions.

Summary: Among the three broad categories of runtime checkpointing, ‘all annotations’ is the most straightforward approach, while ‘no annotations’ is the most flexible approach as we are not constrained by the annotated locations. Furthermore, the latter approach removes the overheads of static annotation, such as manual (or automatic) analysis of the program’s CFG and large code size.

6. EXPERIMENTAL ANALYSIS

The aim of this section is to evaluate the major checkpointing approaches on a common baseline architecture. We shall compare a variety of runtime parameters notably the time of execution, energy consumed, number of checkpoints, and time spent in recovery. We shall also assess the impact of different ambient energy sources, battery/capacitor capacities, and the device’s SRAM sizes. At the end of this section, we present a set of concrete recommendations. These recommendations would help a professional system designer decide which approaches are best under different scenarios.

Figure 19 presents a high-level flow of the experimental section. We initially discuss the experimental setup, wherein we discuss our evaluation platform, the benchmarks considered for evaluation, the analyzed checkpointing approaches, and different ambient power profiles considered for evaluation. Next, we compare the different checkpointing approaches when they are executed with an infinite power supply. Subsequently, we compare the approaches under different system configurations. This study considers that the device has 2KB of SRAM. We further extend this analysis to incorporate the behavior of the approaches with increasing SRAM sizes. Particularly, we compared the performance of approaches with four different SRAM sizes – 1KB, 2KB, 4KB, and 8KB. These sizes are typically found in the existing embedded devices used in energy harvesting systems.

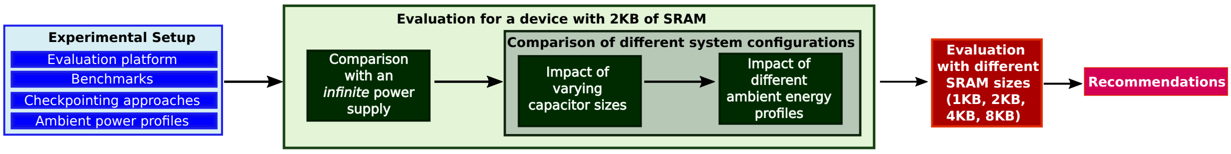

Experimental Setup
Evaluation platform
Benchmarks
Checkpointing approaches
Ambient power profiles
Evaluation for a device with 2KB of SRAM
Comparison with an *infinite* power supply
Comparison of different system configurations
Impact of varying capacitor sizes
Impact of different ambient energy profiles
Evaluation with different SRAM sizes (1KB, 2KB, 4KB, 8KB)
Recommendations

Figure 19: Overview of the experimental section

6.1. Experimental Setup

Evaluation platform: For our experiments, we use the standard approach wherein we accurately simulate the microcontroller, considering the processor configuration and power values according to the device’s datasheet. This approach has been followed by existing works [39] for doing research since these EHD devices are not always available easily, and their hardware implementation cannot be modified. Furthermore, when a new hardware is proposed, it is practically infeasible to modify, fabricate, and characterize it. Therefore, simulation is a valid approach and is widely accepted in the whole embedded systems and computer architecture communities.

In line with this, we use the popular cycle-accurate architectural simulator - Tejas [108] that has been rigorously validated against native hardware. We modeled an in-order processor along the lines of the TI MSP430 microcontroller, MSP430FR5969 in particular, which has been used in the state-of-the-art works [51, 54, 25]. The modeled system is a 16-bit, 5-stage in-order processor with a constant frequency of 16MHz. It has a 32-byte set-associative cache, 2KB of SRAM, and 64KB of FRAM (NVM). The power and timing values of the processor components have been computed using McPAT, while the SRAM parameters have been generated using Cacti 6.0.

Benchmarks considered: We consider three widely used benchmarks from diverse application areas. The benchmarks are (i) Activity Recognition (AR): A machine learning benchmark that learns and classifies the accelerometer samples, (ii) Cuckoo Filter (CF): Similar to Bloom filters, this benchmark tests for set membership, and (iii) Data encryption benchmark: It uses the RSA algorithm to encrypt the data.

Approaches analyzed: According to our proposed taxonomy (§5), we have implemented various approaches and have tried to include at least one approach from each category. Table 3 summarizes the considered approaches along with their *checkpointing classes* as per our taxonomy. The table also mentions the distinguishing features adopted by a particular approach. E.g., Daulby et al. [48] use DMA for checkpointing, Elastin [47] and Ratchet [40] are fully NVM-based systems, unlike others that have both volatile and non-volatile memories. Please note, we consider that each approach checkpoints the entire SRAM and the register file unless explicitly specified by the approach. This consideration is for maintaining fairness across the results. The approaches have been primarily compared based on the total time taken and the total energy consumed to run a benchmark. Furthermore, for each approach, we report the geometric mean of the considered benchmarks (as per standard practice).

Among the mentioned approaches (Table 3), QCLP - Quadratically Constrained Linear Program – provides the optimal solution of the checkpointing problem. Consequently, it is chosen as the baseline for comparison. Please note that the considered QCLP formulation has been optimized for the total time taken for a particular ambient profile. So, other approaches may consume lesser energy than the QCLP formulation; however, the time taken by QCLP is the least.

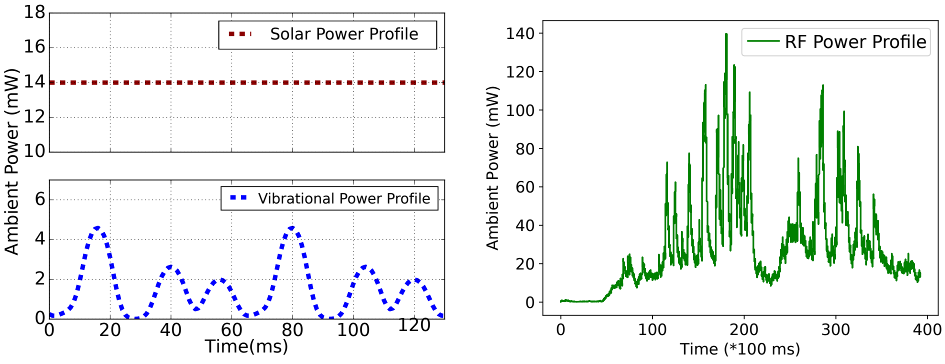
Approach	Checkpointing class [Section No.]	Approach	Checkpointing class [Section No.]
Dino [51]	All Annotations [5.1]	Clank [62]	WAR Dependency Based (NA) [5.3.4]
Mementos [7]	Device's Voltage Based (SA) [5.2.1]	Ratchet^4 [40]	
Chinchilla [64]	Timer Based (SA) [5.2.2]	Chain [54]	Avoiding Checkpoints [6]
Hibernus [49]	Interrupt Based (NA)[5.3.1]	Alpaca [55]	
Daulby^1 [48]		FlexiCheck [25]	Learning/Prediction Based (NA) [5.3.5]
Selective^2 [103]		QCLP^{*5} [25]	
Elastin^3 [47]	Timer Based (NA) [5.3.2]		

Distinguishing features: 1-use DMA for checkpointing, 2-checkpoints only selective data, 3-use copy-on-write and fully NVM, 4-fully NVM, 5-optimized for total time;SA:Selective Annotations; NA: No Annotations; *:Baseline Approach

Table 3: Summary of approaches considered

Ambient power profiles: We have considered three different ambient power profiles having different power densities and different degrees of variation (Figure 20(a,b)). The considered ambient sources include: (i) a solar source that provides a constant power of 14mW, (ii) a vibrational energy source that provides a low power (between $0 - 4mW$) varying sinusoidally at the granularity of ms, and (iii) a representative RF source that offers low power in the beginning (for around 5000ms) and then high-intensity power that fluctuates at the granularity of seconds.

We used popular models that have been used in prior work. The solar power trace has been generated according to [21], while the vibrational power trace has been generated according to [25], which considers a machine tool with several vibrational parts with resonant frequencies around 70-100Hz and $10\,\mathrm{m\,s^{-2}}$ peak acceleration. The RF power trace has been taken from Mementos [7]. The trace was captured by a hardware device equipped with an RF-harvesting analog frontend, and a 10kΩ resistor was used as the device's load. The device was moved within $2\,m$ of an RFID reader, and the voltage across the device's in-built resistor was captured.



(a) (b)

Figure 20: Ambient power profiles **(a)** solar and vibrational power profile (from [25]), **(b)** RF power profile (from [7]).

Along with the various ambient profiles, we also studied the impact of varying capacitor sizes for each approach. We considered three values of capacitances - 10µF, 16µF, and 47µF, which have been used in the existing literature [49, 64, 47]. We charge the device to a predefined level (75% of its capacity, decided empirically) upon every failure and then run the program allowing the device to charge in parallel. We experimented with 50%, 75%, and 100% levels; however, certain checkpointing approaches did not finish with the corresponding amount of energy when the predefined level was set to 50%. The results for 75% and 100% charge levels were similar, but for 100%, the device took a long time to charge; thus, increasing the total execution time. So, we used 75% as the predefined level in our experiments.

6.2. Experimental Results

6.2.1. Comparison with an infinite power supply

Figure 21 compares the approaches when executed with an infinite power supply, i.e., the device's energy never exhausts. The approaches have been divided into various classes as per our taxonomy (§5 - 'Runtime Checkpointing'), as indicated in the figure's legend.

Figure 21(a) shows the total number of instructions executed by each approach (all the approaches run the same high-level program). As seen in the figure, certain checkpointing classes execute considerably more instructions, while the other classes execute almost the same number of instructions. Notably, the classes - 'All Annotations, Selective Annotations, and Avoiding Checkpoints,' execute around $1.5 - 12.7\times$ more instructions than others. Additional instructions are executed because, in these classes, the high-level program has been statically instrumented. The program is either annotated to mark the checkpointing locations or modified to use custom libraries. Therefore, the instrumentation-based checkpointing classes increase the number of executed instructions. To study the impact of executing additional instructions, we analyzed the energy and the time consumed.

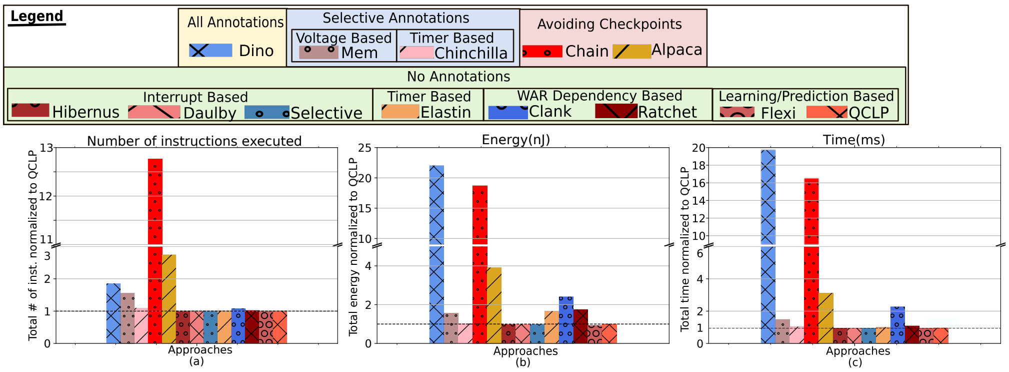


Figure 21: Comparison of the approaches based on the **(a)** total number of instructions executed, **(b)** total energy consumed with an infinite power supply, and **(c)** total time taken with an infinite power supply.

Figures 21(b-c) show that the annotation-based classes have enormous energy and time overheads. They consume around $4 - 22\times$ more energy and take around $3 - 20\times$ more time than the baseline approach (QCLP), while the energy and time overheads of other classes are within $2.2\times$ of the baseline. Thus, in general, the checkpointing approaches that use static instrumentation and as a result execute a large number of additional instructions result in high overheads. Along with this insight, the graphs provide us with two more insights:

(1) The 'All Annotations' class has the largest energy and time overheads, indicating that checkpointing at all the annotation points is an extremely expensive proposition. In contrast, the 'Selective Annotations' class performs much better even though the number of instructions executed in the two classes does not differ significantly. In particular, 'All Annotations' approaches execute only 20% more instructions than the 'Voltage Based Selective Annotations' approaches, but they consume around $15\times$ more energy and take around $13\times$ more time. This is the checkpointing overhead.

(2) Most of the 'No Annotations' based approaches perform amazingly well with an infinite power supply. The exceptions are Elastin (timer based), Ratchet, and Clank (WAR dependency based). These approaches have high overheads despite executing almost the same number of instructions as other approaches in their class (No Annotations).

Large overheads in Ratchet and Elastin are due to their fully NVM-based structure, i.e., they use NVM as the main memory: accessing the NVM is much more expensive than accessing the SRAM. Clank, on the other hand, uses a memory with mixed volatility (NVM+SRAM). Nonetheless, its high overhead is accounted for by the large number of checkpoints, which is proportional to the number of WAR dependencies in the program.

Summary: We conclude the above discussion with the following key points:

- Annotation-based classes generally execute a very large number of instructions, resulting in significant energy and time overheads.
- Checkpointing at all the instrumented locations is overkill.
- Fully NVM-based systems are expensive (60% more energy as compared to the baseline).

6.2.2. Comparison of Different System Configurations

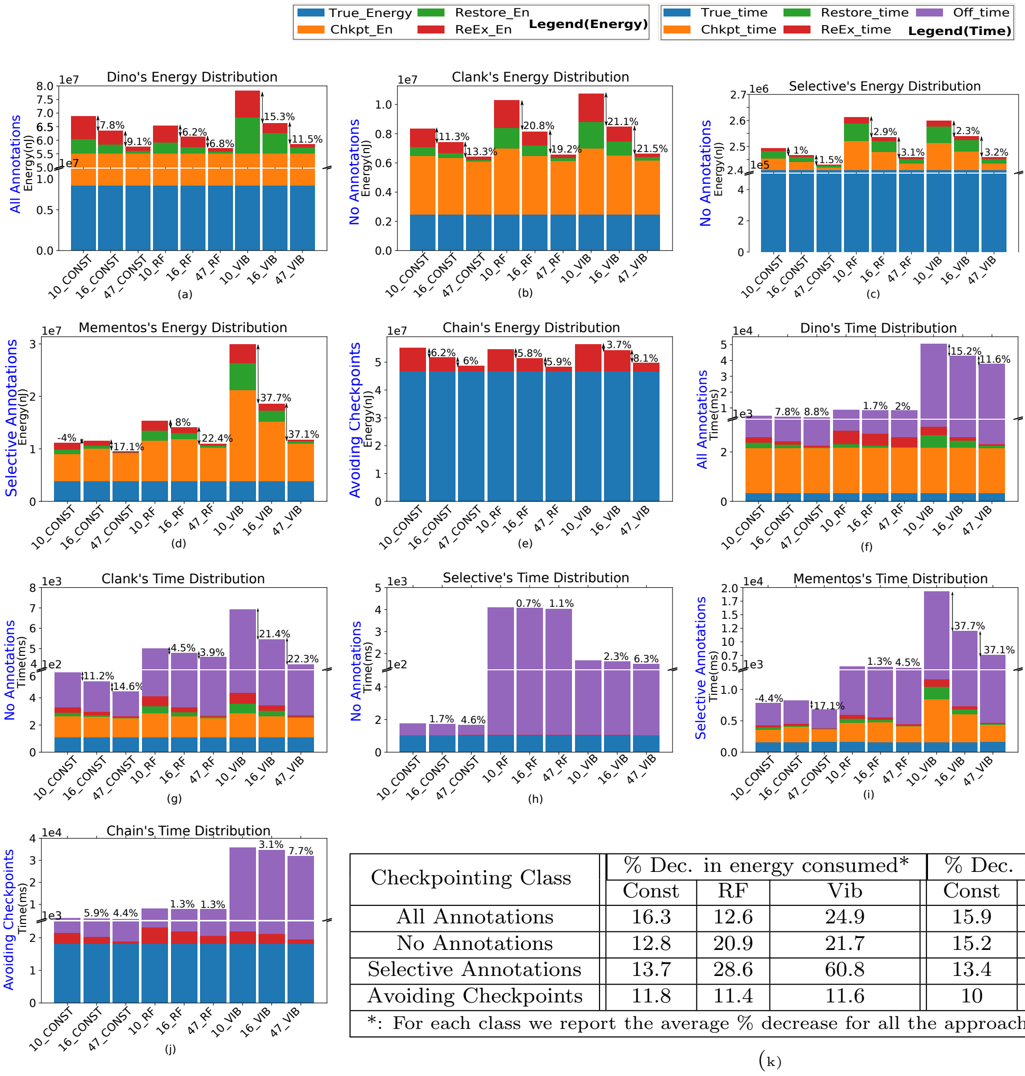
We now compare the approaches when executed under different system configurations. In our experiments, the system configuration comprises two parameters: the device's capacitance (energy storage capacity) and the ambient energy profile. Henceforth, we analyze the approaches' behavior with different combinations of capacitances and ambient sources.

1. Impact of varying capacitor sizes: We initially study the performance (energy and time) of different classes with varying capacitor sizes. Figure 22 presents the results for all checkpointing classes. We have shown results for one or two approaches from each class. The rest of the approaches in each class followed a similar behavior.

Figure 22 presents stacked bar plots showing various components of the total energy consumed and the total time taken for several checkpointing classes. In each graph, the bar labels on the x-axis indicate the capacitance and the ambient source. E.g., 10_CONST means that a 10μF capacitor and a constant energy source are used for the experiment.

A bar in the energy graph is composed of: (i) True_Energy - the energy consumed when the approach is run with an infinite power supply; this energy does not include the checkpoint/restore overheads, (ii) Chkpt_En - the energy consumed in checkpointing, (iii) Restore_En - the energy consumed while performing a restore operation, and (iv) ReEx_En - total energy consumed during the re-execution. Analogous to the energy components, the time graph has similar components. An additional component, *off_time*, indicates the time for which the device was off (or sleeping/hibernating) and charging its capacitor.

We initially discuss the impact of increasing capacitor sizes for one class and then discuss the impact on other classes. We shall generalize the observation based on the impacts on other classes. Figure 22(a) shows that the energy consumed by Dino (All Annotations) decreases (roughly 10-30%) with an increase in the capacitor size (for all the ambient sources). The reason behind the decrease is that with the increased capacitance, the number of failures decreases. Fewer failures result in fewer restores and re-executions. We can see from the figure that the energy spent in re-executions and restores decreases with increasing capacitor sizes. Similar to restore_en and reEx_en, it intuitively appears



Checkpointing Class	% Dec. in energy consumed*			% Dec. in time taken*		
	Const	RF	Vib	Const	RF	Vib
All Annotations	16.3	12.6	24.9	15.9	3.7	25.1
No Annotations	12.8	20.9	21.7	15.2	5.1	23.7
Selective Annotations	13.7	28.6	60.8	13.4	5.8	60.8
Avoiding Checkpoints	11.8	11.4	11.6	10	2.5	10.6

*: For each class we report the average % decrease for all the approaches in that class

(k)

Figure 22: **(a-e)** Energy distribution and **(f-j)** Time distribution for different checkpointing classes with different ambient power profiles and capacitor sizes, **(k)** Relative performance gains for different classes, across different ambient sources when capacitor size was increased from $10uF$ to $47uF$.

that the chkpt_en should also have decreased. But, this did not happen because 'All Annotations' based approaches checkpoint at all the annotated locations, irrespective of the device's energy. Thus, this class is associated with constant checkpointing energy across all the capacitor sizes. A similar pattern is observed for the execution time (Figure 22(f)). Additionally, we notice that the off_time dominates among all the other time components. The off_time primarily dominates because, for energy harvesting devices, the rate of charging is far lower than the rate of discharging due to the sporadic nature of the ambient sources.

This decreasing pattern is a general observation; for each ambient source, an increase in the device's capacitance benefits all the classes. However, the amount of performance gained and energy reduced by the approaches differ across the ambient sources. For each graph, we have annotated the percentage energy (time) decrease with an increase in the capacitor size. E.g., in Figure 22(a), for the constant ambient source, as the device's capacitance is increased from 10μF to 16μF, the energy consumed decreases by 7.8%, and then it reduces by a further 9.1% when the capacitance is equal to 47μF. To quantify the impact of the increase in the capacitor size for each class, we summarize the average performance gains for each class for different ambient sources when the device capacitance is increased from 10μF to 47μF in Figure 22(k) (numbers are relative to each class).

As we can see from the table, the performance gain (both energy and time) is maximum for the vibrational energy source for all the classes (Chain being the only exception). This is because of its periodic nature, which is germane to an EHD scenario. On the other hand, the reduction in the execution time is the least with the RF source across all the approaches. However, this pattern is not followed for the energy consumed. The average % decrease in the energy consumed for the 'All Annotations' and 'Avoiding Checkpoints' classes is the minimum for the RF source, while for the 'No Annotations' and 'Selective Annotations' classes, it is the least for the constant source. One reason for the reduced sensitivity of RF (in terms of time) is because of the dominating off_time component. For energy consumption, the constant and RF energy sources are slightly less sensitive because they have regular phases in terms of power delivery: either very little power or constant power.

Apart from the above insights, we discuss some general observations:
(1) *The 'constant' ambient source is the best among all the sources:* As we can see from Figure 22(a-j), the total energy consumed and time taken by most of the classes is the least with the constant energy source. Furthermore, for most classes, the off_time with the constant ambient source is an order of magnitude lower than other sources. The reason is the steady flow of incoming energy that quickly charges the device, unlike the sources with fluctuating power. Particularly, for a 10μF capacitance, all the classes consume $1 - 1.37\times$ and $1.02 - 2.7\times$ more energy with the RF and vibrational sources, respectively. Similarly, the RF and vibrational sources take $1.7 - 23.2\times$ and $9.3 - 24.6\times$ more time, respectively, compared to the constant source. As the capacitance increases to 47μF, the energy consumption for all the approaches becomes almost the same. However, the time taken by the approaches still varies significantly. This is because of the off_time, as it would take more time to charge a capacitor with a fluctuating ambient source. Furthermore, the off_time would depend upon the degree of fluctuation. Therefore, considering both the energy and time performances, the constant solar source

is the best among all our considered sources. A disclaimer is due: the RF source can have many different avatars. We have considered a typical scenario, which is in line with assumptions made in prior work.

(2) *For some approaches, the off_time is more with the vibrational energy source, while for others, it is more with the RF source:* For some approaches, the off_time is more with the vibrational energy source due to its low intensity (see Figure 20(a,b) for the intensities of energy sources). In contrast, other approaches have a small off_time for the vibrational source but a large off_time for the high-intensity RF source. For explaining this, we consider two approaches from the 'No Annotations' class - Clank and Selective. Figure 22(g) and Figure 22(h) show the time distribution plots for Clank and Selective, respectively. Despite the high-intensity RF source, the Selective approach's poor performance (Figure 22(h)) can be explained based on the RF trace (Figure 20(b)) and the approach's execution time. The RF trace in Figure 20(b) shows that in the beginning, the power available to the benchmark is relatively less (a few mW for around $4000ms$). Due to this low-intensity phase, the Selective approach performed poorly. Though the available power increases subsequently, the benchmark would have already finished by that time (indicated by a total execution time of approximately 4000ms in Figure 22(h)). In contrast, Clank performed well with the RF source due to its large execution time ($> 5000ms$). The benchmark was executing when the high-intensity phase of the RF arrived, resulting in Clank's better overall performance. This experiment indicates that *irrespective of the fluctuations in the ambient power profile, the amount of the total energy available in a particular time interval determines an approach's performance.*

(3) *$16\mu F$ is a reasonable capacitor size for all the ambient sources:* Intuitively, a larger increase in capacitance should result in larger energy and time performance gains than a smaller increase in capacitance. However, for various approaches (with some ambient sources), this expected pattern is not followed. E.g., for Dino with a vibrational source, the energy consumption decreases by 15.3% when the device's capacitance increases from 10μF to 16μF. In comparison, the energy consumption decreases only by 11.5% when the capacitance is increased from 16μF to 47μF (almost a 3× increase in capacitance). Similar numbers are observed for the execution time. Like Dino, Mementos (with a vibrational source), Clank (with an RF source), and Chain (with an RF source) present similar observations. From these observations, we infer that 16μF is better than 10μF. Though 47μF results in maximum performance for all the approaches across all the sources, large capacitances also add to the cost. So, we can conclude that 16μF represents a reasonable trade-off. Moreover, a large capacitor would take extremely long to charge and would not result in a good performance for ambient sources with feeble power delivery.

Summary

- Increase in capacitance from 10μF to 47μF results in increased performance (both time: 10.6% – 60.8% and energy reduction: 11.6% – 60.8%). The relative gains with the vibrational energy source are the best for all the approaches.
- The constant (solar) ambient source offers the best energy and time performance for all the classes.
- Irregular sources such as the RF source are the least sensitive to the capacitor size. In comparison, regular and periodic sources are more sensitive. Among the different classes, the 'Avoiding Checkpoints' class is the least sensitive (within 10%), whereas 'Selective Annotations' is the most sensitive primarily because energy availability at specific time instants is of paramount importance.

2. Impact of different ambient energy profiles:

Next, we show the relative performance of all the checkpointing classes with different ambient energy profiles. Figure 23 presents the energy and time overheads of all the considered approaches normalized to the baseline – QCLP. We have shown results for only one capacitor value (10μF) as the high-level relative patterns with other capacitances were mostly similar. The experimental results and the corresponding observations have been summarized below:

General results:

(1) With all the considered ambient sources, most of the approaches broadly follow the same pattern for both the energy consumed and time taken. This pattern is similar to that witnessed when the approaches were executed with an infinite power supply. The approaches that performed well with an infinite supply also performed relatively better with all kinds of ambient sources.

However, Elastin [47] – a timer-based approach, is an exception that does not follow this pattern. Its relative energy consumption increased around $8 - 15\times$ compared to the relative energy consumption when the approaches were run with an infinite power supply. Similarly, its relative time taken increased by $2.5 - 40\times$. Elastin's poor performance is not due to its timer-based nature but because of its implementation. It uses copy-on-write (COW) in a fully NVM-based system. Whenever an address in a page (the page size is 256 *bytes* as used by the authors in the paper) is accessed, rather than doing an in-place update, it performs an expensive COW operation. This energy-intensive COW operation results in numerous failures and re-executions, thus increasing the overall energy. Due to frequent energy depletion, the off_time is also very high.

(2) Among all the approaches, the 'interrupt based (NA)' and the 'learning/prediction based (NA)' approaches are the best with all the ambient sources. The time overhead of these approaches is within 70% of the baseline (QCLP). The energy consumption of these approaches is around 10 – 40% lower than the baseline (QCLP). Please note that the reason behind lower energy consumption as compared to QCLP is that the latter has been optimized for minimizing the total time rather than the total energy consumed.

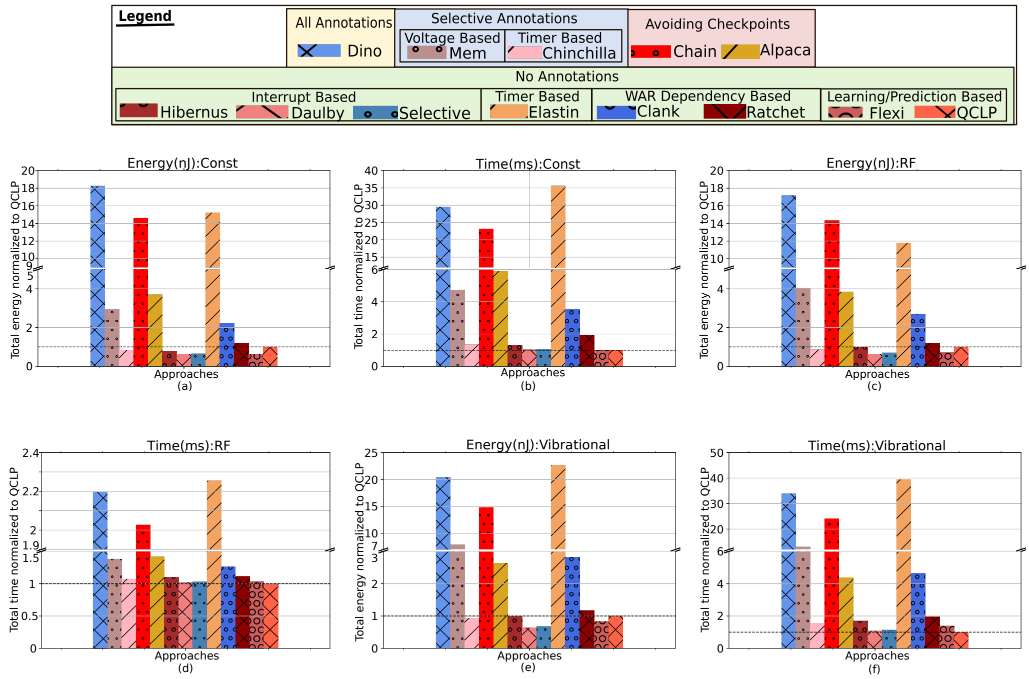


Figure 23: Energy consumed and time taken with **(a-b)** constant power supply, **(c-d)** RF power supply, and **(e-f)** vibrational power supply.

These classes are the most efficient, and we term them as '*superior classes.*' Among the approaches in the *superior* classes, Daulby and Selective consume the least energy as the former uses DMA for taking checkpoints, while the latter checkpoints only the modified data. In contrast to Selective, other approaches checkpoint the entire SRAM. Interestingly, we observe that learning/prediction based approaches perform reasonably well despite checkpointing the full SRAM. The better performance is because they can act according to the energy available in the future and take fewer checkpoints. Combining these observations and results, a learning/prediction based approach that selectively checkpoints using DMA would be the best among all the approaches.

In contrast to the *superior* classes, the 'All Annotations' and 'Avoiding Checkpoints' approaches are the most expensive (with all the ambient sources). They consume around $2.8 - 21\times$ more energy and take around $1.4 - 33\times$ more time than the approaches in the *superior* classes. Therefore, we call these classes as '*expensive classes.*' The remaining classes can be categorized as '*intermediate classes*' as their performance lies between the two extremes (*superior* and *expensive*).

Please note that approaches in the *superior* classes do not require any programmer intervention or any compiler passes. In contrast, some of the approaches in the *expensive* and *intermediate* classes need the programmer to either annotate the program or write a customized library as in Chain or write compiler passes to annotate the code (e.g., Chinchilla uses five passes).

Ambient source specific results: After providing insights for the best and worst checkpointing classes from a general perspective, i.e., irrespective of the ambient source, we now discuss the relative performance of the approaches with respect to each source.

(1) *Solar Source:* After comparing the relative performance of the checkpointing classes when executed with solar energy (Figure 23(a-b)) to when executed with infinite energy (Figure 21(b-c)), we can make the following observation. For the *solar source*, all the checkpointing classes' relative performance does not degrade much compared to when the experiments were performed with an infinite power supply (Figure 21). For the superior and intermediate classes, the energy consumption and time taken are within 1.13× and 1.9×, respectively, with respect to an infinite power supply. For the expensive classes, the relative energy consumption remains almost the same as with an infinite power supply, while the relative time taken is around $1.8 - 2\times$ more.

(2) *RF Source:* Figure 23(c-d) shows that with an RF energy source, all the approaches' relative time performance is within 2.3× the baseline, indicating that all the approaches perform reasonably well with this source. Please note that the overhead of 2.3× is not prohibitive in this context; we need to contrast it with the $25-35\times$ overhead experienced by the 'expensive classes' for other ambient sources. However, from the perspective of the energy consumed, the approaches in the *superior* classes still consume considerably lesser energy. The reason behind the similar relative performance (execution time) across all the approaches (within 2.3× of the baseline) is that with the RF energy source, the approaches in the *superior* classes (which include the baseline approach - QCLP) perform poorly. Their off_time is increased drastically due to the reason described above in the 'Impact of varying capacitor sizes' subsection. Thus, the total time taken by the superior classes increases considerably, which makes the relative performance of all the approaches comparable.

(3) *Vibrational Source:* The performance of the *expensive* classes with the vibrational energy source becomes relatively worse (vis-a-vis the RF source) due to the fact that it is a low intensity source.

Summary:

- The considered checkpointing approaches can be categorized into three broad classes: superior, intermediate, and expensive.
- With the constant solar source, the relative performance of all the approaches is within 2× with respect to an infinite power supply, indicating that it is a reasonably good source.
- With the RF source, the time performance of the approaches in the superior class degrades in comparison to the performance with other sources. However, these approaches are still relatively better than other approaches.
- The vibrational source is inadequate for the approaches in the expensive class.

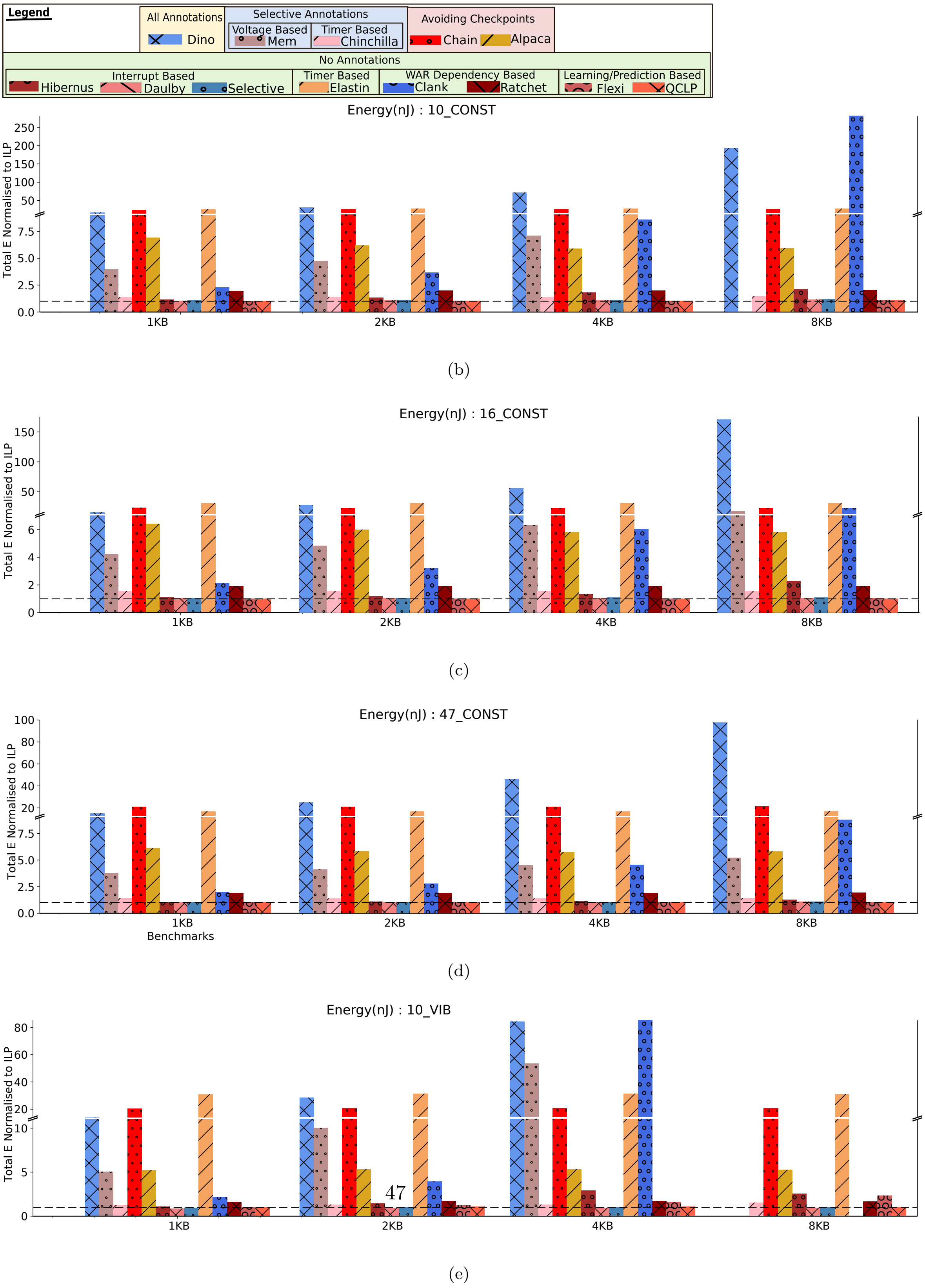
6.2.3. Evaluation with different SRAM sizes

Along with the capacitor size and the power source, the SRAM size may significantly affect the performance of certain checkpointing techniques. Larger SRAM sizes would increase the size of the checkpoint, thus increasing the energy consumed and time taken. Alternatively, a larger SRAM might lower the number of SRAM misses, thus increasing the performance.

Therefore, to study the impact of different SRAM sizes on a checkpointing technique's performance, we evaluated the behavior of different checkpointing techniques with varying sizes of SRAM - 1KB, 2KB, 4KB, and 8KB. These sizes have been taken as they are available in the most popular EHD devices.

Figures 24 and 25 show the energy consumption and time taken by different checkpointing techniques for different capacitor sizes and different power sources, respectively. In each graph, the title indicates the capacitance and the ambient source. E.g., 10_CONST means that a 10μF capacitor and a constant energy source are used for the experiment. The energy and time values have been normalized against the values of the QCLP approach (which has been considered the baseline for all our experiments). We summarize the main observations for the energy consumption plots.

1. Most of the checkpointing approaches follow a common pattern with increasing SRAM sizes – as the size of the SRAM increases, the energy consumed by the approach increases. However, the amounts by which the energy increases vary across the benchmarks, and it depends upon the type of the checkpointing approach. For example,
    - Dino, which belongs to the 'all annotations' class, is the most impacted, e.g., for a constant power source and a 10μF capacitor, the relative energy consumption increases from 16× to 30× to 71× to 191× as the SRAM size increases from 1KB to 2KB to 4KB to 8KB, respectively. This is because Dino blindly takes checkpoints at all the annotated locations. Considering that the



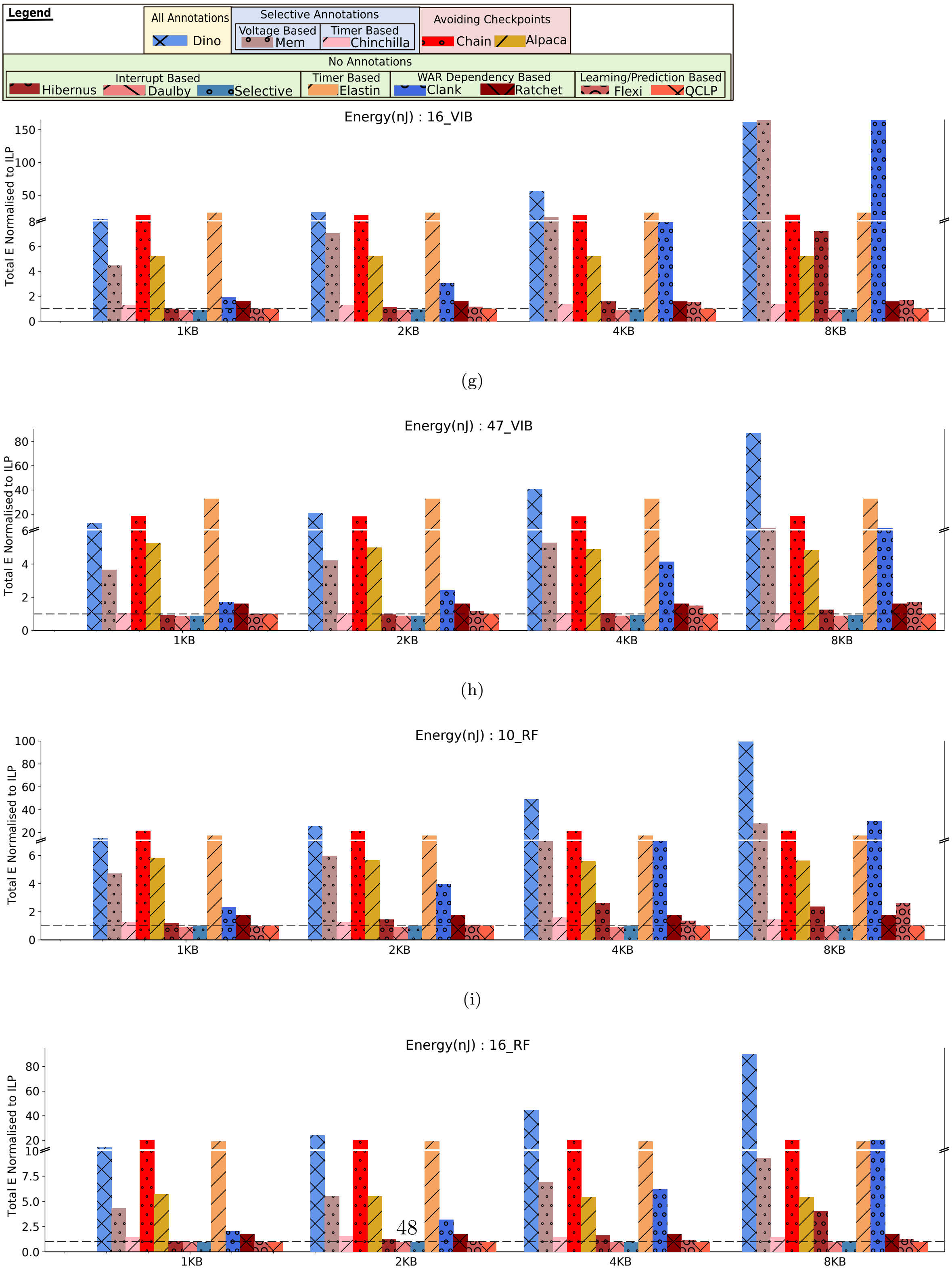
(b)

(c)

(d)

(e)

Figure 24: Relative energy consumption of the checkpointing approaches for different capacitor sizes (10μF, 16μ, and 47μF) and power sources: **(b,c,d)** constant source, **(e,g,h)** vibrational source, and **(i,j,l)** RF source.



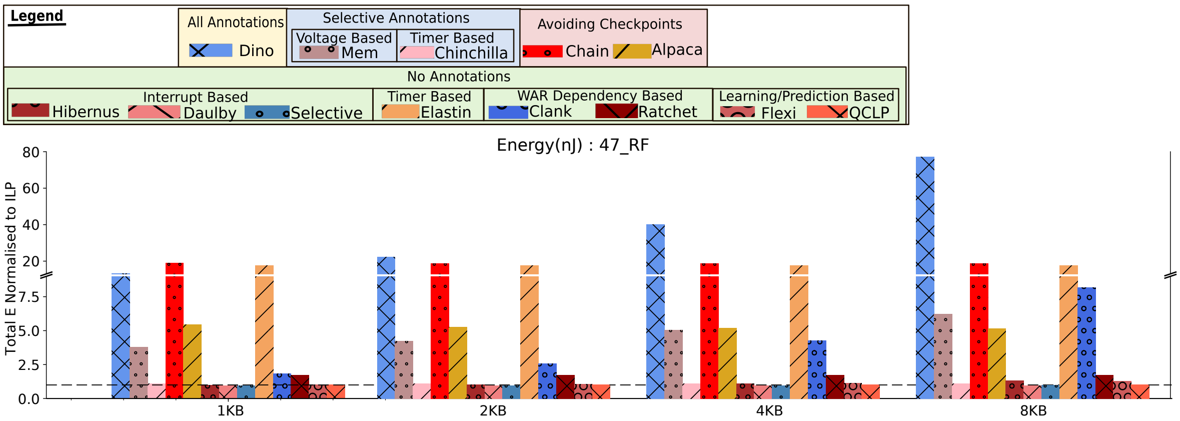
(g)

(h)

(i)

(j)

Figure 24: Relative energy consumption of the checkpointing approaches for different capacitor sizes (10μF, 16μ, and 47μF) and power sources: **(b,c,d)** constant source, **(e,g,h)** vibrational source, and **(i,j,l)** RF source.



(l)

Figure 24: Relative energy consumption of the checkpointing approaches for different capacitor sizes (10μF, 16μ, and 47μF) and power sources: **(b,c,d)** constant source, **(e,g,h)** vibrational source, and **(i,j,l)** RF source.

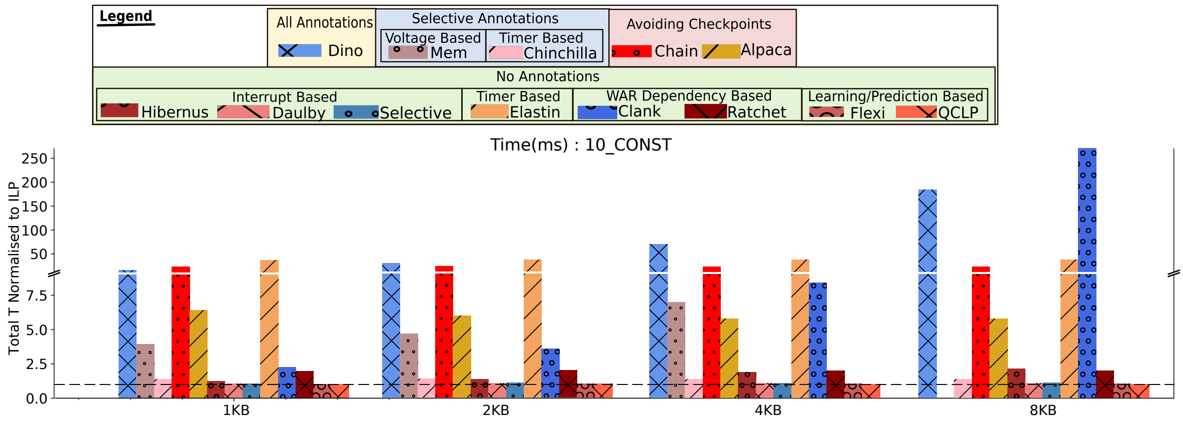
entire SRAM is checkpointed at each annotation, the total cost is enormous. Please note that for a common baseline across all the checkpoint approaches, we checkpoint the entire SRAM unless specified by the approach. This is also done in prior work [51, 7].

This behavior is seen across different ambient sources.

- Certain approaches are oblivious to an increase in the SRAM size. For example, approaches such as Chain, Elastin, Ratchet, Selective are not affected by a change in the SRAM size. Different approaches have different reasons for this. Chain (from the avoiding checkpoints class) does not take any checkpoint. Similarly, Elastin does not take a checkpoint but performs copy on write, which is independent of the SRAM size. Ratchet uses an entirely NVM system, so a checkpoint includes only the register variables. Finally, Selective does not checkpoint the entire SRAM, but only the data changed since the previous checkpoint is backed up.
- For some approaches, such as Chinchilla, Hibernus, Daulby, FlexiCheck, and QCLP the energy overhead increases as the SRAM size is increased; however, the relative energy consumption does not increase much as these approaches do not blindly checkpoint like Dino, but some intelligence is embedded in the algorithm, due to which the total energy of checkpoints does not increase much but increase in a similar proportion and the relative values remain almost constant.
- The approaches also benefit from access locality with an increase in the SRAM size. This results in more SRAM hits and thus a decrease in total energy consumption. This is clearly evident from approaches such as Alpaca, where the relative energy consumption decreased from 6.9× to 5.8× as the SRAM size increased from 1KB to 8KB.

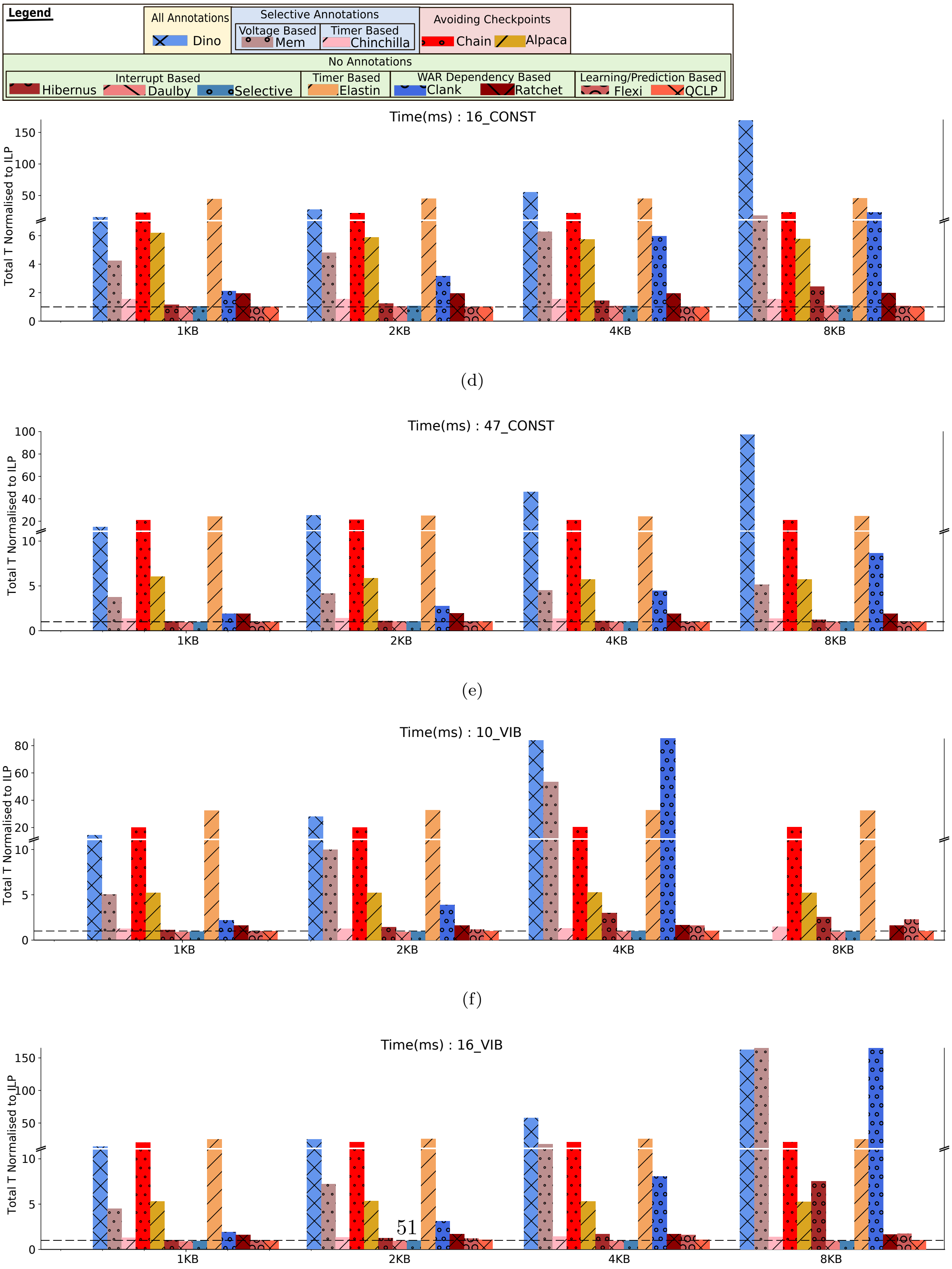
2. The energy consumed by an approach decreases with an increase in capacitor size. This effect is more clearly visible at larger SRAM sizes. For example, for the constant source, the maximum relative energy decreases from 250 to 150 as the capacitor size increases from 10μF to 16μF. It further reduces from 150nJ to 100 nJ as the capacitor size increases from 16μF to 47μF. This indicates that larger capacitors are beneficial.
3. As the checkpoint size increases, so does the energy consumption and the probability of failures. Therefore, it is possible that for large SRAM sizes, the total energy for checkpointing and the total number of reboots and hence energy spent in restores increases very much. This might result in the non-termination of the approach. For example, as we can see in Figure 24 (e), the Dino, Mementos, and Clank energy bars are absent for 8KB of SRAM. This is because the checkpoint-restore overhead increased to extremely high values and resulted in the non-termination of the approach. A similar observation can be made in Figure 24(b), where the energy bar for Mementos is missing for 8KB SRAM. However, if the capacitor size is increased from 10μF to 16μF, we can see that termination can be achieved in Mementos (Figure 24(c). This again indicates that larger capacitors are better.

Similar observations have been found in the latency plots (Figure 25).



(b)

Figure 25: Relative taken by the checkpointing approaches for different capacitor sizes (10μF, 16μ, and 47μF) and power sources: **(b,d,e)** constant source, **(f,g,i)** vibrational source, and **(j,k,l)** RF source.



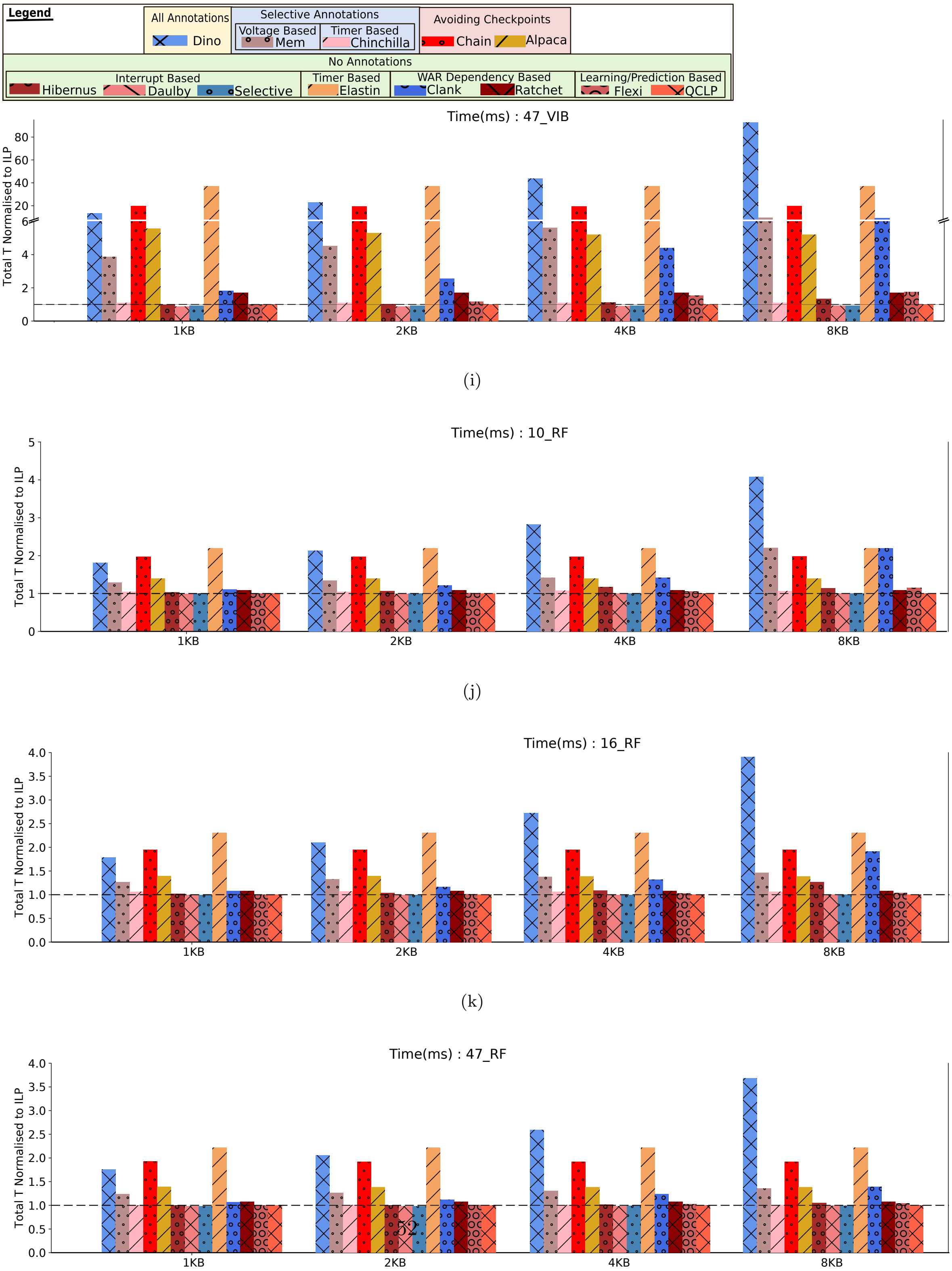
(d)

(e)

(f)

(g)

Figure 25: Relative taken by the checkpointing approaches for different capacitor sizes (10μF, 16μ, and 47μF) and power sources: **(b,d,e)** constant source, **(f,g,i)** vibrational source, and **(j,k,l)** RF source.



(i)

(j)

(k)

(l)

Figure 25: Relative taken by the checkpointing approaches for different capacitor sizes (10μF, 16μ, and 47μF) and power sources: **(b,d,e)** constant source, **(f,g,i)** vibrational source, and **(j,k,l)** RF source.

Summary:

- Dino is maximally impacted by increasing an SRAM size. Its relative energy consumption doubles as the SRAM size increases from 4KB to 8KB.
- Devices with large SRAM (8KB) and small capacitors (10μF) often face the issue of non-termination of approaches such as Dino, Mementos, and Clank. This can be avoided by using large capacitor sizes.
- Checkpointing the entire SRAM is not a good approach, especially if we have a large SRAM. In such cases, we should either intelligently reduce the number of checkpoints or checkpoint only the modifications (as in Selective).
- Approaches such as Ratchet, Elastin, Chain, and Selective are agnostic to SRAM sizes, while Alpaca benefits with increased SRAM size due to the increased spatial locality.

6.3. Qualitative Discussion

We further extended our analysis to compare the approaches on a host of different metrics such as the memory footprint - FRAMFootprint, average checkpoint size - AvgChkptSize, and the number of checkpoints, re-executions, and NVM accesses. Figure 26 presents the Kiviat plots of our results. In these plots, the region formed by joining the values of the results across the different axes indicates the overall utility of the approach. The larger the area, the poorer the approach is and hence, should have a lower preference. Figure 26(a) shows the Kiviat plots for approaches from the three classes: superior, intermediate, and expensive. We have considered two representative approaches from each class. The enclosed region's area is the maximum for the expensive classes, indicating that these are the worse classes. To see the cost of the intermediate and superior classes, we refer to Figure 26(b), which is a zoomed-in version of Figure 26(a). The minimum area for the superior classes confirms that these are the best approaches even with the additional metrics.

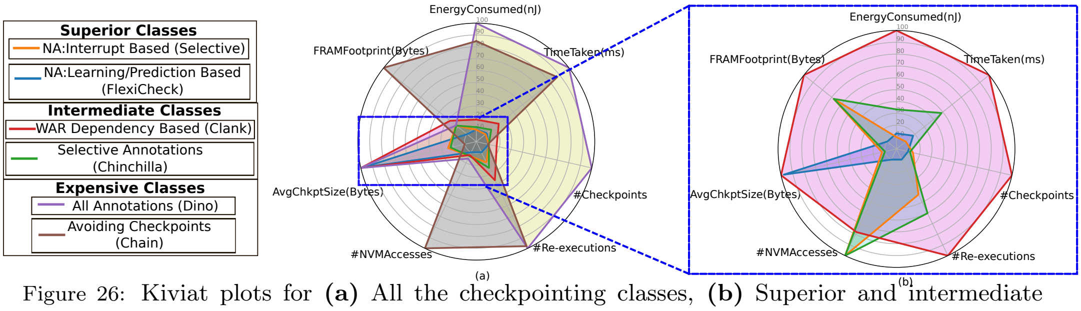


Figure 26: Kiviat plots for **(a)** All the checkpointing classes, **(b)** Superior and intermediate classes.

Summary: Among all the checkpointing approaches, the superior class's approaches are most preferred, while the approaches in the expensive classes are preferred the least. Thus, the system designers can choose among the superior approaches while designing their systems.

Though, as per our summary, a system designer should always use a checkpointing approach from the superior class, but it is possible that the chosen approach is not applicable according to the designer's EH device. For example, Daulby - an interrupt based (NA) approach from superior class might not be applicable to devices that do not have DMA support. Similarly, Clank - a WAR dependency based (NA) approach requires special hardware to track the WAR dependencies. So, going into the depths of a particular approach, a system designer might find a particular approach might not be applicable. So, we provide a set of recommendations for various checkpointing approaches, i.e., given a particular checkpointing approach, we provide various system configuration parameters that are preferable. These recommendations are summarized in Table 4. Using these recommendations, a system designer can choose an optimal scheme depending on her requirements. For example, the table indicates that 47uF is a preferred capacitor size for almost all the checkpointing approaches. However, suppose the system designer has only a 10uF sized capacitor. In that case, she can use either Selective, or Daulby, or FlexiCheck, as the performance of these approaches is similar for all the capacitor sizes. Similarly, if the device has 8KB of SRAM, then Dino, Mementos, and Clank approaches should not be considered.

Approach - Checkpointing class	Capacitance(uF)	SRAM size(KB)	Ambient source
*Dino [51] - All Annotations	$47 > 16 > 10$	$1 > 2 > 4 > 8$	CONST $\sim$ RF > VIB
*Mementos [7] - Device's Voltage Based (SA)	$47 > 16 > 10$	$1 > 2 > 4 > 8$	CONST > RF > VIB
Chinchilla [64] - Timer Based (SA)	$47 > 10 > 16$	$1 \sim 2 \sim 4 \sim 8$	CONST > VIB > RF
Chain [54] - Avoiding Checkpoints	$47 > 16 > 10$	$1 \sim 2 \sim 4 \sim 8$	CONST > RF > VIB
Alpaca [55] - Avoiding Checkpoints	$47 > 16 > 10$	$1 \sim 2 \sim 4 \sim 8$	CONST > RF > VIB
Hibernus [49] - Interrupt Based (NA)	$47 > 16 > 10$	$1 > 2 > 4 > 8$	CONST > VIB > RF
Daulby [48] - Interrupt Based (NA)	$47 \sim 16 \sim 10$	$1 \sim 2 \sim 4 \sim 8$	CONST $\sim$ VIB $\sim$ RF
Selective [103] - Interrupt Based (NA)	$47 \sim 16 \sim 10$	$1 \sim 2 \sim 4 \sim 8$	CONST > VIB > RF
Elastin [47] - Timer Based (NA)	$47 > 16 > 10$	$1 \sim 2 \sim 4 \sim 8$	CONST $\sim$ RF > VIB
*Clank [62] - WAR Dependency Based (NA)	$47 > 16 > 10$	$1 > 2 > 4 > 8$	CONST > RF > VIB
Ratchet [40] - WAR Dependency Based (NA)	$47 > 16 > 10$	$1 \sim 2 \sim 4 \sim 8$	CONST > VIB > RF
FlexiCheck [25] - Learning/Prediction Based (NA)	$47 \sim 16 \sim 10$	$1 > 2 > 4 > 8$	CONST > VIB > RF
QCLP [25] - Learning/Prediction Based (NA)	$47 > 16 > 10$	$1 > 2 > 4 > 8$	CONST > VIB > RF
A > B: A is preferable over B, A $\sim$ B: Both A and B result in similar performance, *: 8KB is not preferred			

Table 4: Recommended configurations for different checkpointing approaches

7. CONCLUSION AND FUTURE DIRECTIONS

Checkpointing plays a significant role in intermittent systems. This paper presents a comprehensive survey the checkpointing approaches proposed by researchers for correct and efficient program execution. Along with the theoretical comparison of the approaches, we performed an extensive evaluation of 13 state-of-the-art approaches and showed detailed time and energy figures for these approaches. We also evaluated the sensitivity of the approaches with respect to different capacitor sizes and ambient energy profiles.

Based on the energy consumed and performance of these approaches, we categorized them into three broad classes namely *superior, intermediate, and expensive.* For all the considered ambient sources, the approaches in the superior class showed the best performance, while those in the expensive class had a very low performance (also high energy consumption). The performance of the approaches in the intermediate class were in between.

We believe that such a survey would inspire both amateurs as well as seasoned professionals. For amateurs, it provides an easy guide to understand the topic deeply. At the same time, system designers can use our experimental analyses to decide which approach to choose. They can also propose new approaches based on the pros and cons of the existing approaches.

With recent advancements in processor technology, the current approaches might not remain efficient. For example, most of the current approaches are for single-threaded programs. However, modern embedded devices are being developed with multi-threading support. Thus, we would require more sophisticated and intelligent checkpointing algorithms. Notably, we need to decide if the checkpointing should be done per thread or for the entire program. Furthermore, we can have controlled context switching among the threads based on the thread's progress and the available ambient energy. Another direction to increase efficiency is to avoid taking the complete checkpoint at once. Instead, we can take it partially depending upon the available ambient energy and possibly coordinate that with the operation of DMA engines and peripherals.

Apart from efficiency, another important aspect is the safety of the checkpoint. Current approaches assume that a discharged EHD would restart and resume once the ambient energy is available. However, these approaches do not consider scenarios where the device might fail permanently, and there is no way to recover the device's data. This issue can be handled by storing the checkpoint in a distributed fashion using multiple EHDs. Each EHD would dynamically store partial data of its neighboring EHDs along with running its program. In case of permanent failure of one node, other nodes might collaborate to generate the failed device's state. Collaboration has its challenges in an energy harvesting scenario where the devices switch on/off randomly.

References

[1] Market Research Report, Energy Harvesting System Market, `https://www.marketsandmarkets.com/Market-Reports/energy-harvesting-market-734.html` (2020).

[2] M. T. Penella, J. Albesa, M. Gasulla, Powering Wireless Sensor Nodes: Primary Batteries versus Energy Harvesting, in: 2009 IEEE Instrumentation and Measurement Technology Conference, IEEE, 2009, pp. 1625–1630.

[3] Runar Finanger, Here's why energy-harvesting trumps batteries, `https://www.onio.com/article/energy-harvesting-trumps-batteries.html` (2020).

[4] R. Vullers, R. van Schaijk, I. Doms, C. Van Hoof, R. Mertens, Micropower energy harvesting, Solid-State Electronics 53 (7) (2009) 684–693.

[5] Peter Harrop, Environmental issues with energy harvesting, `https://www.printedelectronicsworld.com/articles/1245/environmental-issues-with-energy-harvesting` (2009).

[6] G. V. Merrett, B. M. Al-Hashimi, Energy-Driven Computing: Rethinking the Design of Energy Harvesting Systems, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, IEEE, 2017, pp. 960–965.

[7] B. Ransford, J. Sorber, K. Fu, Mementos: System Support for Long-Running Computation on RFID-Scale Devices, in: ASPLOS, 2011, pp. 159–170.

[8] S. Sudevalayam, P. Kulkarni, Energy Harvesting Sensor Nodes: Survey and Implications, IEEE Communications Surveys & Tutorials 13 (3) (2011) 443–461.

[9] S. Chalasani, J. M. Conrad, A Survey of Energy Harvesting Sources for Embedded Systems, in: Southeastcon, 2008. IEEE, IEEE, 2008, pp. 442–447.

[10] X. Lu, P. Wang, D. Niyato, D. I. Kim, Z. Han, Wireless Networks with RF Energy Harvesting: A Contemporary Survey, IEEE Communications Surveys & Tutorials 17 (2) (2015) 757–789.

[11] A. S. Weddell, M. Magno, G. V. Merrett, D. Brunelli, B. M. Al-Hashimi, L. Benini, A Survey of Multi-Source Energy Harvesting Systems, in: Design, Automation & Test in Europe Conference & Exhibition, 2013.

[12] T. Rault, A. Bouabdallah, Y. Challal, Energy Efficiency in Wireless Sensor Networks: a top-down survey, Computer Networks 67 (2014) 104–122.

[13] C. R. Valenta, G. D. Durgin, Harvesting Wireless Power: Survey of Energy-Harvester Conversion Efficiency in Far-Field, Wireless Power Transfer Systems, IEEE Microwave Magazine 15 (4) (2014) 108–120.

[14] S. Ulukus, A. Yener, E. Erkip, O. Simeone, M. Zorzi, P. Grover, K. Huang, Energy Harvesting Wireless Communications:A Review of Recent Advances, IEEE Journal on Selected Areas in Communications 33 (3) (2015) 360–381.

[15] K. Qiu, M. Zhao, Z. Jia, J. Hu, C. J. Xue, K. Ma, X. Li, Y. Liu, V. Narayanan, Design Insights of Non-volatile Processors and Accelerators in Energy Harvesting Systems, in: Proceedings of the 2020 on Great Lakes Symposium on VLSI, 2020, pp. 369–374.

[16] S. Umesh, S. Mittal, A survey of techniques for intermittent computing, Journal of Systems Architecture (2020) 101859.

[17] P. A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency control and recovery in database systems, Vol. 370, Addison-wesley Reading, 1987.

[18] R. Koo, S. Toueg, Checkpointing and rollback-recovery for distributed systems, IEEE Transactions on software Engineering (1) (1987) 23–31.

[19] G. Gobieski, B. Lucia, N. 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, pp. 199–213.

[20] A. S. Krishnan, C. Suslowicz, D. Dinu, P. Schaumont, Secure Intermittent Computing Protocol: Protecting State Across Power Loss, in: Design, Automation & Test in Europe Conference & Exhibition, IEEE, 2019.

[21] S. J. Roundy, Energy Scavenging for Wireless Sensor Nodes with a Focus on Vibration to Electricity Conversion, Ph.D. thesis, University of California, Berkeley Berkeley, CA (2003).

[22] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, V. Narayanan, Architecture Exploration for Ambient Energy Harvesting Nonvolatile Processors, in: IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2015, pp. 526–537.

[23] M. Habibzadeh, M. Hassanalieragh, A. Ishikawa, T. Soyata, G. Sharma, Hybrid Solar-Wind Energy Harvesting for Embedded Applications: Supercapacitor-Based System Architectures and Design Tradeoffs, IEEE Circuits and Systems Magazine 17 (4) (2017) 29–63.

[24] M. Prauzek, J. Konecny, M. Borova, K. Janosova, J. Hlavica, P. Musilek, Energy Harvesting Sources, Storage Devices and System Topologies for Environmental Wireless Sensor Networks: A Review, Sensors 18 (8) (2018) 2446.

[25] P. Singla, S. S. Singh, S. R. Sarangi, Flexicheck: An adaptive checkpointing architecture for energy harvesting devices, in: Design, Automation & Test in Europe Conference & Exhibition, IEEE, 2019.

[26] N. S. Shenck, J. A. Paradiso, Energy Scavenging With Shoe-Mounted Piezoelectrics, IEEE micro (3) (2001) 30–42.

[27] Z. Ghodsi, S. Garg, R. Karri, Optimal Checkpointing for Secure Intermittently-Powered IoT Devices, in: Proceedings of the 36th International Conference on Computer-Aided Design, IEEE Press, 2017, pp. 376–383.

[28] S. Carnot, Reflections on the Motive Power of Heat and on Machines Fitted to Develop that Power, J. Wiley, 1890.

[29] K. K. Win, X. Wu, S. Dasgupta, W. J. Wen, R. Kumar, S. Panda, Efficient Solar Energy Harvester for Wireless Sensor Nodes, in: 2010 IEEE International Conference on Communication Systems, IEEE, 2010, pp. 289–294.

[30] X. Jiang, J. Polastre, D. Culler, Perpetual Environmentally Powered Sensor Networks, in: Proceedings of the 4th international symposium on Information processing in sensor networks, IEEE,

2005, p. 65.

[31] A. Hoseinghorban, M. R. Bahrami, A. Ejlali, M. A. Abam, CHANCE: Capacitor Charging Management Scheme in Energy Harvesting Systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[32] A. Colin, E. Ruppel, B. Lucia, A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices, in: Proceedings of the Twenty-Third International Conference on ASPLOS, ACM, 2018.

[33] D. Zhang, Y. Liu, X. Sheng, J. Li, T. Wu, C. J. Xue, H. Yang, Deadline-aware Task Scheduling for Solar-powered Nonvolatile Sensor Nodes with Global Energy Migration, in: Proceedings of the 52nd Annual Design Automation Conference, ACM, 2015, p. 126.

[34] J. Hester, L. Sitanayah, J. 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, pp. 5–16.

[35] A. Gomez, L. Sigrist, M. Magno, L. Benini, L. Thiele, Dynamic Energy Burst Scaling for Transiently Powered Systems, in: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016, pp. 349–354.

[36] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, L. Benini, Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices, Transactions on Computer-Aided Design of Integrated Circuits and Systems 35 (12) (2016) 1968–1980.

[37] C. Wang, An Ultra-Low Power Voltage Regulator System for Wireless Sensor Networks Powered by Energy Harvesting, Ph.D. thesis (2014).

[38] A. Didioui, Energy-Aware Transceiver for Energy Harvesting Wireless Sensor Networks, Ph.D. thesis, Université Rennes 1 (2014).

[39] J. Eriksson, A. Dunkels, N. Finne, F. Osterlind, T. Voigt, Mspsim–an extensible simulator for msp430-equipped sensor boards, in: Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Vol. 118, 2007.

[40] J. Van Der Woude, M. Hicks, Intermittent Computation without Hardware Support or Programmer Intervention, in: 12th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 16), 2016, pp. 17–32.

[41] H. Li, Y. Liu, Q. Zhao, Y. Gu, X. Sheng, G. Sun, C. Zhang, M.-F. Chang, R. Luo, H. Yang, An Energy Efficient Backup Scheme with Low Inrush Current for Nonvolatile SRAM in Energy Harvesting Sensor Nodes, in: 2015 Design, Automation & Test in Europe Conference & Exhibition, IEEE, 2015, pp. 7–12.

[42] H. Jayakumar, A. Raha, V. Raghunathan, HYPNOS: An Ultra-Low Power Sleep Mode with SRAM Data Retention for Embedded Microcontrollers, in: Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis, 2014, pp. 1–10.

[43] G. Chen, H. Ghaed, R.-u. Haque, M. Wieckowski, Y. Kim, G. Kim, D. Fick, D. Kim, M. Seok, K. Wise, et al., A Cubic-Millimeter Energy-Autonomous Wireless Intraocular Pressure Monitor, in: 2011 IEEE International Solid-State Circuits Conference, IEEE, 2011, pp. 310–312.

[44] H. Williams, X. Jian, M. Hicks, Forget Failure: Exploiting SRAM Data Remanence for Low-overhead Intermittent Computation, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 69–84.

[45] F. Su, K. Ma, X. Li, T. Wu, Y. Liu, V. Narayanan, Nonvolatile Processors: Why is it Trending?, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, IEEE, 2017.

[46] C.-K. Kang, C.-H. Lin, P.-C. Hsiu, M.-S. Chen, HomeRun: HW/SW Co-Design for Program Atomicity on Self-Powered Intermittent Systems, in: Proceedings of the International Symposium on Low Power Electronics and Design, 2018, pp. 1–6.

[47] J. Choi, H. Joe, Y. Kim, C. Jung, Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE, 2019, pp. 331–344.

[48] T. Daulby, A. Savanth, G. Merrett, A. S. Weddell, Improving the Forward Progress of Transient Systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[49] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, L. Benini, Hibernus: Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems, IEEE Embedded Systems Letters 7 (1) (2014) 15–18.

[50] W.-M. Chen, P.-C. Hsiu, et al., Enabling Failure-resilient Intermittent Systems Without Runtime Checkpointing, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[51] B. Lucia, B. Ransford, A Simpler, Safer Programming and Execution Model for Intermittent Systems, ACM SIGPLAN Notices 50 (6) (2015) 575–585.

[52] K. Maeng, B. Lucia, Supporting Peripherals in Intermittent Systems with Just-In-Time Checkpoints, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 1101–1116.

[53] A. Colin, B. Lucia, Termination Checking and Task Decomposition for Task-Based Intermittent Programs, in: Proceedings of the 27th International Conference on Compiler Construction, 2018, pp. 116–127.

[54] A. Colin, B. Lucia, Chain: Tasks and Channels for Reliable Intermittent Programs, in: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2016.

[55] K. Maeng, A. Colin, B. Lucia, Alpaca: Intermittent Execution without Checkpoints, Proceedings of the ACM on Programming Languages 1 (OOPSLA) (2017) 96.

[56] W.-M. CHEN, T. KUO, P.-C. HSIU, Heterogeneity-aware multicore synchronization for intermittent systems (2021).

[57] E. Ruppel, B. Lucia, Transactional Concurrency Control for Intermittent, Energy-Harvesting Computing Systems, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 1085–1100.

[58] W.-M. Chen, Y.-T. Chen, P.-C. Hsiu, T.-W. Kuo, Multiversion Concurrency Control on Intermittent Systems, in: Proc. of IEEE/ACM ICCAD, 2019.

[59] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, P.-C. Hsiu, Everything Leaves Footprints: Hardware Accelerated Intermittent Deep Inference, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39 (11) (2020) 3479–3491.

[60] H. R. Mendis, P.-C. Hsiu, Accumulative Display Updating for Intermittent Systems, ACM Transactions on Embedded Computing Systems (TECS) 18 (5s) (2019) 1–22.

[61] H. Jayakumar, A. Raha, V. Raghunathan, QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers, in: VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on, IEEE, 2014, pp. 330–335.

[62] M. Hicks, Clank: Architectural Support for Intermittent Computation, ACM SIGARCH Computer Architecture News 45 (2) (2017) 228–240.

[63] H. Jayakumar, A. Raha, J. R. Stevens, V. Raghunathan, Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices, ACM Transactions on Embedded Computing Systems (TECS) 16 (3) (2017) 1–23.

[64] K. Maeng, B. Lucia, Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing, in: 13th $USENIX$ Symposium on Operating Systems Design and Implementation ($OSDI$ 18), 2018, pp. 129–144.

[65] N. Bhatti, L. Mottola, Efficient State Retention for Transiently-powered Embedded Sensing, in: International Conference on Embedded Wireless Systems and Networks, 2016, pp. 137–148.

[66] F. A. Aouda, K. Marquet, G. Salagnac, Incremental checkpointing of program state to NVRAM for transiently-powered systems, in: 2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), IEEE, 2014, pp. 1–4.

[67] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, L. Mottola, Fast and Energy-efficient State Checkpointing for Intermittent Computing, ACM Transactions on Embedded Computing Systems 19 (6) (2020).

[68] G. Berthou, K. Marquet, T. Risset, G. Salagnac, MPU-based incremental checkpointing for transiently-powered systems, in: 2020 23rd Euromicro Conference on Digital System Design (DSD), IEEE, 2020, pp. 89–96.

[69] D. Pala, I. Miro-Panades, O. Sentieys, Freezer: A Specialized NVM Backup Controller for Intermittently-Powered Systems, Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[70] M. Xie, C. Pan, M. Zhao, Y. Liu, C. J. Xue, J. Hu, Avoiding Data Inconsistency in Energy Harvesting Powered Embedded Systems, ACM Transactions on Design Automation of Electronic Systems (TODAES) 23 (3) (2018) 1–25.

[71] X. Zhang, C. Patterson, Y. Liu, C. Yang, C. J. Xue, J. Hu, Low Overhead Online Checkpoint for Intermittently Powered Non-volatile FPGAS, in: 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), IEEE, 2018, pp. 238–244.

[72] S. T. Sliper, D. Balsamo, N. Nikoleris, W. Wang, A. S. Weddell, G. V. Merrett, Efficient State Retention through Paged Memory Management for Reactive Transient Computingor, in: Proceedings

of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[73] Y. Liu, J. Yue, H. Li, Q. Zhao, M. Zhao, C. J. Xue, G. Sun, M.-F. Chang, H. Yang, Data Backup Optimization for Nonvolatile SRAM in Energy Harvesting Sensor Nodes, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36 (10) (2017) 1660–1673.

[74] M. Xie, M. Zhao, C. Pan, H. Li, Y. Liu, Y. Zhang, C. J. Xue, J. Hu, Checkpoint Aware Hybrid Cache Architecture for NV Processor in Energy Harvesting Powered Systems, in: 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS), IEEE, 2016, pp. 1–10.

[75] F. Li, K. Qiu, M. Zhao, J. Hu, Y. Liu, Y. Guan, C. J. Xue, Checkpointing-Aware Loop Tiling for Energy Harvesting Powered Nonvolatile Processors, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38 (1) (2018) 15–28.

[76] K. Maeng, B. Lucia, Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020.

[77] J. De Winkel, V. Kortbeek, J. Hester, P. Pawełczak, Battery-Free Game Boy, Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies 4 (3) (2020) 1–34.

[78] S. S. Baghsorkhi, C. Margiolas, Automating Efficient Variable-Grained Resiliency for Low-Power IoT Systems, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 38–49.

[79] J. Choi, Q. Liu, C. Jung, CoSpec: Compiler Directed Speculative Intermittent Computation, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 399–412.

[80] A. Hoseinghorban, A. M. H. H. Monazzah, M. Bazzaz, B. Safaei, A. Ejlali, COACH: Consistency Aware Check-pointing for Nonvolatile Processor in Energy Harvesting Systems, IEEE Transactions on Emerging Topics in Computing (2019).

[81] A. Maioli, L. Mottola, M. H. Alizai, J. H. Siddiqui, Discovering the Hidden Anomalies of Intermittent Computing.

[82] Z. Li, Y. Liu, D. Zhang, C. J. Xue, Z. Wang, X. Shi, W. Sun, J. Shu, H. Yang, HW/SW Co-design of Nonvolatile IO System in Energy Harvesting Sensor Nodes for Optimal Data Acquisition, in: Proceedings of the 53rd Annual Design Automation Conference, 2016, pp. 1–6.

[83] A. Rodriguez Arreola, D. Balsamo, G. V. Merrett, A. S. Weddell, RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems, Sensors 18 (1) (2018) 172.

[84] A. Branco, L. Mottola, M. H. Alizai, J. H. Siddiqui, Intermittent Asynchronous Peripheral Operations, in: Proceedings of the 17th Conference on Embedded Networked Sensor Systems, 2019, pp. 55–67.

[85] Y.-C. Lin, P.-C. Hsiu, T.-W. Kuo, Autonomous I/O for Intermittent IoT Systems, in: 2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), IEEE, 2019, pp. 1–6.

[86] J. Hester, K. Storer, J. Sorber, Timely Execution on Intermittently Powered Batteryless Sensors, in: Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, 2017, pp. 1–13.

[87] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burleson, J. Sorber, Persistent Clocks for Batteryless Sensing Devices, ACM Transactions on Embedded Computing Systems (TECS) 15 (4) (2016) 1–28.

[88] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, P. Pawełczak, Time-sensitive Intermittent Computing Meets Legacy Software, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 85–99.

[89] M. Surbatovich, L. Jia, B. Lucia, I/O Dependent Idempotence Bugs in Intermittent Systems, Proceedings of the ACM on Programming Languages 3 (OOPSLA) (2019) 1–31.

[90] H. Zhang, M. Salajegheh, K. Fu, J. Sorber, Ekho: Bridging the Gap Between Simulation and Reality in Tiny Energy-Harvesting Sensors, in: Proceedings of the 4th Workshop on Power-Aware Computing and Systems, 2011, pp. 1–5.

[91] A. Colin, G. Harvey, B. Lucia, A. P. Sample, An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems, ACM SIGARsCH Computer Architecture News 44 (2) (2016) 577–589.

[92] K. Geissdoerfer, M. Chwalisz, M. Zimmerling, Shepherd: A Portable Testbed for the Batteryless IoT, in: Proceedings of the 17th Conference on Embedded Networked Sensor Systems, 2019, pp. 83–95.

[93] M. Dahiya, S. Bansal, Automatic Verification of Intermittent System, in: International Conference on Verification, Model Checking, and Abstract Interpretation, Springer, 2018, pp. 161–182.

[94] G. Berthou, P.-É. Dagand, D. Demange, R. Oudin, T. Risset, Intermittent Computing with Peripherals, Formally Verified, in: The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2020, pp. 85–96.

[95] M. Surbatovich, L. Jia, B. Lucia, Towards a Formal Foundation of Intermittent Computing, arXiv preprint arXiv:2007.15126 (2020).

[96] J. Li, M. Zhao, L. Ju, C. J. Xue, Z. Jia, Maximizing Forward Progress with Cache-aware Backup for Self-powered Non-volatile Processors, in: 54th ACM/EDAC/IEEE Design Automation Conference, 2017.

[97] A. Mirhoseini, E. M. Songhori, F. Koushanfar, Idetic: A High-level Synthesis Approach for Enabling Long Computations on Transiently-powered ASICs, in: 2013 IEEE International Conference on Pervasive Computing and Communications (PerCom), IEEE, 2013, pp. 216–224.

[98] A. Mirhoseini, B. D. Rouhani, E. Songhori, F. Koushanfar, Chime: Checkpointing Long Computationson Intermittently Energized IoT Devices, IEEE Transactions on Multi-Scale Computing Systems 2 (4) (2016) 277–290.

[99] N. A. Bhatti, L. Mottola, Harvos: Efficient code instrumentation for transiently-powered embedded sensing, in: 16th ACM/IEEE International Conference on Information Processing in Sensor Networks, IEEE, 2017.

[100] N. Shoemaker, R. Piskac, M. Santolucito, Towards Checkpoint Placement for Dynamic Memory Allocation in Intermittent Computing, in: Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis, 2020, pp. 20–22.

[101] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, L. Mottola, The Betrayal of Constant Power×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, pp. 97–109.

[102] C. Pan, M. Xie, S. Han, Z.-H. Mao, J. Hu, Modeling and Optimization for Self-powered Non-volatile IoT Edge Devices with Ultra-low Harvesting Power, ACM Transactions on Cyber-Physical Systems 3 (3) (2019) 1–26.

[103] W. Song, X. Cai, M. Zhao, Z. Shen, Z. Jia, A lightweight online backup manager for energy harvesting powered nonvolatile processor systems, Journal of Systems Architecture 113 (2021).

[104] W. Fan, Y. Zhang, W. Song, M. Zhao, Z. Shen, Z. Jia, Q-learning Based Backup for Energy Harvesting Powered Embedded Systems, in: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2020, pp. 1247–1252.

[105] W. S. Lim, C.-H. Tu, C.-F. Wu, Y.-H. Chang, iCheck: Progressive Checkpointing for Intermittent Systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[106] P. Zhang, D. Ganesan, B. Lu, Quarkos: Pushing the operating limits of micro-powered sensors, in: Presented as part of the 14th Workshop on Hot Topics in Operating Systems, 2013.

[107] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, P. Pawełczak, Dynamic Task-based Intermittent Execution for Energy-harvesting Devices, ACM Transactions on Sensor Networks (TOSN) 16 (1) (2020) 1–24.

[108] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, E. Peter, Tejas: A java based versatile micro-architectural simulator, in: PATMOS, 2015.