Dependable Dynamic Checkpoints for Batteryless Devices

Vito Kortbeek
Dependable Dynamic Checkpoints for Batteryless Devices

Master’s Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Vito Kortbeek
vito.kortbeek@student.tudelft.nl
vito.kortbeek@gmail.com

19th February 2019
Abstract

Tiny batteryless sensing systems that do not have a constant supply of power have become an increasingly appealing way to gather data, in hard to reach places, at low cost. These systems suffer from frequent and unpredictable power loss. Preserving information through power failures is an ancillary challenge. Unfortunately, state-of-the-art systems do not fully tackle this issue. We introduce a system that offloads the implementation burden and evaluate it against state-of-the art alternatives.
Preface

Before you lies the work resulting from my MSc thesis, which was carried out from July 2018 to February 2019 to fulfil the graduation requirements of the Embedded Systems Programme at the TU Delft. This MSc thesis has been carried out within the Embedded and Networked Systems group under the supervision of Dr. Przemyslaw Pawelczak and was co-supervised by Dr. Kasum Sinan Yildirim. Batteryless sensors bring forth the need for intermittent computing, which is an increasingly popular topic within the embedded systems community. This inspired me to find a less cumbersome and restricted way for application developers to develop systems running on energy harvested power. The resulting system is a step towards practical application development for intermittently powered systems. This project was executed together with Dr. Josiah Hester (Northwestern University), Dr. Jacob Sorber (Clemson University) and Abu Bakar (Northwestern University) in addition to my supervisors.

I would like to thank my supervisor Przemyslaw Pawelczak for giving me this opportunity and guiding me throughout all the phases of this thesis. I would also like to give a special thanks to my co-supervisor Kasum Sinan Yildirim for not only guiding me through the process, but who also intensively supported and advised me with regard to the actual design and implementation. Without our critical discussions this system and accompanying thesis would not have existed. I also benefited from debating issues and concepts with Amjad Majids during the initial stages of my research. During the final stages I also enjoyed the help and guidance of Dr. Josiah Hester, Dr. Jacob Sorber and Abu Bakar, who introduced some new ideas which made the work more complete. My mother deserves a particular note of thanks, you provided me with support that allowed me to undertake this adventure.

Vito Kortbeek

Delft, The Netherlands
19th February 2019
## Contents

**Preface**

1. Introduction 1
   1.1 Challenges of Intermittent Computing 1
   1.2 State of the Art 2
   1.3 Contributions 3
   1.4 Outline 3

2. Background 5
   2.1 Anatomy of a Battery-Free Sensor 5
       2.1.1 Data Consistency 6
       2.1.2 Forward Progress with Checkpoints 6
       2.1.3 Forward Progress with Tasks 7
   2.2 Porting Existing Programs 7
       2.2.1 Problems with Time in C 8

3. Related Work 11
   3.1 Automatic Checkpointing Systems 11
   3.2 Other Checkpointing Systems 12
   3.3 Task-Based Programming Models 12
   3.4 Compiler-Based Systems 13
   3.5 Non-volatile Processors Architectures 13

4. Design 15
   4.1 Design Considerations 15
       4.1.1 Task-Based Transformation Versus Checkpointing 15
       4.1.2 Checkpointing Scalability 17
       4.1.3 Pointer-Handling and Memory Consistency 17
       4.1.4 Double-Buffering Memory Requirements 17
       4.1.5 Timely Execution 18
   4.2 Scalable Checkpoints 18
       4.2.1 Checkpointing Time 18
       4.2.2 Stack Segmentation 19
       4.2.3 Segment-Swap Driven Checkpoints 21
Chapter 1

Introduction

Tiny embedded computing systems and sensor networks have created a revolution—changing how we monitor buildings and other infrastructure, treat disease, and protect endangered wildlife—but, the decades-old vision of ubiquitous computing (and now the Internet-of-Things) are frustrated by energy storage issues. Today, most untethered devices rely on batteries—fragile, short-lived (even rechargeables), bulky, relatively expensive, chemical energy stores. Enabled by improvements in energy harvesting technologies and low-power circuit design, as well as the commercialization of byte-addressable non-volatile memories (like FRAM, MRAM, and ReRAM), batteryless devices with minimal energy storage that run solely off energy scavenged from the environment, promise a more scalable and sustainable alternative.

1.1 Challenges of Intermittent Computing

Reducing energy storage to near zero comes with consequences across the board, from the architecture to the programmer and users. As energy harvesting conditions fluctuate, power failures can occur frequently—as often as one hundred times per second in some cases [43]—clearing the call stack, program counter, and volatile register values, making it difficult to ensure forward progress (Fig. 2.1). With off-the-shelf microcontrollers’ mixed memory volatility (SRAM and FRAM[1]), maintaining memory consistency is challenging as writes to non-volatile memory are not typically captured by checkpoints. Additionally as power outages increase, data gathered and processed before a power failure may no longer be relevant when a device turns back on.

To be successful, intermittent runtime systems must (i) maintain forward progress despite power failures, (ii) ensure memory consistency of the applications despite frequent power failures, (iii) provide mechanisms for timely

---

[1]Texas Instruments’ MSP430FR* processor line is a common example.
execution of the application tasks, and finally (iv) simplify the process of creating and porting programs that work well on intermittent power. Existing systems achieve the first three goals, but the cognitive burden of creating and porting intermittent applications remains challenging—especially when they involve legacy code, written for reliable power.

1.2 State of the Art

Today’s intermittent computing systems use two general programming approaches: traditional standard C programs paired with automatic or manual checkpointing, and task-based programs where developers break programs into idempotent and atomic tasks (that can be restarted when interrupted by a power outage) and then describe how control and data flow between tasks. Other approaches consist of: compiler assisted which is based on transforming the intermediate representation through compiler passes which reorder the instructions to create idempotent regions with regard to the non-volatile memory [50], hardware assisted where additional hardware is added to track memory accesses and detect write after read violations [21], and fully non-volatile processors which do not require additional software support to guarantee consistency as every part of the system is non-volatile [31, 32].

Static checkpointing systems [42, 30, 51] journal the processor’s volatile state; i.e. registers, stack and heap, in non-volatile memory at specific points in the program code annotated at compile time by the programmer and/or the compiler. Dynamic checkpointing systems [22, 4, 3, 5, 21] monitor the supply voltage or execution time at runtime to detect if the voltage is under a pre-defined threshold in order to journal the volatile state. In all cases, upon resuming operation the computation is continued from the latest journaled volatile state. In task-based systems [8, 43, 14, 52] (which are all static), the programmer decomposes an application into a collection of tasks, provides inter-task input/output relations and control flow at compile time. The runtime ensures atomic execution of the tasks; i.e. all-or-nothing semantics so that volatile and non-volatile memory structures remain consistent, and switches to the next task in the control flow. Task-based runtimes have less runtime overhead than checkpointing-based systems since they journal only the current task pointer and the inputs/outputs of the current task. As a rule, checkpointing systems reduce the cognitive burden of porting, but do not handle time, have high memory overhead, and are slow because of large checkpoints. Inversely, task-based systems are usually faster and use less memory, but developers must transform a program to fit the programming model, which may involve significant effort. This work seeks to find a middle ground.
1.3 Contributions

**VISP: Virtualized Intermittent Software Platform.** VISP finds the middle ground by enabling programmers to execute unaltered C-programs on hardware supporting a form of non-volatile memory in addition to volatile memory, and optionally annotate the program with simple structures to specify custom timing requirements—protecting against timing errors not possible in non-intermittent programs. Compared to existing automatic checkpointing systems that require significant memory overhead and restore times, VISP limits the size of a checkpoint to a segment of stack plus the register file, greatly reducing the time it takes to checkpoint and to restore a checkpoint, and providing a upper bound on these times.

In all prior systems, porting an embedded application that runs on continu-ous power to work with intermittent power was challenging; either because of the cognitive burden of translating a program to sequential tasks, the inability to handle data freshness and time in general, or the challenge of dealing with intermittent power. With VISP, programmers can run any C program, even ones that use important but often unsupported \[42\] features like pointers, on an intermittent device. We make the following contributions in this work:

1. Introduction of *time sensitivity semantics for task-less intermittent systems.*
3. A *solution for the problem of fixed worst case checkpoint* and checkpoint restore time.
4. A novel combination of *low overhead and energy efficient methods* to significantly reduce memory overheads for task-less intermittent programs.
5. VISP as an open-source tool for developers via [https://github.com/iiKoe/intermittent-checkpoint](https://github.com/iiKoe/intermittent-checkpoint)

A paper on which this thesis is based has been submitted to [USENIX ATC ’19](https://github.com/iiKoe/intermittent-checkpoint).

1.4 Outline

This thesis will firstly cover the background and related work. Next our solution is specified and its implementation details are discussed. After this the system is evaluated against both a custom application and a set of benchmarks also used to evaluate related systems. A user study follows next outlining some issues regarding usage difficulties on the user end—where the user is the application developer using VISP or a competing solution. Lastly this thesis contains a discussion outlining some limitations followed
by a section concluding our findings and results closing with possible future additions and improvements.
Chapter 2

Background

For almost two decades, the community around wireless sensor networks has worked together to enable long-term, affordable, sustainable sensing across many application domains, ranging from wildlife tracking [53, 15, 44], to infrastructure monitoring [7, 14], health and human sensing [16, 29, 36], autonomous vehicles [13], and even space exploration [35]. During this time terms like “ubiquitous sensing,” or “smart dust,” [23] the “TerraSwarm,” [26] and the “Internet-of-Things” have been used to describe this single vision of computing.

Recently, wireless sensing devices that work without batteries have become viable [18, 43] due to advances in miniaturization of energy harvesters for solar, RF, and kinetic energy, and energy-efficient microcontrollers with non-volatile memory [47]. Making these devices work is challenging, because of intermittent power failures caused by unreliable and sporadic energy sources as shown in Fig. 2.1. However, the benefits are well worth it—besides reduced cost, size, and weight, these devices promise decade long lifetimes in the field without requiring human maintenance or replacement.

2.1 Anatomy of a Battery-Free Sensor

Like Telos [38] or MicaZ [39] motes, batteryless platforms use ultra low-power micro-controllers (MCUs); e.g. MSP430FR5969 [48], whose main architectural components; e.g. registers and main memory, are volatile, while the code and data retained from power failure to power failure are held in non-volatile byte addressable RAM (like Ferroelectric RAM (FRAM) [49]). The MCU coordinates sensing and communication as well as computation onboard. Batteryless devices like the WISP [43], Capybara [10], or Flicker [17, 18] have hardware support for intelligently managing incoming energy. Flicker charging circuitry wakes up the MCU when certain energy thresholds are met for particular sensing tasks. When the energy in the capacitor is depleted, the device dies due to a power failure that resets the volatile state of the
Figure 2.1: **Energy harvesting causes intermittent power failures; complicating program execution.** Carefully inserted checkpoints can preserve forward progress despite power failures—but create potential consistency violations and waste computation. This figure shows the voltage and program execution trace with checkpoints across intermittent power failures.

device; e.g. the contents of stack, program counter, registers are lost. This prevents existing programs and libraries designed for continuously-powered computers from running correctly on batteryless sensors.

### 2.1.1 Data Consistency

Complicating these checkpointing systems are consistency requirements (shown in Fig. 2.2); intermittent operation can easily make programs fall into an errant state [30]. In task-based systems, memory consistency is ensured by storing inputs and outputs of tasks at different locations in non-volatile memory. Since the inputs and outputs are separated, the re-execution of any task will read the same inputs and in turn will output the same results—tasks are idempotent [11]. Consistency violations identified in previous work include the Write After Read (WAR) violation illustrated in Fig. 2.2. After a checkpoint, non-volatile global variable \( len \) is changed, but these actions are not included in the checkpoint. When the checkpoint is restored, \( len \) is again updated, leading to an incorrect value of \( len \).

### 2.1.2 Forward Progress with Checkpoints

Checkpointing volatile state to non-volatile memory—which can be restored after recovering from a power failure—allows battery-free devices to maintain
Figure 2.2: Consistency violations encountered with automatic checkpointing schemes are shown. Non-volatile variable \( len \) is initialised to 0. During execution without interruption \( len \) will be 1 at the end of the code section (a). When a reboot occurs after the two statements the system is reset to the last checkpoint. This can lead to a Write-After-Read (WAR) violation with improper checkpoint placement (b).

forward progress across power failures. Fig. 2.1 shows an example of this with checkpointing at boundaries of basic blocks in a program; this method reduces wasted computation and allows programs to complete. Many checkpointing systems have been explored [42, 51, 30, 22, 5, 3] which back up the processor’s whole hardware state at statically defined or dynamically decided points. Of course, the main challenge lies in determining what to backup and when.

2.1.3 Forward Progress with Tasks

In task-based systems [8, 33, 19, 52] programmers decompose a program into atomic and idempotent tasks with defined control and data flow, and sometimes timing information [19]. Program state is distributed among the tasks and only the state of the active task is backed up at each task transition to ensure forward progress.

2.2 Porting Existing Programs

There is a massive set of existing applications and legacy code—written both in plain C and for software platforms such as TinyOS [28], Contiki [12] and others—from the past two decades of wireless sensor network deployment. Unfortunately, the existing corpus of applications is difficult, or impracticable to port to work with battery-free platforms and applications—especially
when porting to a task-based programming model. Fig. 2.3 illustrates these challenges.

For all task-based programming models, the program development is tightly coupled with reasoning about the outcomes of power failures. To make a successful port, the programmer needs to (i) identify and annotate persistent variables; (ii) provide a task-division and annotate task boundaries/inter-task dependencies; and (iii) specify an explicit control/data flow. C language libraries must be re-implemented to be resilient to power failure. Porting old code or developing new code becomes time-consuming and applications become incomprehensible and hard to debug. Task decomposition in particular is hard [9], as task boundaries are not defined by logical breaks in a program (such as those shown on the left part of Fig. 2.3: classify, transform, etc.) but are best set based on the energy cost of blocks of code, and/or data interconnections between code. This reasoning is difficult for developers, especially amateurs, who may not have the background or tools to estimate energy consumption. Moreover, this decomposition is not as useful for determining intent (as shown in the right part of Fig. 2.3) since logical tasks are broken up into seemingly arbitrary sub-tasks.

2.2.1 Problems with Time in C

With checkpointing systems porting seems simpler. No task decomposition is necessary; there is no need to re-implement ordinary C language implementations of libraries from scratch. Static checkpointing systems require the programmer to go over the code and annotate the checkpointing boundaries and the persistent variables to be versioned. Automatic checkpointing systems do not have this requirement but over-instrument the executable with checkpoints and often do not allow for a complete use of pointers.

We have encountered three other types of consistency violations in addition to the one in Fig. 2.2 (where errant execution happens because of intermittent power, leading to program states not possible), all having to do with time, shown in Fig. 2.4. The errors stem from the fact that clocks internal to the MCU are reset after each power failure, meaning that devices have difficulty tracking how long they have been off [20, 10]; even when using external timekeepers, time-sensitive portions of a program must be handled differently in checkpointing systems by careful checkpoint placement or time management.

Timely branching. If a checkpoint is placed in a line of code before a time stamp is gathered, and that timestamp is used in a predicate statement, execution can execute both branches if the timestamp elapses. This is shown in Fig. 2.4.

Time and Data Misalignment. Often in embedded code, a timestamp is gathered every time sensor data or an event happens. If a checkpoint is
Figure 2.3: Porting legacy embedded-C programs to a task graph is not trivial. On the left an activity recognition program is shown, on the right is one possible port to a task-graph, with timing constraints defined. The final task graph does not always match the original program as tasks must be small enough to allow forward progress; reducing readability, and increasing graph complexity. Timing constraints, control flow, and data flow must also be determined.

placed between the timestamp and the data gathering, the timestamp will be inaccurate. After a power failure recovery at that checkpoint, new data will be gathered associated with an old timestamp, causing incorrect execution of the program. This is shown in Fig. 2.4b.

Data Expiration. Data gathered in one power cycle may not be fresh enough for usage in a later power cycle. This was shown in Mayfly [19], a task based programming language, but has not been handled by any automatic checkpointing systems to date. This error is shown in Fig. 2.4c.
Figure 2.4: **Time based consistency violations encountered with automatic checkpointing schemes are shown.** These violations occur because of incorrect execution caused by bad checkpoint placement, leading to an execution that is not possible on a continuously powered device. We focus on time based violations that have not been previously explored in automatic checkpointing systems.

Previous automatic checkpointing works Ratchet [50] and Chinchilla [34] have ignored these consistency violations caused by time.
Chapter 3

Related Work

Intermittent power failures complicate development of efficient, useful, and correct programs. While disruption tolerance and resilience has been a core area of computer science research for decades, frequent loss of power due to energy harvesting has only recently been considered. In Table 3.1 key characteristics of VISP are compared to those of Mayfly [19], Alpaca [33], Ratchet [50], Chinchilla [34] and InK [52]. Different programming models have different characteristics; in this section we compare some of these characteristics from the state of the art to VISP.

3.1 Automatic Checkpointing Systems

Checkpoint based systems do not require custom programming models, relieving the programmer from the cumbersome task of energy aware programming. Systems that automatically determine checkpoint placement at compile-time like Ratchet [50] HarvOS [5], and Chinchilla [34] are most closely related to this work. HarvOS parses the control flow graph of a program and instruments with energy-aware checkpoints, requiring a small amount of programmer intervention to place effectively. Ratchet functions by placing checkpoints at the boundaries of idempotent sequences of instructions. Chincilla over-instruments programs with checkpoints by storing some variables in non-volatile memory, and disabling/enabling checkpoints heuristically. Chinchilla uses a non-volatile stack for register spills and return addresses that is unconstrained and double-buffered. Even though Chinchilla aims to leverage the availability of volatile memory present within existing systems targeted at intermittent computing, it’s only used for 1%-4% of all variables [34]. Other data is promoted to global data, including the local variables of each function. This means the required stack memory is the sum of the stack frame of each function in the program. VISP only requires the working stack to be double buffered within non-volatile memory, and does not rely on variable promotion. Unlike VISP, each of these systems
Table 3.1: Comparison of the current state of the art in terms of programming models for intermittently powered devices. In particular, prior systems are not scalable: task-based systems [19, 33, 52] lead to task explosion for ordinary C programs; Ratchet [50] has checkpoint explosion for especially programs including pointers; Chinchilla [34] has global variable explosion since it transforms each local variable into a global variables.

3.2 Other Checkpointing Systems

Mementos [42] was the first checkpointing scheme, using intermittent voltage checks to decide when to save state. QuickRecall [22] and Hibernus [3] extended this work with newer non-volatile memories. DINO [30] laid out the memory consistency problems that will arise with intermittent computing for mixed-volatility processors. VISP builds on these early techniques, however, these systems did not consider the effects of time on the consistency of execution, nor did they provide a way to automatically instrument a program with checkpoints and provide worst-case checkpoint times even when volatile main memory is used.

3.3 Task-Based Programming Models

Task based systems provide a different way to program and intermittent computer. Alpaca [33] and related works [8] focus on providing control flow and data flow mechanisms while reducing the memory footprint from multi-versioning. Mayfly [19] provides explicit semantics for specifying timing constraints on sensor data in a task based language. InK [52] provides a way to handle events and interrupts from clock sources, sensors, and energy in the environment, despite power failures. Task based systems require a custom programming model, which leads to added programmer intervention and complexity. Task decomposition is a manual process that is error prone
and not resilient to changes in the availability of energy in the deployed environment.

3.4 Compiler-Based Systems

Compiler-based systems aim to create idempotent sections [11, 50] with regard to non-volatile memory accesses on an instruction level. Compiler passes are used to optimize the interaction with non-volatile memory to create these sections. After each idempotent section a checkpoint is performed to avoid write-after-read (Section 2.1.1) violations. Difficulties with this approach arise due to a limited overlap between the intermediate representation (IR) of the compiler and the target instruction set architecture (ISA) [50]. Current compilers often have limited support for feedback between the IR optimization passes and the code generation, complicating the idempotent region creation. As with task based systems, the task length directly dictates the minimal required on-time to guarantee forward progress. But due to the automatic decomposition, and the smaller size of the idempotent regions (tasks), this is less of an issue. A downside however is the need to recreate and tune the algorithm creating the idempotent regions for different architectures.

3.5 Non-volatile Processors Architectures

Integration of non-volatile components; e.g. non-volatile registers, to the processor architecture provides mostly automatic management of forward progress and memory consistency [31, 32]—eliminating the need for handling these properties explicitly by the programmer. However, non-volatile architectures consume more power, they have increased area and decreased frequency as compared to general purpose volatile processors with SRAM-based flip-flops [21]. Our focus is on off-the-shelf processors with hybrid volatile and non-volatile memory on the market, like the Texas Instruments MSP430FR5994 [48].
Chapter 4

Design

VISP consists of a runtime combined with code instrumentation for the C language—Fig. 4.1 presents the logical flow and the main components of the VISP system. Developing batteryless applications is a tedious job; legacy (C) codes/libraries cannot be re-used and must be re-implemented in such a way that the program logic and the intermittency handling are intermingled. The main motivation behind the design and development of VISP is to provide the view of a continuously-powered system to the programmer—so that legacy C code can be run without any modification to the program source. Additionally, VISP allows programmers to annotate their C files in order to define timing constraints that were implicit (or explicit) in the original program. VISP allows programmer to focus only on the correct and timely execution of the application, eliminating the explicit need for intermittency handling, and requiring few modifications to their original C program.

4.1 Design Considerations

Before delving into the design and implementation details of VISP and its components, we would like to elaborate on the challenges and alternative designs and their trade-offs.

4.1.1 Task-Based Transformation Versus Checkpointing

Automatic transformation of a C program into a task-based program requires significant work (as discussed in Chapter 2). First, the determination of task boundaries so that none of the tasks exceeds the device’s energy buffer is challenging. Second, the set of variables that are alive across task boundaries; i.e. the task-shared variables, should be identified and versioned; e.g. through channel abstractions, in non-volatile memory. This might lead to a large non-volatile memory burden as well as execution inefficiency due to read/write operations on a multitude of versions of these variables—i.e.
Figure 4.1: VISP system overview. A runtime combined with code instrumentation for the C language ensures memory consistency via virtual memory and stack segmentation; the progress of computation via checkpointing; and timely execution via handling time annotations.
the identification of optimal task boundaries that minimizes the task-shared variables is challenging. Last, task-based programming models are strict in their memory model and they do not allow dynamic manipulation of arbitrary memory locations by pointers at run-time—C programs with pointers cannot be transformed into any task-based programming model. Therefore, instead of task-based transformation, VISP uses checkpointing in order to support any kind of C program source.

4.1.2 Checkpointing Scalability

As the amount of checkpointed state grows, the checkpoint overhead increases—potentially leading to overheads that may exceed the device’s capabilities or memory capacity. To remedy this issue, VISP segments the stack and keeps only the segment which is often accessed in volatile memory—the rest of the stack is kept in non-volatile memory. By segmenting the stack VISP can provide a fixed worst case checkpoint time, as the variable stack size is fixed to the size of a stack segment.

4.1.3 Pointer-Handling and Memory Consistency

Since pointer access cannot be determined at compile time, existing systems need to checkpoint the whole .data and .bss sections as well as the whole stack in order to keep memory consistent—again leading to increasing checkpointing overhead and in turn non-scalable worst-case system overhead. On the contrary, VISP implements a virtual memory scheme to handle pointers and ensure memory consistency—VISP keeps track of only manipulated non-volatile memory locations at runtime and keeps their original values in a non-volatile undo log. If a checkpoint has not been performed upon a power failure, VISP restores the original contents of the non-volatile memory using the undo log—ensuring the memory consistency.

4.1.4 Double-Buffering Memory Requirements

Checkpointing the device’s volatile state requires an atomic two-phase commit operation to ensure its consistency: in the first phase the checkpointed data is copied to a temporary buffer in non-volatile memory; then in the second phase the buffered data is committed to the original location, or a pointer marking the active buffer is switched (a ping-pong scheme). To this end, existing checkpointing systems often double buffer the stack, bss and data sections—their memory requirements increase with the volatile state. On the other hand, VISP only requires the active stack segment and the modified memory locations in bss and data sections to be double buffered—significantly reducing the non-volatile memory requirement.
4.1.5 Timely Execution

Languages like C do not provide any keyword/statement to express time constraints of the data and handle it—programmers manually need to timestamp data and keep track of data expiration to discard data and skip consumption of outdated data. This complicates the program code as well as it is a tedious process and might lead to bugs due to manual timing and expiration checks, control flow delivery and recovery due to data expiration—as summarized in Section 2.2.1. VISP provides annotations to relate data and time and all underlying time management is performed at run-time without programmer intervention. Moreover, VISP provides special statements to change control flow and perform recovery upon data expiration.

4.2 Scalable Checkpoints

There are several existing works, e.g. [50, 21, 5] that exploit architectural support, and ensure constant and scalable checkpointing overhead. As an example, Ratchet [50] uses non-volatile memory as the main memory so that stack and global data are already persistent—leading to constant checkpoint time since only the volatile state, i.e. registers, of the processor is checkpointed. This requires decomposing programs into idempotent code sections via the compiler using static analysis at the instruction level and gluing them together with checkpoints. However, dynamic memory manipulations that cannot be determined at compile time; e.g. writes via pointers, require a checkpoint after each instruction—leading to a considerable checkpointing frequency and in turn overhead.

On the contrary, VISP targets devices that include a mixture of volatile and non-volatile memory: the working stack segment is maintained in volatile memory and non-volatile memory is used for global data (i.e. data and bss), the other stack segments and for checkpointing purposes. In general, in order to keep the volatile state and the checkpointed state consistent with each other despite power failures, the volatile stack should be double buffered in non-volatile memory and checkpointed using a ping-pong buffer scheme. However, this is not scalable since checkpointing overhead grows with the size of the volatile stack. Moreover, recovery time; i.e. restoring the volatile state after a power failure, is not fixed and might exceed device’s energy budget. VISP remedies this problem with stack segmentation.

4.2.1 Checkpointing Time

Current systems leveraging benefits of volatile memory in terms of power consumption such as MementOS [42] can not provide a worst case checkpoint time when using automatic checkpoints (Section 3.1) due to the stack size changes being uncorrelated with time. VISP can make decisions regarding
when to checkpoint not only based on triggers such as time or an interrupt generated by an comparator at a set voltage level, but also when a critical amount of stack memory is consumed. Thereby limiting the maximum checkpoint time to a checkpoint containing this critical amount of stack memory and constant checkpoint data such as saving the register content and performing the checkpoint logic.

Systems such as MementOS or HarvOS require either static analysis or profiling combined with either manually placed, or deterministically placed dynamic checkpoints to guarantee forward progress even if a minimal energy buffer guarantee is provided. This is due to the otherwise unknown execution flow, and therefore stack usage of the program being executed.

### 4.2.2 Stack Segmentation

The stack allocation within the execution of the applications might vary significantly when the system enters or returns from a function that allocates many space on the stack. In order to explore this phenomenon, we ran an activity recognition application used in benchmarking of MayFly, InK, and Alpaca and recorded the stack usage on of this benchmarks on MSP430FR5994 target platform. As can be seen in Fig. 4.2, the stack size is not constant and varies significantly during program execution. As a consequence, it is not possible to guarantee a worst case checkpoint size since the stack size depends on dynamic program flow and it can be unknown at compile time. Differential checkpoints reduce the amount of stack memory to be checkpointed—but a differential checkpoint can still be unpredictably large if a checkpoint is taken after a chain of functions calls.
VISP journals only the working stack and the registers during a checkpoint. This ensures scalability and deterministic worst-case overhead. Previously checkpointed segments belonging to the lower parts of the main stack are maintained in a segment pool. The global variable and pointer access are handled by the virtual memory which performs necessary undo logging in order to keep memory consistent.

VISP segments the stack into blocks of predetermined size. The fixed size of these segments is determined by the programmer at compile time considering the maximum stack requirement among the functions. VISP maintains the stack segment which is actively used as stack in volatile memory, henceforth referred to as the **working stack**. This enables a fixed worst-case checkpoint overhead since only the working stack should be double buffered and saved in non-volatile memory. Moreover, recovering from a power failure only requires the working stack to be restored, instead of restoring the whole stack.
4.2.3 Segment-Swap Driven Checkpoints

Checkpoints in VISP contain all necessary data to capture the volatile state of the system—the registers of the processor and only the working stack that stays in volatile memory. Swapping stack segments from non-volatile memory into the volatile working stack can be seen as a special type checkpoint, henceforth referred to as a segment checkpoint—stack memory consumption dictates when to perform a segment checkpoint. When a function is entered, the stack pointer is adjusted so that the stack memory required to execute the function is allocated. VISP inserts a check before the modification of the stack pointer to determine whether there is enough space in the working stack to execute the function. When enough space is in the working stack, the execution resumes and the function interfaces with the working stack without any additional steps. If there is not enough space left on the working stack, a segment checkpoint is performed so that the contents of the working stack is saved in non-volatile memory. Then, the working stack is cleared so that there is enough memory for the function to execute.

When a function that triggered a segment checkpoint returns, it is also required to restore the previous contents of the working stack from non-volatile memory since it holds data relating to the caller function. When performing a segment checkpoint, a new stack segment is allocated from a segment pool in non-volatile memory—see Fig. 4.3. Next, the working stack is copied to this stack segment and a link to this segment is placed on the working stack. This way a linked list of stack segments that represents the whole stack of the program is maintained in non-volatile memory. Restoration of a segment occurs when the stack shrinks: the segment pointed to by the link in the current working stack is copied to the working stack. Upon reboot, the recovery mechanism restores the checkpointed working stack and the registers only to ensure forward progress.

4.3 Non-volatile Memory Consistency

As depicted in Fig. 4.3, VISP maintains the working stack in volatile memory, but the global variables; i.e. data and bss sections, and the rest of the stack segments are maintained in non-volatile memory. To keep the non-volatile memory consistent with the checkpoint of the volatile data, VISP provides a virtual memory interface to the applications for non-volatile memory write operations. Intermittent execution might create inconsistencies if the application modifies non-volatile memory directly and the modified locations are not versioned; i.e. double buffered [41, 30]. VISP instruments non-volatile memory write operations and enables on demand versioning: undo logging is employed so that if any memory location outside of the working stack has been modified, the original version is saved in an undo log. After a successful checkpoint the undo log is cleared. Upon a power failure, the contents of
the undo log is written back to the original locations, thereby restoring the consistency between the non-volatile memory and the checkpoint states.

### 4.3.1 Pointer Handling

Existing systems; e.g. [5, 3], insert a checkpoint at compile time to journal the whole volatile memory upon pointer manipulations since the modified locations cannot be determined statically. On the contrary, in VISP, pointer writes to global variables within the data and bss sections, or a stack segment in non-volatile memory are managed by the virtual memory at runtime. In particular, pointers to the stack can manipulate memory locations in non-volatile segment checkpoints as well as in volatile working stack—Fig. 4.3. These pointers are handled by converting the physical addresses of data to virtual addresses. These virtual addresses contain a logical segment index and the offset in the corresponding stack segment. If the virtual address points to the working stack, then the data can be directly modified since working stack is checkpointed separately. On the other hand, if it points to a segment already checkpointed in non-volatile memory, the virtual memory employs undo logging to keep the modified memory locations consistent.

### 4.4 Timely Execution

VISP provides annotations to denote the expiration constraints of the data and necessary keywords for checking if time constraints are met—see Fig. 4.1. A timestamp value is associated with each programmer annotated variable and the write operations on these variables are instrumented by the compiler. VISP updates the value of the timestamp when specified by the application developer using a persistent timekeeper; which keeps track of time across power failures [19]. Programmers can check the expiration of the data using expired keyword—VISP compares the current time with the timestamp to identify if programmer-defined timing constraints are met. It is the responsibility of the programmer to provide necessary program logic within these syntactic structures that handles data expiration.
Chapter 5

Implementation

The current implementation of VISP targets the MSP430FR5994 [48] MCU that has 256 KB non-volatile (FRAM) and 8 KB volatile (SRAM) memory. A compiler back-end addition is introduced so that the VISP runtime can handle stack segmentation. Code instrumentation—implemented using the LLVM utility library LibTooling [1] which is intended for code transformations—is used to support timely execution and the consistency of non-volatile memory via virtual memory. We employed code transformation rather than complete compiler support, to allow for portability—enabling the use of multiple compilers, and in turn eliminating the need for re-implementing the instrumentation. In order to produce the target binary, we used MSP430-GCC version 7 [46].

5.1 Stack Segmentation Instrumentation

In VISP, the stack segmentation and restoration is employed at function entries and exits respectively. Before the stack is allowed to grow or shrink, a check must be inserted which determines if the operation is allowed—see Fig. [5.1]. Since the stack frame of a particular function is known at compile time, this check can determine if there is enough space in the working stack to continue operation without a segment checkpoint. If there is not enough stack space, the working stack is swapped; i.e. a segment checkpoint is performed: a stack segment in the segment pool is allocated, the contents of the working stack is copied to this segment in non-volatile memory, the working stack is cleared and a link to the stack segment in non-volatile memory is created. Since the arguments of the function (if any) remain in the checkpointed segment, these arguments are copied from this segment to the—now empty—working stack. To allow for these operations, we modified the compiler back-end to insert the required stack space check and argument copying operations. It is worth mentioning again that the size of a stack segment is determined at compile time and its minimum size is dependent
int foo(int x){
  [requires 128 bytes]
  if (remaining_stack<128)
    swap_out_working_stack();
  char bar[128];
  x = foobar(x, bar);
  if (swapped_out)
    swap_in_working_stack();
  return x;
}

Figure 5.1: **Stack segmentation and checkpointing during execution.**
Light gray and red represents the code inserted during the compiler pass, the red lines only execute when the available stack on the working stack is insufficient for the current function. `swap_in_working_stack()` is only called when the corresponding swap out call was also performed. These are achieved using stack manipulation.
on the minimal stack requirements of the functions in the source.

5.2 Atomic Checkpoint Commit

Due to the unpredictability of system failures, it has to be taken into account that the system can run out of power during a checkpoint procedure. This could lead to a partial checkpoint which has the possibility to incorrectly restore the system state. For this reason all components that compose the full checkpoint, such as the working stack checkpoint, register checkpoint and the virtual memory management are synchronised using a single atomic flag. If the system fails during a checkpoint it must still recover correctly to the last successful checkpoint. Therefore all checkpoint related data is double buffered and the buffer selection is dependent on the singular atomic flag. All components containing checkpoint related data that are discussed next use the atomic flag and double buffering where required. This is not specifically stated when the components are discussed as this would convolute their specification.

5.3 Virtual Memory Instrumentation

In order to implement undo logging so that the changes on non-volatile memory locations can be undone, non-volatile writes and pointers are instrumented. The virtual memory internally uses an interval tree data structure to keep track of the address range present in the virtual memory. As the virtual memory is based on an undo logging approach, performing a checkpoint will clear the interval tree, as at this point all modification made to the non-volatile memory are committed. When data already present in the interval tree is requested to be logged, the virtual memory ignores the request as the data is already logged. When a partially overlapping data region is requested the virtual memory will trigger a checkpoint and after this continue servicing the logging request, now with an empty interval tree. When a checkpoint has to be restored, the data in the virtual memory is copied back and the interval tree is cleared. To this end, write accesses to global variables and to non-volatile memory cells through pointers are instrumented—pointers can point to not only global data but also data on the working stack or data on a segment checkpoint in non-volatile memory. In order to decide if the memory writes should be logged or not, the instrumented code invokes virtual memory to check if the destination is in non-volatile memory. The virtual memory controls this by checking if the physical address is in non-volatile memory. If so, the virtual memory logs the contents of the memory cell in undo log. These operations are implemented fully in software since microcontrollers, such as the one we target [48], do not include a dedicated memory management unit due to the energy cost it imposes.
Figure 5.2: Virtual address composition.  
A pointer referring to a location within a stack segment is converted to a virtual address to allow for the segment location to move in memory.

5.4 Virtual Addressing Instrumentation

The working stack segment is always used when executing code and whenever more stack memory is required it is copied to a non-volatile stack segment. This means that the location of data in the memory can and will change during execution. One of our contributions is to allow for pointer usage, therefore we need to virtualize addresses that point to data on the stack. This is done using source instrumentation passing pointers through a virtual address translation subsystem. The virtual address consists of a 1-bit marker signalling whether this is a virtual address or a physical address followed by a virtual index of the stack segment holding the data and the actual offset within the stack segment as can be seen in Fig. 5.2. This virtual index does not directly indicate the index of the stack segment, as this can change during execution. In addition to transforming the addresses the virtual addressing subsystem also tracks what stack segment is connected to what virtual index. This virtualisation method does lead to one less bit that can be used for addresses as the most significant bit of the address holds the marker, but currently there are no MPS430 variations requiring all 20-bits of address space. The address translation is done fully in software and redirects pointer dereferences data on a stack segment which has already been copied to non-volatile memory to its correct location, as this eventual destination in non-volatile memory is unknown during the time the address of data on the stack is taken.

5.5 Checkpoint Instrumentation

Current implementation of VISP provides two checkpointing strategies. First, VISP allows programmer to manually place checkpoints. Another approach is timer-driven checkpointing; where the runtime interrupts program execution and checkpoints the system state periodically at a given frequency. Apart from these strategies, hardware-assisted checkpointing; e.g. [3], could also be employed: using the ADC to measure the remaining energy to decide on a checkpoint. Since the size of working stack is fixed—leading to a fixed checkpoint time—it is possible to set an accurate trigger voltage level to perform the checkpoint.
5.6 Instrumentation for Time Annotations

Each write to a time-annotated variable is instrumented so that the timestamp value associated with the variable is updated. VISP with time sensitive programs requires the ability to measure time across power outages using either a remanence-based timing technique [20, 40] or using a RTC supplied by a small capacitor [18]—persistent timekeeping is mandatory to update timestamps and in turn to handle time annotated source files. Time-annotated variables are handled by internally connecting them to timing information. this is done using source code instrumentation. Whenever a time-annotated variable is used it is automatically check if its validity has expired. Resetting the expiration time is done explicitly by the programmer, as it is not known whether a write should reset the expiration time or not.
Chapter 6

Evaluation

We investigate the execution overhead of VISP for various applications, comparing to the state-of-the-art intermittent runtimes. We then demonstrate how VISP enables porting of arbitrary C programs, including TinyOS programs, to work under intermittent execution. We also show results from a user study we conducted comparing VISP to task based programming. We found that VISP has comparable overhead to state-of-the-art runtimes, while providing a complete set of features available to regular C programmer. Also, VISP (for the first time) demonstrates successful execution of legacy code for sensor networks into intermittently-powered domain.

6.1 Application Benchmarks

We have compared VISP against three task-based systems: InK, Mayfly and Alpaca. In addition, we compared VISP against checkpoint-based runtime MementOS that journals the complete stack and all global variables (using our implementation) and plain C versions for completeness. As benchmarks we chose three representative applications, used earlier by most studies on systems for intermittently-powered devices: (i) bitcount (BC), (ii) Cuckoo filter (Cuckoo) and (iii) Activity Recognition (AR). BC implements bit counting in a random string with seven different methods, later cross-verifying for correctness; Cuckoo implements cuckoo filter over a set of pseudo-random numbers, then performing sequence recovery using the same filter; while AR implements physical activity recognition based on machine learning with locally stored accelerometer data. It should be noted that Cuckoo filter cannot be implemented in Mayfly since loops are not allowed in a Mayfly task graph. The experiments were conducted using a continuously-powered TI MSP-EXPFR5994 evaluation board [48]. We observed the execution time, code size and memory requirements of the aforementioned benchmark applications.
Figure 6.1: **Benchmark performance.** All benchmarks were operating on continuous power for fair comparison. **$S1$–$S3$:** configurations with different working stack sizes imposing a different checkpoint frequency and checkpoint time. **$S1^*$–$S3^*$:** the same stack configurations as **$S1$–$S3$** but with an additional timer checkpointing every 10 ms if there was no checkpoint due too the working stack. Working stack sizes—For Cuckoo: $S1=100$ B, $S2=150$ B, $S3=200$ B. For AR: $S1=100$ B, $S2=200$ B, $S3=220$ B, For BC: $S1=80$ B, $S2=90$ B, $S3=100$ B. **$ST$** is **$S3$** with checkpoints at the task boundaries.
Table 6.1: The memory consumption (in B) for three benchmark applications written in InK, MementOS and VISP.

<table>
<thead>
<tr>
<th></th>
<th>InK</th>
<th>MementOS</th>
<th>VISP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>3442</td>
<td>4459</td>
<td>4212</td>
</tr>
<tr>
<td>BC</td>
<td>2922</td>
<td>4433</td>
<td>10207</td>
</tr>
<tr>
<td>CF</td>
<td>2648</td>
<td>4693</td>
<td>4210</td>
</tr>
</tbody>
</table>

6.1.1 Results

Results pertaining to execution overhead are presented in Fig. 6.1. The plots on the left present the execution time overhead of VISP as well as the number of checkpoints performed with respect to different working stack sizes. As working stack size gets bigger, the number of segment-swap driven checkpoints decreases since on-demand stack requirement of the applications are fulfilled—observe that configuration S3 did not lead to any stack segment swap and in turn checkpoint and S1 led to considerable number of segment swaps and checkpoints. On the other hand, increasing the working stack size also increases the overhead of a single checkpoint since the journaled data is bigger—there will be always a trade off. Among benchmarking applications, AR led to considerable amount of segment-swap driven checkpoints with configuration S1 due to its varying stack size requirements. Apart from segment-swap driven checkpoints, we also enabled a timer-driven checkpoints with a frequency of 10 ms that ensure the forward progress—configurations S1*-S3* indicate the configurations S2 and S3 with timer-driven checkpoints enabled. VISP checkpoints do not introduce significant overhead since only the working stack and registers are journaled.

We selected configurations S2* and S3* to assess the execution time performance of VISP considering the aforementioned runtimes—see the right hand-side of Fig. 6.1. For the fairness of comparison against task-based systems, apart from timer-driven checkpoints in S2* and S3*, we placed checkpoints to configuration S3 at the task-boundaries for VISP (shown as ST in the figure) and our MementOS implementations—a representative checkpointing-based runtime. We observed that selecting a reasonable working stack size, VISP reaches almost the performance of the existing task-based systems.

Table 6.1 presents a comparison of memory overhead of the benchmarking applications implemented in InK (task-based system), MementOS (checkpoint-based system) and VISP. The .data section overhead of VISP depends on the size of the pre-allocated stack segment pool (which was 8196 B) and undo log (which was 4096 B). The code size in selected applications is dependent on not only the application source but also on the stack segmentation and virtual memory implementations in VISP. Overall, the increase in the memory and
Table 6.2: Results of running a real-world TinyOS program with VISP on intermittent power. Experiment executed on MSP430FR5994 [48] with artificially generated power intermittency traces. Runs without checkpoints complete more often but run the risk of consistency errors; CP: checkpoint.

<table>
<thead>
<tr>
<th>Duty Cycle</th>
<th>'Greenhouse monitoring' application tasks</th>
<th>Sense Moisture</th>
<th>Sense Temp.</th>
<th>Compute</th>
<th>Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o CP (plain C)</td>
<td>4%</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>w/ CP (plain C)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>w/o CP (TinyOS)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>w/ CP (TinyOS)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>w/o CP (plain C)</td>
<td>48%</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>20</td>
</tr>
<tr>
<td>w/ CP (plain C)</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>w/o CP (TinyOS)</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>w/ CP (TinyOS)</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>w/o CP (plain C)</td>
<td>100%</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>w/ CP (plain C)</td>
<td>45</td>
<td>45</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>w/o CP (TinyOS)</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>w/ CP (TinyOS)</td>
<td>44</td>
<td>44</td>
<td>44</td>
<td>44</td>
<td>44</td>
</tr>
</tbody>
</table>

code size overhead of VISP is reasonable as compared to state-of-the-art.

6.2 Running Unmodified TinyOS Code

To prove the claim that VISP enables automatic porting of existing/legacy C code for non-intermittently powered systems, we instrument and unmodified TinyOS program for Greenhouse Monitoring (GHM). GHM executes in an infinite loop sense moisture of soil, sense temperature of ambient, Compute measurement averages and Send over a wireless interface). We compare Plain-C and TinyOS [28] versions of GHM with and without VISP instrumented checkpoints. Both apps were executed on MSP430FR5994 [48] evaluation board with artificially generated power intermittency traces, i.e. MCU was brought to a hardware reset following a pre-programmed pattern. We compare the results of executing the Plain-C and TinyOS versions of GHM in Table 6.2 for varying levels of intermittency. VISP provides constant, but small, overhead for any task of the application. This demonstrates that VISP can run semi-sophisticated legacy TinyOS programs without any programmer porting required.

There are limitations to this experiment however. Handling events is core to TinyOS function, and we did not explore porting event handling mechanisms in a robust way, nor did we experiment with larger, more sophisticated programs. This demonstration is a first step towards portability.
6.3 User Study and Developer Effort

We have also measured developer effort of using VISP. For this we have designed a large online user study. The goal was to objectively assess the time it takes to design an application in VISP compared to the state of the art task-based programming language. The study was approved by the Human Research Ethics Committee of this research team’s university. The study, conducted among almost 100 participants suggests that it is easier to code applications for intermittently powered systems with VISP than with task-based languages. In all tested cases it was harder and slower to debug a task-based program compared to VISP. Our study is unique considering legacy systems for intermittently-powered devices targeting ease of programming, e.g. [21, 42]—none of them support that statement with any end user assessment.

6.3.1 Methodology

At the beginning of an online survey each participant was given an introduction to intermittent execution and to VISP and InK [52]. Then, we have then asked participants to find bugs in three simple programs: (i) swap of two variables (with no use of a temporary variable), (ii) bubble sort, and (iii) program that considers variable expiration based on time. Each of three programs was written separately in VISP and in InK and had exactly the same type of bug, at exactly one line of the program. Users were asked to point to a line which contained that bug and specify what the correct statement should be. Our study is consistent with modern methods of testing programming language comprehensibility, as described e.g. in [37].

Each program with a bug was presented to a user on a separate page. Additionally, time spent on finding a bug in each of the program was measured indirectly by measuring time user spent on a page with a bug and its related question. This was possible as the users could access next page with a new buggy program only after giving the answer to the current question. No corrections of the given answers were possible once the answer was submitted. We have randomized order in which each program appeared at respondent’s screen in order to remove presentation bias against one language and objectify bug finding time (as time to find a bug in a program written in one language is faster once the bug has been found in a program written in an other language). Correct answers were revealed to users only after completion of the questionnaire. Overall, we believe our method of measuring developer effort was the most objective as we have eliminated human in assessing the correctness of programs submitted by the survey participants, contrary to methodology used in [52].

We have also asked each participant to subjectively assess VISP and InK based on the online introduction and the programs they assessed. The ques-
tions targeted difficulty, intuitiveness, completeness, flexibility, conciseness, usability and speed of development to run sensing applications, and ease of understanding among others. Users were selecting answers from a five-level Linkert-type scale, ranging from strongly agree to strongly disagree.

6.3.2 User Pool

Link to a survey was sent via various social media channels of the authors and via personal invitations. Additionally, we crowdsourced data collection to MTurk service of Amazon to increase and diversify the user pool. This way we managed to obtain a very diverse crowd of users with varying experience and programming background.

At the time of writing this paper a total of 90 responses were collected (with 36 from MTurk). Majority of respondents (89%) were men, with an average age of 30 years (the youngest and oldest participant was, 18 and 57 years old, respectively). Three most popular programming languages of choice of all participants were Java, C and Python. 78% of all respondents had at least two years of programming experience. Almost 83% of respondents had average or below average knowledge of embedded systems powered by energy harvesting technologies. Participants were located at 12 different countries spread across Asia, Europe and North America.

6.3.3 Result

Core results of the study are given in Fig. 6.2. We immediately observe that in all program cases it was (i) harder to find a bug and (ii) users were more prone to error when exposed to a task-based language. Statistically, Wilcoxon T Test on all programs’ bug search time rejected the hypothesis that VISP/InK results were the same with p-value below 0.001. In other words VISP is a more user-friendly system than a task-based one.

As the complexity of a program increased users had a difficulty of finding a bug in an InK program (for Bubble Sort in half of the cases users were wrong). We note that we have tested user study results for both MTurk and non-MTurk respondents and overall conclusions on VISP were holding irrespective of the user pool type.

Regarding subjective evaluation of VISP against InK, participants considered VISP to be more intuitive, easier and conciser than InK. In none of the categories we asked respondents considered VISP to be superior to InK. For instance, almost 70% of respondents would use VISP to “teach a new student how to program Intermittently Powered Systems”.

Of course, we are aware of the limitations of this survey (e.g. more programs with bugs should have been introduced, users were supposed to

\footnote{We note that use of crowdsourcing to increase user participation is a common practice in software engineering research [45], [24].}
Figure 6.2: **VISP user study results.** For all three test programs, *Swap*, *Bubble* and *Timekeeping*, users found that it is easier with VISP to identify a bug and were more accurate in correcting the VISP program than that of InK [52]. Whiskers in the right-hand side figure denote standard deviation.

be asked to write a simple program themselves). However, all additional complexities would result in far lower response rate. Therefore, we claim that the way we constructed our survey was the best choice considering *response rate to study completeness* trade-off.
Chapter 7

Discussion

The field of devices powered by energy harvesting—resulting in intermittent execution—is still in a fundamental stage of development. Researchers are still hard at work to come up with and develop a not only functional but also practical solution for the problem of intermittent computation and the interaction with the outside world. This work aims to bring the field a step closer to the eventual goal of being able to easily and efficiently write applications for intermittently powered devices. To aid future development towards this goal, the limitations of the system are briefly discussed.

As shown in Fig. 6.1 VISP provides a performance competitive with that of standard checkpointing systems. And does not significantly trail behind the performance figures generated by task based systems. More comparisons are however needed to properly judge the effectiveness of the VISP system. Not only to other approaches such as Ratchet and Chinchilla which was published while VISP was already in development. But also compared to a version of VISP where all system memory resides within non-volatile memory. This—while still requiring the virtual memory—could eliminate the need for the virtual addressing structure. However the non-volatile memory consumes more energy while reading and writing compared to its volatile counterpart. To effectively compare these scenarios this version of VISP will have to be realized. The code transformation should also be improved so that it’s able to fully support instrumentation of all constructs available in C related to global memory and pointer manipulation. Most of these constructs are already supported, but additional tests and edge cases need to be developed and implemented to make the instrumentation fully automatic in all cases.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this theses we addressed the problem of guaranteeing forward progress and maintaining correctness during intermittent execution. This transition towards batteryless sensors results in a significant reduction in size, cost, weight, and lifetime maintenance. We developed VISP, an automatic checkpointing system that enables correct execution of uninstrumented programs on intermittent power, and provides semantics for easily porting time-sensitive programs to the intermittent domain while maintaining correctness. VISP enables full use of C features like pointers and dynamic memory management through a miniature virtualization technique. Guarantees on worst case checkpointing time are provided using a novel stack segmentation technique, ensuring VISP scales as applications become more sophisticated—while using volatile memory in combination with the main non-volatile memory. Support for time sensitive data is provided by means of variable annotations, allowing for more powerful and convenient decision-making with regard to the age of data. We evaluated VISP against other checkpointing and task based approaches, showing reasonable overhead nearly matching the performance of task-based systems in some configurations. We conducted a large user study, where participants found VISP more intuitive than the task based approach.

8.2 Future Work

The field of intermittent computation is in it’s infancy and VISP strives to be a fundamental component of the advancement within the field. VISP is the first step in providing a method for automatic porting of legacy code to the intermittent domain, potentially saving significant developer time and
capital. There are multiple possible branches of research directly related to VISP.

As of now VISP decides when to perform a segment checkpoint (Section 4.2.3) fully dynamically. This can lead to scenarios where an application can continuously create segment checkpoints and restore them on every function entry and exit respectively. This happens when the stack size is balancing on the edge of the allowed size or when a function which triggers a segment checkpoint is repeatedly called in for example a loop. Static analysis can be employed to create predictions regarding the stack usage at different points in the program and this information could be used to add a form of intelligence to the dynamic segment checkpoint. This does not undermine the criticism raised earlier against systems which rely on static analysis or profiling, as they use information gathered as the sole deciding factor with regard to checkpoint placement. Additionally virtualizing the I/O interface, making it restorable across power failures is an essential step required to fully support both interaction with the outside world and complete support for systems such as TinyOS.

The addition of dedicated hardware features to assist with both the virtualized memory management and address translation would greatly decrease overhead for VISP. Current FRAM based non-volatile memory microcontrollers already include memory protection units as the non-volatile memory is also used to store the firmware. Extending this memory protection unit to more closely resemble the functionality of a memory management unit—often found on more powerful devices—would be a logical step towards better support for intermittent applications.
Bibliography


[51] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In Proc. OSDI, Savannah, GA, USA, Nov. 2–4, 2016. ACM.