Incremental, Near Optimal Checkpointing for Intermittent Computations

by Hassan Ali Khan

Lahore University of Management Sciences

MS THESIS

Incremental, Near Optimal Checkpointing for Intermittent Computations

Hassan Ali Khan

Supervised By

Dr. Hamad Alizai
Dr. Junaid Haroon

May 27, 2017

List of Figures

List of Tables

Contents

Chapter 1

Acknowledgement

First of all, I would like to thank Dr.Hamad Alizai and Dr.Junaid Haroon for giving me the opportunity of working under their supervision. It would not have been possible without their continuous guidance and support. I am also grateful to my parent for their kind support in every step of life. At last, I would like to thank my colleagues for the motivations and advises, especially abubakar and saad ahmed for their significant contributions regarding hardware integration and static code analysis in this project respectively as the scope of work was more than an individual thesis.

Chapter 2

ABSTRACT

We propose incremental checkpointing for transiently powered embedded devices to retain their computational state across multiple activation cycles. Recent comparable solutions are sub-optimal due to both software and hardware limitations.
(i) Such solutions lack the ability to determine which RAM locations have been modified since the last checkpoint, resulting in redundant checkpointing overhead as unmodified locations are rewritten in persistent storage, and
(ii) they are hampered by the semantics of flash based persistent storage, which require erasing complete sectors before overwriting even a single byte, rendering any effort to minimize the size of updates useless.

Our proposed solution addresses both of these limitations through a hardware-software co-optimization. Our incremental checkpointing approach *Inch* instruments the source code to record all modified RAM locations since the last available checkpoint on persistent storage, requiring to update only the corresponding bytes in persistent storage. To maximize the benefits of *Inch*, we interface a low energy, byte addressable FRAM as persistent storage.

Compared to the conventional approach on mote, we show that *inch* requires fewer bytes to be updated in the secondary storage (Avg: 6.65x, Range: 1.06x–13.40x), thus prolonging the application execution time and needing less energy (Avg: 5.48x, Range: 1.02x–12.00x) for checkpointing at the most critical time of system operation. *inch* is near optimal (in terms of update size),platform independent, provides legacy support, and a system based on Inch is more dependable and achieves significantly higher task completion rate. *inch* is near optimal (in terms of update size), platform independent, provides legacy support, and a system based on *inch* is more dependable and achieves significantly higher task completion rate.

Chapter 3

INTRODUCTION

The increasing dependence of embedded sensing devices exclusively on harvested energy breaks the assumption of uninterrupted energy supply prevalent in existing computing paradigms [19, 7]. Harvested energy, either from natural sources or due to intentional provisioning through wireless energy transfer, is intermittent [6, 24, 25]. Since these transiently powered devices harvest and compute simultaneously, they are expected to loose power several times in a second [19, 25]. Developing new *energy conscious* computing paradigms (aka. *intermittent computing*) is thus essential for their successful operation. A preliminary requirement is to enable these devices to resume and not restart the previously running computations.

The conventional approaches of intermittent computing [5, 20, 3], borrowing from the history of fault tolerance in a wide range of computing systems [2, 11, 4], checkpoint computational state (registers, global variables, and call stack etc.) before a power blackout and restore the state at the start of the next activation cycle. In the context of embedded sensing, using computational RFIDs [7] or motes, any system support for intermittent computing must be *energy efficient*, to perpetuate maximum energy for application execution, and *execute quickly*, to minimally disrupt normal execution. In particular, the checkpointing operation is initiated at a very critical time of system operation, i.e., when the power is about to be lost. Any failed checkpointing attempt, along with wasting precious time and energy resources, will lose all prior computations.

Recent intermittent computing solutions for transiently powered devices are suboptimal due to both software and hardware limitations.

- *Software*: They always checkpoint the complete program state, either the whole RAM [16] or at least its used portion [5, 3, 20], even if few RAM locations have been modified since the last checkpoint.

- *Hardware*: The flash technology, which is the prevalent choice for persistent storage in commodity embedded devices, requires to erase sufficiently large sectors before rewriting even a single byte.

The resulting penalties significantly reduce the overall performance of existing solutions. For example, redundant checkpointing of unmodified RAM locations (an I/O operation) consumes considerable amount of execution time and energy that could otherwise be used for application execution. These solutions accept this penalty because they lack the ability to precisely determine which RAM locations have been altered since the last checkpoint. Even with that ability, the *write-sectorErase-write* semantics of flash storage restrict the system from realizing the maximum benefits of a smaller update size: Updating a single byte requires to erase the complete sector and then rewrite the corresponding (potentially unmodified) bytes of RAM. Previous studies [14, 13], and our experiments on TelosB external flash, reveal that erasing a 64KB sector could take up to one second, which is equivalent to four million computation cycles on MSP430 running at 4MHz, and could cost around 14.4mJ of energy, which is sufficient to transmit dozens of application packets over IEEE 802.15.4 [15].

We resolve these shortcomings through both software and hardware instrumentation. While software instrumentation by itself can arguably show the promise of **IN**cremental **CH**eckpointing (***Inch***) on flash storage with smaller sector sizes (fewer sectors erased and updated), externally interfacing modern FRAM for persistent storage allows for maximum benefits. FRAM is a cheap and, when compared with flash, by order-of-magnitude more energy efficient alternative for storage previously employed in storage-centric WSN [14].

The key idea of *Inch* is to pro-actively record the *locations of modifications* in RAM and, when required, only update the corresponding bytes of the checkpoint in persistent storage. Depending on what is being modified, *Inch* employs two different techniques for tracking changes in RAM.
(1) **Global variables** are tracked individually. This technique is based on the observation that global variables last for as long as the program, and depending upon the application, are rarely modified (cf. Table 3.1 and Figure 3.2).
(2) **Local variables** are tracked collectively. As they are frequently modified, short-lived, and wiped off from the stack as soon as a function returns, the computational overhead of their fine-grain tracking cannot be justified.
Therefore, *inch* trades additional computational cycles of tracking changes in RAM

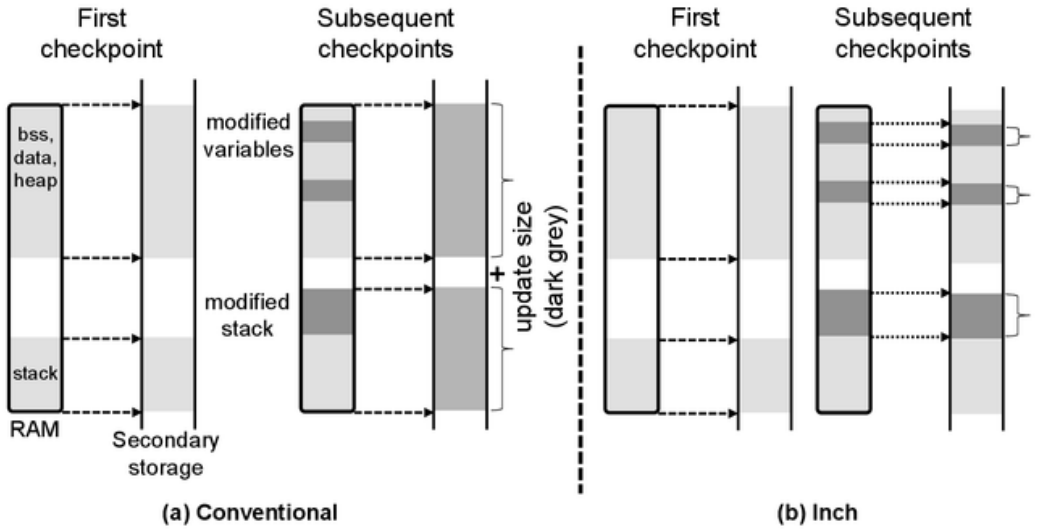


Figure 3.1: **Conventional approaches vs. *Inch*: Instead of always checkpointing the complete program state, *Inch* only incrementally updates the checkpoint with modified RAM locations.**

for a reduction in the expensive *write* operations to external storage. This tradeoff makes *inch* more dependable as fewer write operations are more likely to succeed at the critical time of system operation: Tracking is done when the power is available while checkpointing is done when the power is about to run out.

An *optimal checkpointing* solution in terms of memory usage will precisely determine, on the level of single registers and RAM cells, what changes have occurred in the program state since the last checkpoint. *inch* gets close to but does not fully reach this optimal case, as it tracks changes in global variables individually but (i) is unable do so for a handful of *processor registers*, requiring to extract non-trivial platform dependent information from the compiler, and (ii) avoids tracking of *local variables* to eliminate wasteful computational overhead. We deliberately decided against this additional tracking of changes to avoid unnecessarily complicating our platform-independent system design and gain performance benefits, respectively, aiming for maintainability and overall energy efficiency instead.

With the general aim to explore the feasibility of an incremental checkpointing solution, we make the following contributions:

- Through the design of *Inch*, we show that it is possible to proactively identify changes in the global state of a program and avoid the redundant checkpoint-

Table 3.1: Code lines modifying global variables

App	OS	LoC1	Global Variables [bytes]	# LoC Updating Global Variables
tail (cmd)	Linux	1552	26	20
ls (cmd)	Linux	3436	2554	104
μIP-nd2	Contiki	753	45	45
μIP2	Contiki	1387	6393	72
RPL	TinyOS	19980	4627	433
CTP	TinyOS	18864	3886	351

This is where authors provide additional information about the data, including whatever notes are needed.

ing overhead through automatic, platform independent, and legacy support enabling code instrumentation (Sections 4 and 5).

- To avoid unnecessary computational overhead of tracking a function's short-lived local variables, we develop a novel algorithm that monitors the call stack for changes at the level of function *call* and *return* requiring to only update the modified stack frames in persistent storage (Section 5).

- Through the implementation of *inch* and a straightforward interfacing of FRAM with TelosB using SPI, we maximize the benefits of incremental checkpointing at a cheap price (i.e., $2 per node) while highlighting the limitations of flash storage (Section 6).

- Our comparative evaluation with the conventional approach on sensor motes reveals that *inch* requires fewer updated bytes in secondary storage (Average: 6.65x, Range: 1.06x–13.40x), thus prolonging the available time for application execution and needing less energy (Average: 5.48x, Range: 1.02x–12.00x) for checkpointing at the most critical time of system operation. The data structure for recording memory modifications at run time consumes $\frac{1}{8}$ of RAM and the resulting computations incur processing overhead (Average: 31.26%, Range: 4.08%-85%). Using simulations, we show that the energy and time salvaged by *inch* due to smaller update size cancels out the processing overhead and significantly improves the overall performance of systems under transient power (Section 7).

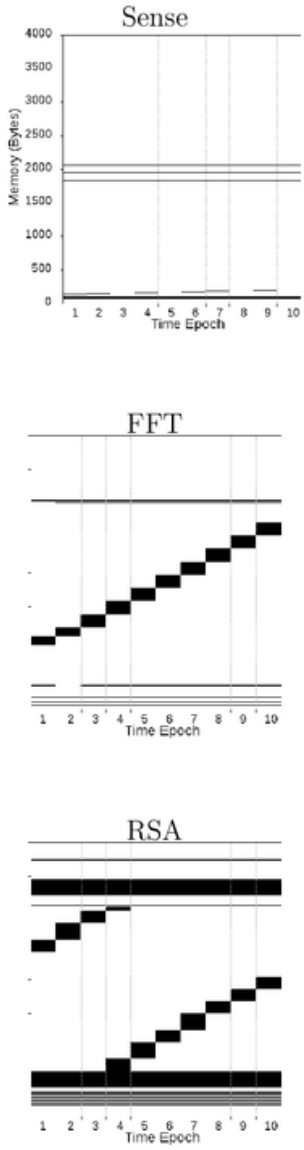


Figure 3.2: Global context (bss + data segment) of sense, FFT, RSA applications (details in Section 7) running on MSP430. Memory dumps are compared after every 32ms execution-time epochs. Black overlines mark modified memory locations, which are by an order-of-magnitude fewer than the total area occupied by global variables of the application and the OS.

Chapter 4

OVERVIEW AND APPROACH

The underlying idea of *Inch*, also depicted in Figure 3.1, is to only incrementally update the existing checkpoint in persistent storage. While both, conventional approaches and *Inch*, would initially require to create an identical main memory map in secondary storage, *Inch* optimizes subsequent checkpointing operations. *Inch* will only update potentially modified global variables and stack frames. In order to enable incremental checkpointing, *Inch* has to track modifications in two logically separated contexts of RAM, (i) *global context*, including the area occupied by *bss*, the data segment, and (possibly) the heap, and (ii) the *call stack*. While global context is tracked for modifications at the level of a single variable, the call stack is monitored at the level of a stack frame (a function call and its data).

To track changes in the global context, we instrument the source code to record modifications in the global/static variables. Our approach is based on a simple primitive: The global context is updated through a predetermined set of statements including assignment statement and unary operations. A *precompiler* identifies these statements in the program and, for their each occurrence involving global variables, augment the code to update the list (in RAM) of possibly changed memory locations. We estimate these updates to be seldom: As shown in Table 3.1 for a number of exemplary applications with several thousands lines of code on different OS platforms, the global context was updated in fewer than 5% of the source lines. Similarly, Figure 3.1 depicts the status of RAM over increasing timing epochs for three TinyOS applications (sense, FFT, RSA), which are periodically interrupted for memory dumps. We can see that, in the typical case, the number of modified RAM locations in a certain time period is by order-of-magnitude smaller than the total space occupied by global variables. Our conjecture is that these observations may have their roots in the basic college programming courses where future computer scientists are taught *to use global variables sparingly and only update them when critical.*

The call stack is monitored for modifications at the level of function call and return: If the stack frame is modified, its complete data will be updated in persistent storage when checkpointing. This is to avoid the wasteful computational overhead of tracking local variables individually as they are wiped off as soon as the the function returns. Our proposed algorithm for monitoring the call stack requires just one additional variable, which is accessed only once after each return from a function call to keep track of the stack- and base-pointers since the last checkpoint.

Concluding: *Tracking is the cost of our system while reduced checkpointing is the benefit. Traditional solutions are zero cost (no tracking) with little benefit (checkpointing whole RAM). A space-optimal solution will have high cost (fine-grained tracking) and most benefit (only changes are in a checkpoint). Our proposed solution is efficient (little tracking overhead) and near-optimal (very few extra memory locations are in checkpoint) and thus hits the sweet spot in the cost-benefit spectrum.*

Chapter 5

INCREMENTAL CHECKPOINTING

In this section, we first describe the code instrumentation technique to track changes in the global context and how we resolved some specific challenges. We then expand on the algorithm to monitor the call stack for its growth and decline to ensure that we only update the modified stack frames when checkpointing. Finally, we discuss *Inch* runtime including possible data structures for recording memory modifications and the tradeoffs therein.

The discussion focuses on the generic design and different design options. We discuss specific implementation choices with further implementation details in Section 6.

5.1 Tracking Changes in Global Context

Our approach instruments the source code with the ability to record *all* modifications in the global context (bss, data segment, and heap) at runtime. It is based on an observation that all program variables are modified through a predetermined set of statements, including assignment statement and unary and shift operations. Thus, for each such statement `stmt` modifying the global context through a variable[1] `var`, the *precompiler* adds the call `record(&var,sizeof(var))`, allowing to capture the address and size of `var` at runtime. While this call could be added either immediately before or after a `stmt`, adding it after the `stmt`, even though more optimal as corresponding `var` will only be recorded after being modified, necessitates `atomic` execution of both `record()` and the corresponding `stmt`: If an interruption necessitating a checkpoint occurs immediately after the execution of `stmt` but before the corresponding `var` is recorded, changes to `var` could get lost otherwise. An `atomic`

[1]Including global and static variables, and local pointers which can point to the global context.

block, besides introducing further computational overhead of enabling and disabling interrupts, could potentially critically delay a checkpointing interrupt.

Adding the call to `record()` before `stmt` removes the need for creating an `atomic` block. However, we must still handle the case when the checkpointing interrupt occurs immediately after the call to `record()` is executed. As in this case, upon resumption from the checkpointed state, the corresponding `var` will be modified but not recorded. We can easily handle this by always marking the address(es) of `var`(s) recorded for the latest occurrence of `stmt`, before checkpointing, to be included in the next two consecutive checkpointing operations. This introduces minor additional overhead.

This source-to-source code instrumentation is straightforward when `stmt` resides inside a *compound statement* block (enclosed within a pair of braces {}). However, additional care is necessary when `stmt` occurs at certain unusual locations where the call to `record()` cannot be added trivially. Note that the precompiler always catches these state modifying statements, regardless of where they appear (in a function call or while indexing an array etc.), but it has to decide differently depending upon the exact location inside the program. Table **??** samples a list of these specific scenarios and our corresponding solution, as discussed below.

5.1.1 Singular statement without braces

The C language allows a singular `stmt` succeeding an *if-condition* or a *loop* to be terminated by a semicolon without necessitating braces. In this case, adding a call to `record()` before `stmt` will introduce a semantic error as the preceding *if* condition will only apply to the instrumented call. For all such occurrences of singular `stmt`(s), the precompiler adds a compound statement block (braces) before applying the actual instrumentation, as shown in Table **??**.

5.1.2 The ternary `if`

The ternary operator (`?:`) can also modify global variables as shown in Table **??** but it does not allow adding multiple statements to the *true* or *false* subexpressions (C language peculiarity). Here, we can either convert the ternary operator into a

standard *if-else* or pessimistically record a location as potentially changed in both the subexpressions of the ternary operator. We prefer the latter as the more generic solution: Replacing a ternary operator with an *if-else* may require complex code transformations, e.g., if it is used as a sub expression in another expression.

5.1.3 Optimizing instrumentation of loops

Our final design challenge is optimizing the instrumentation for loops: If a `var` is being modified inside a loop then adding a call to `record()` could introduce redundant computational overhead. It might suffice to record the location of `var` as potentially modified only once before the start of the loop. However, certain peculiarities of such an optimization warrant a more careful examination. We might have to mark the whole loop as `atomic` (see the above discussion in this section). Similarly, loops can contain *if-else* conditions that may apply before updating a `var`. This may lead to false positives in our modification record, thereby increasing the update size and making the resulting checkpointing less optimal. The problem is aggravated further if there are function calls inside a loop. Additionally, this optimization will be very inefficient in the case of arrays, as the array index could be different in each iteration requiring to record the whole array in the beginning. Nonetheless, we believe that such optimizations can be selectively applied depending upon the characteristics of a given loop observable through a precompiler. In particular, local loops without any conditional statements or function calls are earmarked for such optimizations.

5.2 Monitoring the Call Stack

The approach described above only applies to tracking modifications in the global context of the application. For local variables, which are short lived and lost as soon as a function returns, we believe that this approach will introduce significant computational overhead without providing adequate benefits. Thus, we relax our optimality criteria for the call stack, which contains all local variables, from the level of a single variable to the level of local variables within a single function call. Hence, we monitor the call stack for its growth and shrinking (due to function call and return respectively) and, at the time of checkpointing, update the stack frames of all functions that could have been modified since the last checkpoint.

Our algorithm for monitoring the call stack requires one additional variable, *stack tracker* (ST), which keeps track of the *base pointer* (BP) between two successive checkpoints. BP points to the base of the function currently on top of the stack. It was designed to access parameters and local variables which are at a fixed offset from the BP even as the *stack pointer* (SP) moved with push and pop instructions. The algorithm works as follows.

- ST is initialized with and reset to BP after every restore operation.
- ST remains unchanged if more functions are pushed on to the stack.
- ST is set equal to BP whenever a *return* from a function results in BP $<$ ST.
- At the time of checkpointing the memory region between ST and SP is saved.

This algorithm just requires inserting `if (BP < ST) {ST = BP;}` after returning from each non-*inlined* function in the application.

Figure 5.1 depicts the functioning of this algorithm for all possible states of the call stack. Assume at Checkpoint 1, no function call have been initiated so nothing gets saved as ST, SP and BP are all pointing at the base of the call stack. At Checkpoint 2, the frames of all the three functions are added to the checkpoint. Note that ST does not follow BP when stack grows. At the subsequent restore operation, ST is set to BP. At Checkpoint 3, although no new functions are pushed on to the stack, the frame of function F3 is still updated in the checkpoint: During active period between Checkpoint 2 and 3, F3 has not called any other function but its execution might have changed the values of the local variables requiring an update in persistent storage to ensure correctness. Checkpoint 4 and 5 emphasize the fact that ST does not follow BP as long as BP $\geq$ ST. Finally, Checkpoint 6 shows that ST follows BP downwards, i.e., when *return* from a function results in BP $<$ ST. Overall, the idea is to update the function on the top of the stack at the time of last restore along with all the functions pushed on to the stack during execution. Similarly, when the stack remains unchanged or shrinks, only the function at the top of stack is updated.

The two presented approaches (i.e., tracking changes in the global context and monitoring the call stack) can capture all modifications in RAM except for peripherals with direct memory access (DMA). However, in embedded computing platforms, these buffers for DMA are typically allocated by the application or the OS. Depending upon the platform, these buffers can either always be updated in persistent

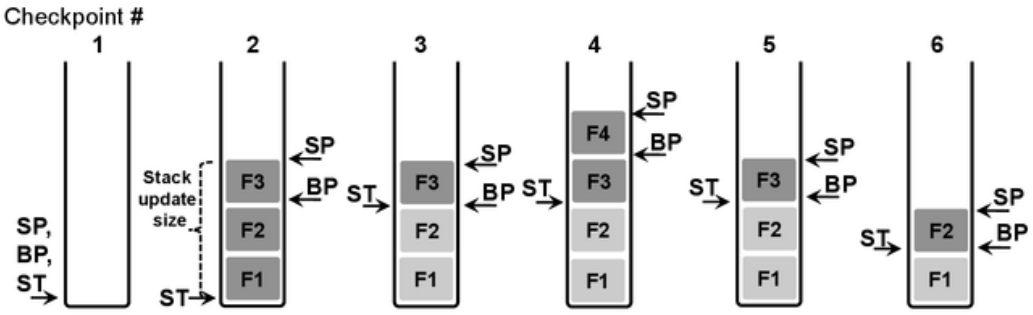


Figure 5.1: Call stack monitoring approach: The memory region between *stack tracker* (ST) and *stack pointer* (SP) is always saved (dark grey). ST is reset to the *base pointer* (BP) at every restore operation. ST always follows the BP downwards but not upwards to determine the modified area of the stack after a previous restore operation. The figure depicts the state of call stack right before each checkpoint.

storage during checkpointing operation or, like global variables, recorded as modified at runtime during the firing of corresponding peripheral interrupts.

5.3 Inch Runtime

The *Inch* runtime is responsible for the following three operations:

- Maintaining the data structure of modified RAM locations and update it whenever `record()` is called.
- Saving the recorded locations along with processor registers and modified stack frames whenever the checkpointing operation is initiated.
- Restoring the application from persistent storage if the previous checkpointing operation was successful.

Below we discuss each operation in more detail.

5.3.1 Recording Modified RAM Locations

We considered two options for the data structure of modification record, a *bit-array* map of all memory addresses or a *list* of modified addresses, both having their own pros and cons. In a bit-array, each bit represents the state of one byte in memory

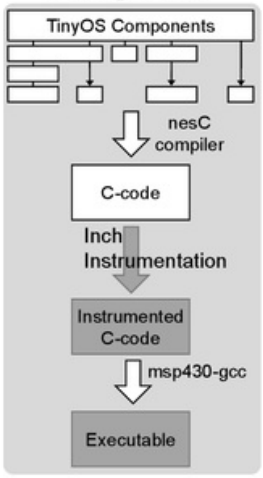
(unmodified = 0, modified=1). This results in a memory overhead of $O(\frac{1}{8}n)$ bytes—for n-bytes memory, an access overhead of $O(1)$—when `record()` is called to update the corresponding bit(s), and a linear traversal overhead of $O(\frac{1}{8}n)$—to find modified bytes at the time of checkpointing. In a list, each node is a `struct` of two elements, a two-byte pointer to store the modified memory address (in the case of a 16-bit address bus) and a one-byte integer to store the size of the modified variable.

A list introduces memory overhead of $O(3k)$—if k locations have been modified ($k \leq n$), access overhead of $O(k)$—for both insertion and traversal as we will have to traverse the whole list to avoid inserting duplicate memory addresses and visit each element to find its address at the time of checkpointing, respectively. The list can only be beneficial in terms of storage and traversal, if the number of modified bytes remains less than $\frac{1}{24}n$ and $\frac{1}{8}n$, respectively. We note that the data structure for recording RAM modifications is not checkpointed as we always need to start afresh and only record modifications between two consecutive checkpoints.

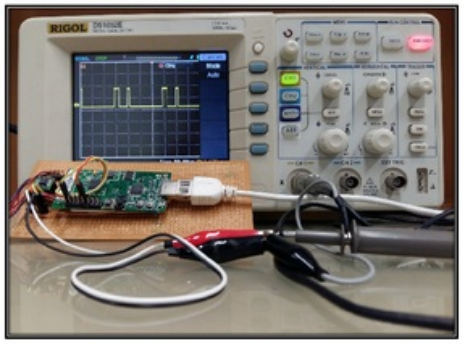
5.3.2 Checkpoint and Restore

The operational principles of the checkpoint/save and restore functions follow the conventional approaches [20, 5] except that we only update the modified RAM locations instead of the complete program state. To validate the integrity of a successful checkpoint, we add a given random byte sequence before and after the saved program state in secondary storage [20]. At restore time, if the random bytes before and after the checkpoint data are equal, the previous checkpoint operation is considered successful and the system is restored from the point in the code where the last checkpoint is taken.

Inch process.



FRAM interfaced TelosB connected to oscilloscope.



Time required to write a byte into FRAM.

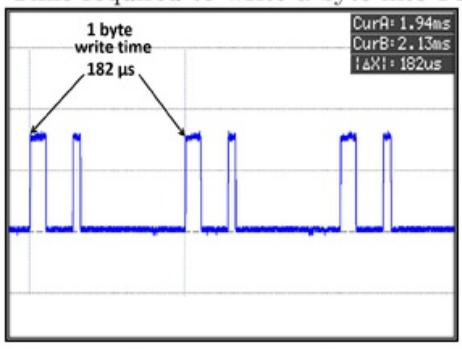


Figure 5.2: *Inch* implementation and experimental setup.

Chapter 6

IMPLEMENTATION

We now present implementation details dealing with software instrumentation and further enhanced by hardware interfacing (see Figure 5.2), followed by a discussion on how we test the correctness of our implementation.

6.1 Software

We developed the *Inch* precompiler for C language based on ANTLR [17]. For our evaluation, we target the MSP430-based TelosB running TinyOS. Our primary reasons are: (1) TelosB hardware has been thoroughly profiled for energy consumption, execution time, and peripheral delays empirically [14] and in simulations [12, 1]. This is an important aspect because it allows us to maintain our focus on evaluating incremental checkpointing rather than micro-benchmarking and verifying the evaluation platform itself. (2) TinyOS includes an extensive and solid code repository enabling us to thoroughly evaluate the correctness of the *Inch* implementation, especially the precompiler used for instrumentation. Finally, (3) TinyOS simplifies code instrumentation across the whole application and OS code, as the nesC compiler generates a single C file containing all code (no precompiled libraries), as shown in Figure 5.3. However, we note that, as also should be evident from our design description, that the presented approach is platform independent and we do not foresee any inherent challenges in porting our implementation to other C based platforms.

With regard to instrumenting the call to `record()`, we opt for adding it before the corresponding `stmt` and providing necessary implementation support to ensure correctness (see Section 5.1). Currently, we do not enable loop-optimizations (see Section 5.1.3). Including these in a later step may further improve incremental checkpointing performance. For the *Inch* runtime, we employ the bit-array data

structure based on its fixed storage requirements and deterministic computational overhead. Our current implementation requires an additional 1464 bytes of data in RAM (including the bit-array) and 1228 bytes of code memory in ROM on TelosB hardware using TinyOS.

6.2 Hardware

To make full use of our incremental checkpointing approach, we attach a 128 Kbit FM25V01 FRAM [8] as persistent storage using SPI (see Figure 5.3). Like MStore [14], which uses a similar FRAM, we also notice that our energy measurements vary from the readings provided in the datasheet [8]. Figure 5.3 shows the write time for one byte on an FRAM, which is substantially less than writing a byte to TelosB's default external flash chip. A detailed comparison between FRAM and flash is available in [14]. Our detailed measurements using an oscilloscope to read and write different sizes of data into FRAM are provided in Table 7.1.

6.3 Testing Implementation

We test the correctness of *Inch* implementation using two methods. First, for a wide range of TinyOS applications in the official repository, we compare the memory modifications recorded by *Inch* with the ground truth, i.e., the memory dump. Second, we run these applications after *Inch*'s code instrumentation and periodically interrupt them for checkpoint and restore to see if the execution completes normally. For the *Blink* application, it would mean that the LEDs are incrementally displaying the counter value even with several checkpointing interruptions. This testing helped us resolve bugs in our precompiler when confronted with different coding styles (cf. Section 5.1). At the same time, this gives us confidence that our implementation correctly checkpoints and restores a (very) wide range of applications. However, the current implementation still has limitations with regard to checkpointing state-full peripherals. We discuss these limitations and potential solutions in Section 9.

Having discussed implementation and and its validity, we next evaluate the performance of *Inch* when compared with the state-of-the-art in intermittent computing. Accordingly, we do not focus on benchmarking different design options discussed

in Section 5. Instead, given our implementation decisions, we want to evaluate if *Inch* performs better than the state-of-the-art, and if yes, by how much?

Chapter 7

EVALUATION

Our evaluation addresses the following four research questions regarding *Inch*:

- **RQ1:** What is the magnitude of *reduction in the number of write operations* to update the checkpoint when compared with the conventional approaches? (Section 7.1)
- **RQ2:** If *Inch* requires fewer bytes to be rewritten in the persistent storage, then what are the corresponding energy gains? (Section 7.2)
- **RQ3:** What is the computational overhead of tracking RAM modifications? (Section 7.3)
- **RQ4:** Finally, what is the cumulative impact of the above two on the overall performance and dependability of a system under transient power? (Section 7.4)

To answer these research questions, we compare *Inch* with a conventional checkpointing approach when both are using FRAM. This is sensible because (i) FRAM interfacing already resolves hardware limitations, and (ii) any such comparison across different storage platforms (e.g, FRAM vs flash) would be inherently unfair to the conventional approach because of the significantly superior energy and timing characteristics of FRAM [14]. Nonetheless, we extrapolate the results to show how *Inch* would turn up in comparison to the conventional approach without FRAM interfacing. That is, when using commodity flash storage devices with different sector sizes.

For our comparison, we define the *best conventional approach* as the one that only checkpoints the registers, the global context (bss + data segment), and the call stack [20]. We note that such a conventional approach, as opposed to *Inch*, lacks the ability to automatically keep track of heap memory allocations. Thus, it

Table 7.1: Measured FRAM Characteristics

Operation	**Size [bytes]**	**Measurement**	**Value**	
Read	1	time	0.146	ms
	256	time	6.47	ms
	512	time	12.8	ms
	-	current	360.0	μA
Write	1	time	0.182	ms
	256	time	7.52	ms
	512	time	14.9	ms
	-	current	360.0	μA

would require either to checkpoint the complete RAM or maintain additional data structures [5] if applied to embedded OS platforms that support heap. However, since we are using a platform that does not inherently support heap, we do not impose the additional penalty of checkpointing the complete RAM on the best available conventional approach.

Our comparison is based on the following three applications:

- **Sense:** A typical sensing application that randomly (range: 1ms–10ms) samples a sensor, and calculates the average over an array that can potentially fill up in 1 second if there are no interruptions.
- **FFT:** This application repeatedly calculates the FFT on a randomly populated (32 bit) integer array of size 448 (1792 bytes), 16 bytes at a time.
- **RSA:** This application repeatedly computes RSA over a randomly populated character array of size 2400 bytes, 60 bytes at time.

While *sense* better represents the typical operation of a sensing application that we expect to be executed on a transiently powered device, both FFT and RSA are synthetically created stress tests, typically used in such comparisons [10], to evaluate the solution from a worst case perspective. Although we evaluated several other applications as well, there were no further insights that are not unveiled by the results shown below for these three applications.

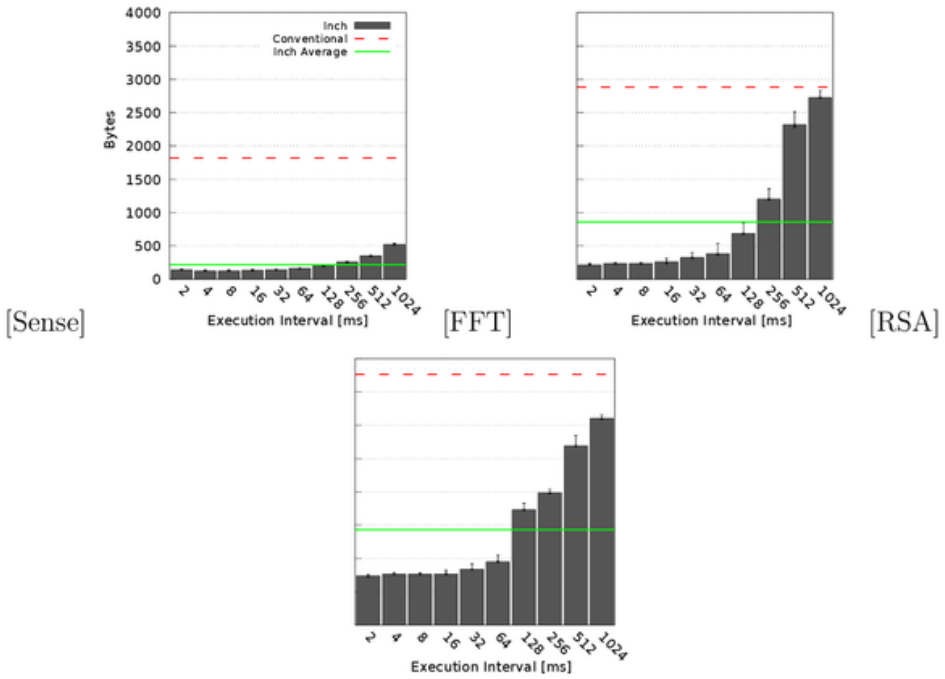


Figure 7.1: Update size: Incremental checkpointing results in a substantial reduction of the number of write operations. The bars represent the average of 10 experiments, the error bars represent their range. We only show the average for conventional approaches (horizontal line) as the variations across experiments are negligible due to severe inlining of functions by nesC and the TinyOS scheduler resulting in a stable call stack—the only potential source of variation in conventional approaches. The *average* improvement across all experiments is 6.65x and the *range* is 1.06x to 13.40x.

7.1 RQ1: Update Size

The principle goal of using an incremental checkpointing approach is to reduce the number of write operations to secondary storage. Figure 7.1 shows the results in terms of the number of bytes that need to be updated for different *execution intervals*. An execution interval of 2ms denotes that the application runs for that much time before it is interrupted to checkpoint its state and restore from secondary storage to resume its operation. Since our primary focus here is to calculate the number of write operations, we ignore the time and energy required by the checkpointing operation until the following sections. We keep execution intervals at sub-second granularity to (i) match the scales of transient power availability [19] , and (ii) remain within the completion time scales of FFT and RSA algorithms, as we want to interrupt them

multiple times during a single execution of the respective application.

In Figure 7.1, each experimental run constitutes ten checkpointing interruptions at the corresponding interval. The bars represent the average of ten repeated runs of each experiment. We ignore the first checkpoint that creates a memory map in the secondary storage for both the approaches, as this checkpoint can be created at system startup. While *Inch* shows minor variations in the results as depicted by the error bars, the variation tends to be negligible for the conventional approach. This is because the call stack, which is the only potential source of variation in conventional checkpointing, is very stable in TinyOS due to (i) aggressive inlining of functions [9] by the nesC compiler to generate the platform dependent `app.c` file, and (ii) the event-based program paradigm underlying TinyOS leading to small computational tasks running directly from a single threaded scheduler.

As expected, we can see that the size of an update, i.e., the number of modified bytes, increases with the execution interval. For the *sense* application, the update size gradually increases because a larger checkpointing interval implies more timer interrupts to read the sensor values and store these in the measurement array. For both FFT and RSA, the update size stabilizes after around 1024ms, when both these applications complete their computations across the whole input array and then repeat the same computations.

After showing the promise of *Inch* when using FRAM, we now want to see how *Inch* would perform, in terms of the update size, when using flash storage. Its is important to note that, when dealing with flash, the overall update size is not only dependent upon how many bytes have changed in RAM but also where the change has occurred. As an example, assuming 10KB of RAM being checkpointed on a Flash with 2KB sector size, modifications in just five memory locations could require erasing and rewriting as little as one sector (when these five bytes are located in the same sector) or as much as all the five sectors (when the five modified bytes are all mapped into different flash sectors). The reason for this lies in the semantics of flash, which require to erase a complete sector before rewriting even a single byte within.

Figure 7.3 shows an extrapolation of our results (of Figure 7.1) for two applications, using different sector sizes available in modern flash storage, at three different execution intervals. We log the bit-array to see where the modifications have occurred in main memory to determine how many sectors would need to be

updated for *Inch*. Since we map the RAM directly on to the secondary storage, RSA application will occupy three sectors when the sector size is 2KB: one for the call stack and two for the global section. Although *Inch* still performs better than the conventional approach especially for smaller execution intervals, the magnitude of improvements is significantly reduced.

7.2 RQ2: Energy Efficiency

We now explore how this smaller checkpoint size can benefit *Inch* in terms of energy consumption of the checkpoint operation, as this is one of the main reason of employing incremental checkpointing at the first place. We evaluated energy consumption of all the three applications at three different execution intervals (2, 64, and 1024 ms), as shown in Table 7.2. The energy consumption is the product of three things: the assumed voltage of TelosB (2.8V), the measured current (mA), and the measured time (ms) of the checkpointing operation. Figure 7.4.1 shows the results in terms of the energy efficiency, i.e., energy consumed by the conventional approach at a particular execution interval divided by the energy consumed when checkpointing using *Inch*. The corresponding gains in terms of the time required to complete the checkpointing operation follow this trend.

However, one of the key observation is that, although encouraging, the energy efficiency results are not fully reflecting the ones from the update size (Figure 7.1). The reason is that writing consecutive bytes to SPI connected peripherals is more efficient in terms of time and energy than writing individual bytes, as these need to be addressed individually and cannot be written in batch. Nevertheless, in *Inch*, we have all the information available to dynamically select between incremental and batch checkpointing taking into consideration *memory layout* and these costs. Overall, however, these results magnify the energy inefficiency of the conventional approach due to redundant writes to secondary storage.

7.3 RQ3: Computational Latency

Tracking changes in RAM leads to computational overhead. We compute the latency introduced by this computational overhead by letting the applications run without

Table 7.2: Energy Measurements

Application	**Execution Int. [ms]**	**Inch [μJ]**	**Conventional [μJ]**
Sense	2	4.47	53.61
	64	7.82	53.61
	1024	18.78	53.61
FFT	2	7.83	84.68
	64	13.24	84.68
	1024	82.80	84.68
RSA	2	23.87	111.25
	64	31.19	111.25
	1024	94.81	111.25

any checkpointing interruptions for sixty seconds and count the number of times the algorithm is completed. Figure 7.4.1 shows the average completion time for the three algorithms (for a single input not the entire array). In the case of sensing application, we define completion time as the average time taken to update the array with a sensor reading and calculate the average including the timer delay. The precision of these results, which is imposed by the TinyOS's timer interfaces, is limited to milliseconds.

We can see that the computational latency of *Inch* is lower for FFT than for RSA. This is due to the nature of these algorithms: The FFT algorithm reads once from the global array and stores its intermediate results in local variables, which are not individually recorded, before writing the final result back to the global array. In contrast, the RSA algorithm has five global intermediate arrays which are updated throughout the computation. The speedup in the case of FFT also justifies our choice of not tracking local variables individually.

7.4 RQ4: System Performance

So far we have independently evaluated two opposing performance factors of *Inch*: energy for checkpointing vs. computational overhead. *Inch* reduces the former but increases the latter, and its overall performance depends on which factor dominates the other. Similarly, at the system level, we want to evaluate *Inch* when the system is concurrently performing other energy hungry tasks (e.g., radio transmissions) as well.

This evaluation is not possible with synthetic checkpointing interruptions being created at constant intervals for stress testing. The energy and time salvaged by *Inch* can only benefit the system if there is a mechanism in place to use these resources for application specific tasks. However, providing more execution time also means that the update size will eventually increase. Thus, to quantify the overall impact of *Inch* on system performance, we need two additional capabilities: First, we need a transient energy source to naturally impose the requirement of intermittent computing. Second, we need a mechanism that can efficiently decide *when to checkpoint* based on the remaining energy and the current size of an update. Although providing an optimal answer to this question is beyond the scope of this paper, which is restricted to *what to checkpoint*, we propose a simple mechanism here primarily for evaluation purposes. To properly address this issue, we build upon the rich simulation support for TelosB sensor nodes, allowing us to accurately and repeatably evaluate system performance having full view and control over parameters, e.g., energy buffers and recharging rates.

7.4.1 Evaluation Setup

Our simulation setup combines the power of two extensions of the TOSSIM simulator, TimeTOSSIM [12] and simulating intermittently powered devices [1]. By instrumenting the code with cycle count information, TimeTOSSIM enables us to model the execution time (and energy) of TelosB, including the drivers of most peripherals with >95% average accuracy. A recent extension [1] extends the PowerTOSSIM [22] framework with laser-based intermittent energy supply, solar panel-based energy harvesting, an energy buffer, energy consumption models of TelosB and its peripherals, and node behaviors such as shutdowns and reboots. Finally, we added a trace-based FRAM model using the general PowerTOSSIM [22, 18] framework.

Combining the capabilities of both these extensions only requires to first instrument the simulation code with TimeTOSSIM's grammar [12] (for timing accuracy), and then use the instrumented hardware simulation wrappers in [1] (for modeling power consumption). With this, we have all the components necessary to evaluate[1] system level performance of *Inch*.

However, there are a few potential sources of inaccuracy: First, the call stack cannot be monitored trivially (cf. Section 5) as TOSSIM runs as a Linux pro-

[1]Link to customized simulation models and parameters omitted for double blind review.

cess. This has a negligible impact (see Figure 7.4.3) due to aggressive inlining by the nesC compiler. Nonetheless, we add a constant overhead, measured empirically, for both the call stack and processor registers to maximize the accuracy of the simulation. Second, TimeTOSSIM cannot model the different sleep modes of the MSP430. Hence, any related optimizations cannot be applied to benefit either checkpointing approach. Finally, as opposed to other peripherals which are replaced by slim simulation wrappers at the device level in TOSSIM, the radio model abstracts at a higher level. However, profiling the radio for simple send/receive, without any competing senders, provides an accurate timing and energy consumption profile [1, 22] to be fed into the simulation.

7.4.2 Simulated Application

We evaluate a simple yet comprehensive sensing application that periodically performs a series of four concrete functions; (i) read from sensor, (ii) compute the average over the past one minute of readings, (iii) store the value in persistent storage, and (iv) transmit the average over the radio. It is further appended with a simple mechanism that decides *when to checkpoint* the application state, as shown by the flowchart in Figure 7.4. Before performing any of the four functions, the application reads the state of the energy buffer to determine if it has sufficient energy to perform the next function and potentially update the checkpoint. If the answer is not affirmative, the application immediately updates the checkpoint. The energy for checkpointing depends upon the size of the update, i.e, the number of write operations required in persistent storage. Factors that contribute to the overall energy consumption of a system running this application include application functions (i) to (iv) and checkpointing, reboot, and restore operations. Hence, checkpointing is only one of several energy expensive operations performed by the system.

Our simulation uses two nodes: one that wirelessly supplies energy, at a certain recharging rate, to the other that is running the application. As an example, a recharging rate of 1% over a recharging cycle of 100s would signify that a node is charged for 1s in each cycle. However, as the recharging time is chosen at random, the node cannot predict future energy and must checkpoint its state whenever the remaining energy falls below a corresponding threshold.

7.4.3 Results

The results are shown in Figure 7.5. With regard to the accuracy of the update size, our simulation results follow the empirical measurements (cf. Figure 7.4.3) closely with the difference disappearing when the update size reaches its maximum under *Inch*. For evaluating system performance, we consider two factors: the number of *tasks completed* and the *dependability* of the system. The former is measured by the number of packets successfully transmitted, as this is the last in the sequence of four functions performed by the application. The latter represents the duration for which the node remains active for a given recharging rate. Our best-case benchmark is 100% for both these factors, i.e., when the node runs uninterruptedly.

Figure 7.4.3 shows that the *Inch* enabled system becomes operational earlier as packet transmission starts at below 1% recharging rate and significantly outperforms the conventional approach (≈30%) by preserving energy for application specific tasks. For example, at a recharging rate of 5%, the *Inch* enabled system completes ≈70% of its tasks compared to a conventional system that only reaches 40%.

Figure 7.4.3 depicts that a transiently powered sensor node that uses incremental checkpointing is more dependable (reduced node-off-times). This is mainly due to it often delaying checkpointing because of its smaller update size, thereby increasing the chance that the system will be able to harvest energy before loosing power altogether and shutting down. In contrast, the conventional approach has to checkpoint as soon as the power level reaches a predefined threshold, which is typically much higher than the dynamically adjustable threshold for *Inch*. Figure 7.4.3 plots this node behavior over time for three different recharging rates.

Overall, an *Inch* enabled system becomes operational earlier, achieves higher task completion rate, and is more dependable because of smaller off times. This is particularly pronounced when the recharging rate varies in the range from the system not being able to do anything to the system being able to run continuously, as it then may avoid checkpointing altogether.

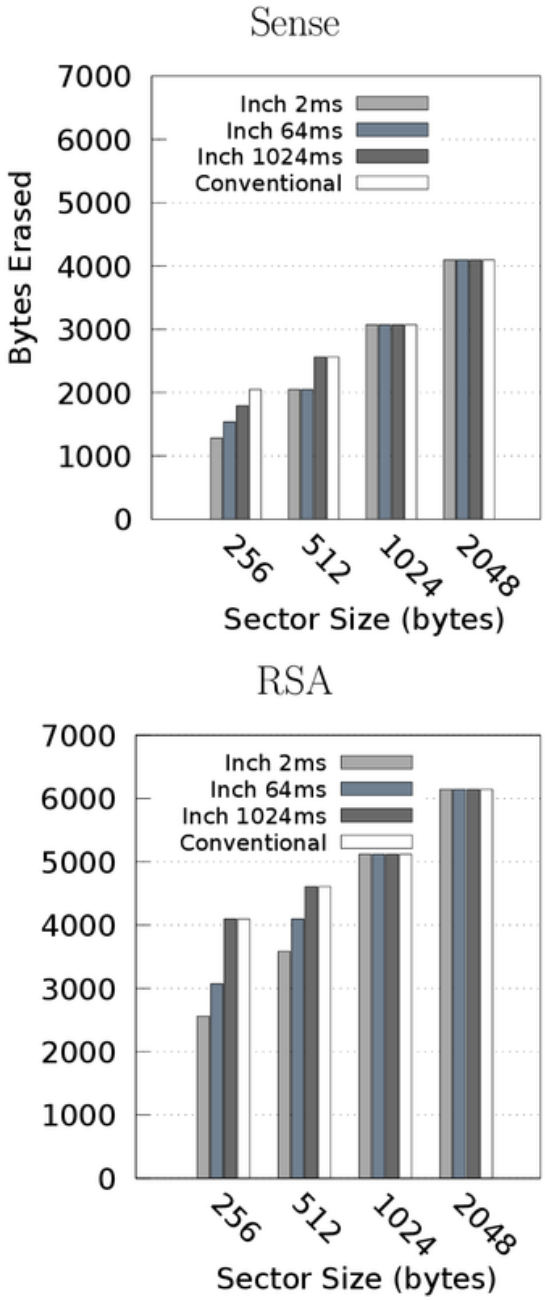


Figure 7.2: Flash-storage: Due to the smaller checkpoint size in *Inch*, fewer sectors will be erased on flash with smaller sector sizes.

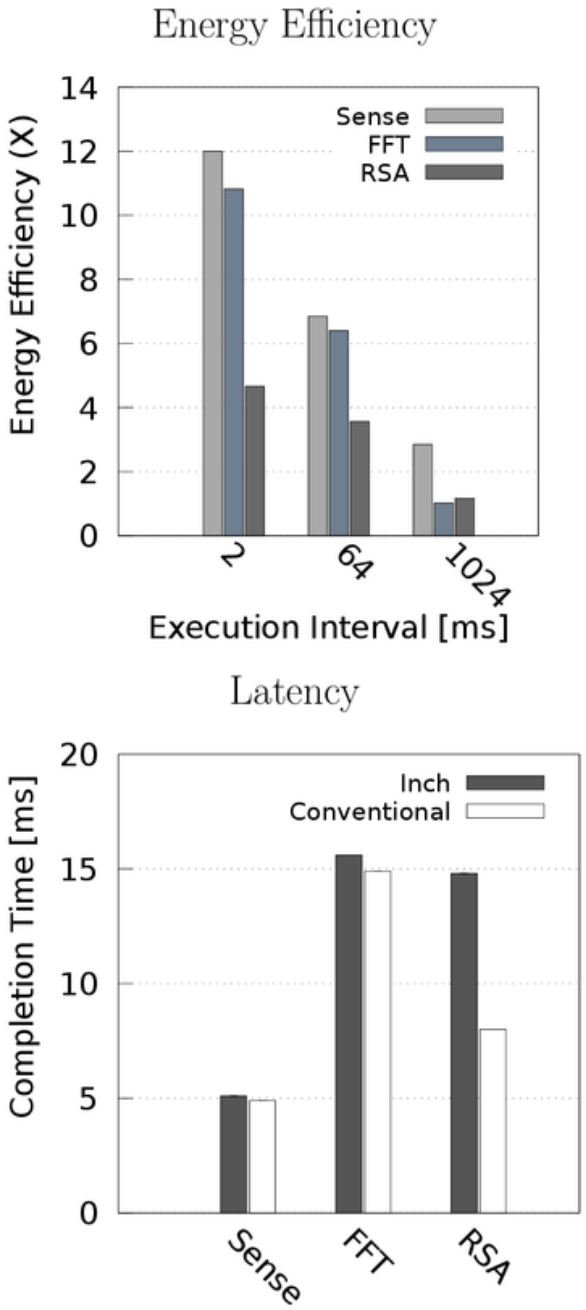


Figure 7.3: Advantages and Disadvantages of *Inch*: Due to fewer writes into FRAM, *Inch* consumes substantially less energy (Average: 5.48x, *Range*: 1.02x–12.00x) for checkpointing. However, tracking modifications in RAM introduces computational overhead increasing an algorithm's completion time (Average: 31.26%).

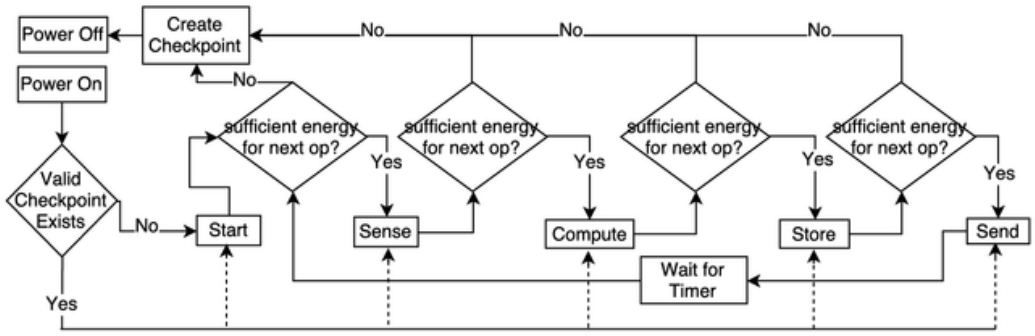


Figure 7.4: Flowchart of sensing application to evaluate system level performance.

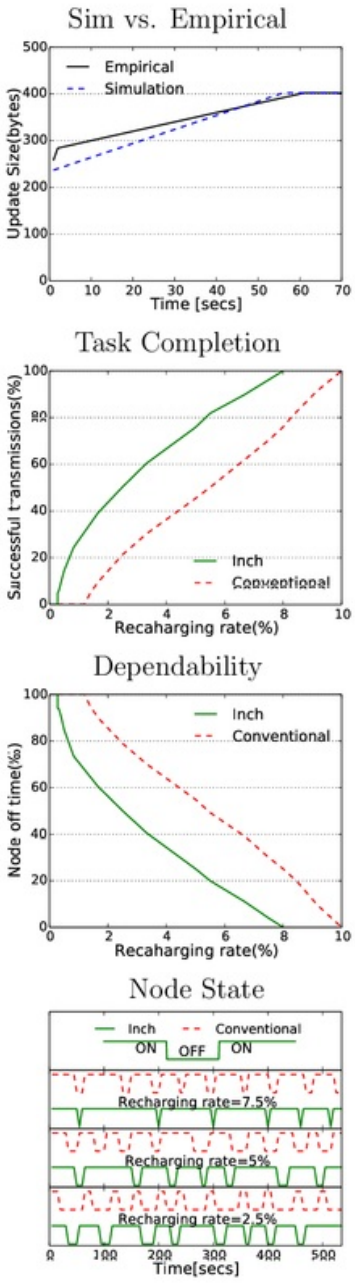


Figure 7.5: System level performance: The *Inch* enabled system becomes operational earlier, achieves higher task completion rate ($\approx 30\%$), and is more dependable, when the recharging rate varies around the threshold of the system not being able to do anything to the system able to run continuously.

Chapter 8

RELATED WORK

Due to the increasing dependence of embedded sensing devices on harvested energy, intermittent computing is gaining increasing attention [3, 20, 5, 10, 21]. However, most of these existing approaches lie within the ambit of the *best conventional approach* that we used in our evaluation. That is, they try to carve the functionality of conventional approaches to suit a certain embedded sensing platform (e.g., to include heap memory support), an issue that is orthogonal to the problems addressed in this paper.

Mementos [20] can be regarded as the pioneering solution of intermittent computing in the embedded sensing domain. It allows to checkpoint the registers, call stack, and programmer-selected part of the bss and data segments. However, it lacks support for heap memory. *Inch* in principle supports heap memory because the approach to track memory modifications is independent of any particular functionality: It captures all the statements that result in memory modifications regardless of the functionality and where they appear in the source code. However, evaluating this claim requires applying *Inch* to platforms that support heap memory, which is a future work.

Bhatti et. al. [5] present extensions to the conventional checkpointing approach, such as Mementos [20], to provide heap memory support without requiring to checkpoint the unused memory space (the area between the end of the heap and the top of the stack). Hence, their first two extensions, *split* and *heap-tracker* [5], are functionally equivalent to the conventional approach in our setting. Their *copy-if-change* approach [5] compares the previous checkpoint with the current RAM content and only updates a certain sector in the flash if its corresponding RAM content has changed. This approach is motivated by a property of modern flash chips: Read operations are faster and more energy efficient than write operations. The premise of this justification is significantly less pronounced in the case of FRAM (cf. Table 7.1): The

byte-by-byte comparison along with writing modified bytes results in more FRAM accesses than simply rewriting used RAM locations. Similarly, this approach does not help to decide *when to checkpoint* the system state? Typically, the decision to checkpoint is made based on how much energy will be consumed by the checkpointing operation [3]. This has to be known in advance. Otherwise, any system support for intermittent computing will in any case have to reserve an energy budget equivalent to saving the complete RAM to ensure successful checkpointing even if *copy-if-change* later determines that lesser energy was needed as fewer bytes were modified. In contrast, *Inch* can expose such information and help make adaptive decisions without requiring expensive comparisons between RAM and persistent storage.

We regard QuickRecall [10] to be a different class of solutions in the intermittent computing domain as it advocates for FRAM to be used as main memory in transiently powered devices. The hardware limitations of FRAM, as it is currently significantly less dense than SRAM, makes it difficult to predict when such a proposal will be rolled out to become a commodity in the embedded sensing domain. There are further limitations. Most importantly, QuickRecall lacks legacy support. It imposes requirements on application programmers, e.g., the use of predefined QuickRecall variables and the provision of own initialization routines when recalling the system state. One of the key advantages of *Inch* is that it provides legacy support, as demonstrated by its implementation in TinyOS. Furthermore, it integrates seamlessly with existing applications through a grammar based code instrumentation.

Chapter 9

LIMITATIONS

In the current state of its implementation, *Inch* has two limitations that make up our primary future work:

First, any approach to intermittent computing has to answer two questions: (i) *when to checkpoint the system state*, and (ii) *what to checkpoint*. This paper only answers the second question. Although we provide a simple solution for evaluation purposes, finding an optimal answer to the first question is a research problem in its own right, orthogonal to improving the efficiency of a single checkpoint operation. For example, hibernus [3] considers this problem and proposes reactive hibernation to ensure that the system state is checkpointed only once before the power is about to be lost. Dewdrop [7] looks at this problem from a different angle: How to reschedule tasks and device wake-up times to achieve maximum runtime? Whether these solutions can integrate with *Inch* or if we need a radically different approach is a primary concern going forward.

Second, *Inch* only checkpoints the processor state which might not be sufficient to recover the system given stateful peripherals, such as a radio chip. Common embedded sensing platforms use multiple peripherals to gather data from sensors and to communicate with the external world. Therefore, retaining the state of these peripherals may be essential to correctly restore the system after a power loss event. Right now, we overcome this problem by only allowing the system to start peripheral operations if the remaining energy guarantees completion (cf. Figure 7.4).

Both these limitations are not technical in nature and solutions do exist [23, 3]. Thus, future versions of *Inch* implementations, which currently deliberately focus on establishing the feasibility of an incremental checkpointing proposal, will be geared towards addressing these limitations.

Chapter 10

CONCLUSION

This thesis work proposed incremental checkpointing for enabling intermittent computations. Our empirical evaluation and simulation results demonstrated the feasibility of this proposal—the goal we set forth for this paper. We showed that the reduction in checkpoint update size salvages precious time and energy resources to improve the performance of systems under transient power.

Besides addressing the limitations discussed in Section 9, we are also interested in exploring alternative and orthogonal incremental checkpointing approaches. We are exploring to bind variables to program paths, whose execution is triggered by a given event, such as an interrupt from the radio, a sensor, or a timer. By remembering the higher level events that have occurred since the last checkpoint instead of tracking all changed variables, we may over-estimate which variables have been modified but at the same time significantly reduce time and memory overhead. We are also considering the possibility of reallocating the variables in RAM based on their frequency and correlation of modification, further improving the efficiency of checkpointing operation. As our current implementation is based on TinyOS, applications typically do not make use of heap memory, which may need special attention such as fragmented memory and latency of writing.

Bibliography

[1] Muhammad Hamad Alizai et al. "Simulating Intermittently Powered Embedded Networks". In: *EWSN*. 2016.

[2] Luís Almeida, Paulo Pedreiras, and José Alberto Fonseca. "The FTT-CAN protocol: why and how". In: *IEEE Trans. Industrial Electronics* 49.6 (2002).

[3] Domenico Balsamo et al. "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems". In: *Embedded Systems Letters* 7.1 (2015).

[4] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[5] Naveed Anwar Bhatti and Luca Mottola. "Efficient State Retention for Transiently-powered Embedded Sensing". In: *EWSN*. 2016.

[6] Naveed Bhatti et al. "Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences". In: *ACM TOSN* (2016).

[7] Michael Buettner, Benjamin Greenstein, and David Wetherall. "Dewdrop: An Energy-Aware Runtime for Computational RFID". In: *NSDI*. 2011.

[8] FM25V01. *Data Sheet*. www.cypress.com.

[9] David Gay et al. "The nesC Language: A Holistic Approach to Networked Embedded Systems". In: *PLDI*. 2003.

[10] Hrishikesh Jayakumar et al. "QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers". In: *J. Emerg. Technol. Comput. Syst.* 12.1 (2015).

[11] Richard Koo and Sam Toueg. "Checkpointing and Rollback-recovery for Distributed Systems". In: *ACM Fall Joint Computer Conference*. 1986.

[12] Olaf Landsiedel, Muhammad Hamad Alizai, and Klaus Wehrle. "When Timing Matters: Enabling Time Accurate and Scalable Simulation of Sensor Network Applications". In: *IPSN*. 2008.

[13] Gaurav Mathur et al. "Ultra-low Power Data Storage for Sensor Networks". In: *ACM TOSN* 5.4 (2009).

[14] Kresimir Mihic et al. "Mstore: Enabling storage-centric sensornet research". In: *IPSN*. 2007.

[15] Hoang Anh Nguyen et al. "Sensor node lifetime: An experimental study". In: *IEEE PerCom Workshop Proceedings*. 2011.

[16] Fredrik Österlind et al. "Sensornet Checkpointing: Enabling Repeatability in Testbeds and Realism in Simulations". In: *EWSN*. 2009.

[17] Terence Parr. *The Definitive ANTLR 4 Reference*. http://www.antlr.org/, 2013.

[18] Enrico Perla et al. "PowerTOSSIM Z: Realistic Energy Modelling for Wireless Sensor Network Environments". In: *ACM PM2HW2N*. 2008.

[19] BENJAMIN RANSFORD. "Transiently Powered Computers". PhD thesis. School of Computer Science, University of Massachusetts Amherst, 2013.

[20] Benjamin Ransford, Jacob Sorber, and Kevin Fu. "Mementos: System Support for Long-running Computation on RFID-scale Devices". In: *ASPLOS*. 2011.

[21] Alberto Rodriguez Arreola et al. "Approaches to Transient Computing for Energy Harvesting Systems: A Quantitative Evaluation". In: *ENSsys*. 2015.

[22] Victor Shnayder et al. "PowerTOSSIM: Efficient Power Simulation for TinyOS Applications". In: *SenSys*. 2004.

[23] Rebecca Smith and Scott Rixner. "Surviving Peripheral Failures in Embedded Systems". In: *USENIX ATC*. 2015.

[24] Vamsi Talla et al. "Powering the Next Billion Devices with Wi-fi". In: *CoNEXT*. 2015.

[25] Guang Yang et al. "Challenges for Energy Harvesting Systems Under Intermittent Excitation". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2014).

Incremental, Near Optimal Checkpointing for Intermittent Computations

ORIGINALITY REPORT

%0	%0	%0	%0
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

EXCLUDE QUOTES	ON	EXCLUDE MATCHES	< 25 WORDS
EXCLUDE BIBLIOGRAPHY	ON		