Poster Abstract: Incremental Checkpointing for Interruptible Computations

Saad Ahmed[†], Hassan Khan[†], Junaid Haroon Siddiqui[†],
Jó Ágila Bitsch[‡], Muhammad Hamad Alizai[†]
[†] Computer Science Department, LUMS
[‡]COMSYS, RWTH Aachen University
{16030047,15030044,junaid.siddiqui,hamad.alizai}@lums.edu.pk, jo.bitsch@rwth-aachen.de

ABSTRACT

We propose incremental checkpointing techniques enabling transiently powered devices to retain computational state across multiple activation cycles. As opposed to the existing approaches, which checkpoint complete program state, the proposed techniques keep track of modified RAM locations to incrementally update the retained state in secondary memory, significantly reducing checkpointing overhead both in terms of time and energy.

CCS Concepts

•**Networks** → *Sensor networks;* •**Computer systems organization** → *Embedded software; Reliability;*

Keywords

Incremental checkpointing, Intermittent computing, transiently powered computing

1. INTRODUCTION

The increasing dependence of embedded sensing devices solely on harvested energy breaks the assumption of a continuous energy supply prevalent in existing computation paradigms. Harvested energy, either from natural sources or due to intentional provisioning of an environment through wireless energy transfer, is typically intermittent [1, 2]. Checkpointing computational state (registers, global variables, and call stack etc.) before power blackout and restoring it at the start of next activation cycle (aka. *intermittent computing*) is thus essential to allow these transiently powered devices to resume, and not restart, the previously running computations. However, any system support for intermittent computing must be *energy efficient*, to perpetuate maximum energy for application execution, and *execute quickly*, to minimally disrupt the normal execution.

Recent state-retention solutions for transiently powered devices [1, 3] are suboptimal; they checkpoint complete program state, either the whole memory [3] or at least its occupied portion [1], each time a call to a checkpointing system is made due to depleting energy buffer. This results in redundant and increased checkpointing overhead (i.e., an IO operation) as even the unmodified state, since the last checkpoint, is unnecessarily rewritten in the secondary storage. These approaches accept this penalty as they lack the ability to determine which RAM locations have changed since the last checkpoint.

An *optimal* checkpointing solution should be able to precisely identify modified RAM locations and only update those in the secondary storage. To this end, we propose two different, platform independent *incremental checkpointing* approaches that can proactively track changes in the computational state.

2. INCREMENTAL CHECKPOINTING

Our first approach is near-optimal, as it accurately tracks and records modifications in the main memory except for processor registers. The second approach avoids such computational overhead by binding variables to program paths, only updating the relevant variables in the secondary storage if the corresponding path has been executed.

2.1 Tracking Changes in State (TCS)

This approach is based on a key observation that the computational state is only changed by a few, well defined statements in the program, such as assignment, increment, shift operations, and function calls and returns. We can instrument the source code to record corresponding memory locations (i.e., by inserting calls to special functions) before the execution of such state-modifying statements. Figure 1 shows that such an instrumentation results in significant reduction in the size of checkpoint. TCS thus offers a tradeoff between computational and checkpointing overhead. Hence, its feasibility can only be established if the energy consumed by these additional computations is *significantly* less than the energy required to checkpoint the complete program state—an expensive IO operation.

2.2 Event to Variable Mapping (EVM)

Our second approach avoids any computational overhead by binding variables to program paths through offline, static program analysis. We define such a program path in an embedded sensing device as the path triggered by a certain interrupt, such as from radio, sensor, or timer. By remembering all the interrupts (or events) that have occurred since the last checkpoint, we can predict which variables could

SenSys '16 November 14-16, 2016, Stanford, CA, USA

ACM ISBN 978-1-4503-4263-6/16/11.

DOI: http://dx.doi.org/10.1145/2994551.2996701

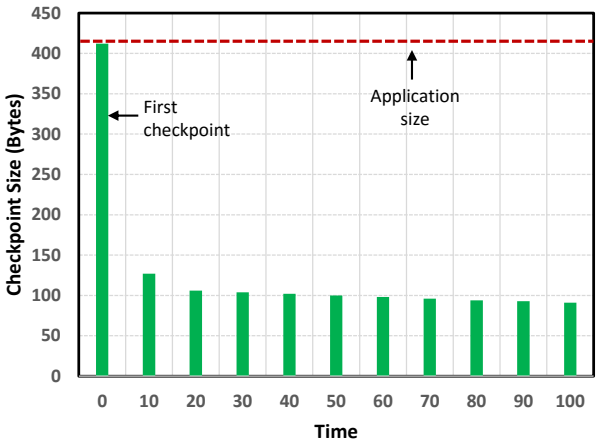

Figure 1: Checkpoint size when using an incremental approach (green bars) is significantly less than the complete application size (horizontal dotted line). This graph depicts the results of a simple TinyOS application that displays a counter, received in a packet, on the three LEDs of TMote Sky. The application is interrupted after every 10 seconds to checkpoint and restore its state.

have been modified. Although EVM is slightly wasteful, as a variable on a certain path might not necessarily be modified after each run (e.g., inside an unexecuted *if* block), it has negligible runtime overhead.

Both these approaches are implemented through platform independent, pre-compiler extensions that automatically instrument the source code with relevant incremental checkpointing functionality.

3. OPTIMIZATIONS

The two presented approaches are restricted by the peculiar characteristic of Flash-storage; requires to erase the complete block before rewriting a certain byte in that block. Hence, in the worst case, these approaches might not offer any advantage over the existing solutions; consider a situation where a few bytes have to be updated in all the occupied blocks (by the checkpoint) requiring a mass erase operation and then rewriting of the complete program state. To address these limitations, we propose both software and hardware optimizations.

3.1 Software Optimizations

Our software optimizations deal with reallocation of variables in the RAM. With regard to TCS, we can, for example, classify variables based on how often they are modified using pre-deployment simulation runs. Variables with higher frequency of modification can be grouped together in RAM so that they can potentially map on to the same Flash block, avoiding frequent mass-erase of all the occupied blocks by the checkpointing system. Similarly, for EVM, we can reallocate variables based on the program paths on which they are encountered. For example, all the variables on the path triggered by a radio interrupt can be grouped together such that a solitary radio interrupt between two consecutive checkpoints will only result in the update of radio-relevant storage block(s).

3.2 Hardware Optimizations

Although software optimizations are useful to mitigate the limitations of block-addressable Flash storage for incremental checkpointing, they still do not allow us to exploit the maximum potential of the proposed approaches. The major limitation is the mass-erase that has to happen at the block level before every rewrite of even a single byte. To this end, we serially interface byte-addressable, random access FRAM with our target platforms (based on 16 and 32 bit MCUs) to maximize the advantage of incremental checkpointing. For an additional cost of just $2 per node, a serially interfaced FRAM that requires only three MCU pins, brings about two additional benefits besides eliminating the mass-erase problem; (i) its energy budget (during read, write, and standby) is in orders of magnitude less than typical Flash storage, and (ii) it liberates the Flash storage from the checkpointing overhead making it completely available for applications.

4. FUTURE WORK

We have already implemented front-end compiler extensions for TCS to instrument the source code with the ability to record state modifications in main memory. Although both the presented approaches are platform independent and can easily be ported to any C based platform, we initially implement it for TinyOS to utilize its extensive application repository for later evaluations. We are optimizing this approach to further reduce the checkpointing overhead by monitoring the size of the call stack between two checkpoints. Furthermore, we plan to maintain a temporary buffer in RAM, i.e., a replica of the checkpointed register values, allowing us to compare fresh register values with the previous checkpoint *locally* before updating in the secondary storage—an expensive IO operation.

In the case of EVM, we are currently automating our static analysis to bind variables to program paths and addressing related challenges. One major concern is the limitation of static code analysis in dealing with *pointer aliasing*, as pointers can point to arbitrarily different memory locations during runtime.

After complete implementation, the bulk of our future work lies in a methodical comparison between the two presented approaches establishing their utility for real world applications. Such a comparison should highlight, for example, the tradeoffs between computational overhead of TCS and the laziness of EVM in identifying modified computational state for incremental checkpointing.

5. REFERENCES

[1] N. A. Bhatti and L. Mottola. Efficient State Retention for Transiently-powered Embedded Sensing. In *Proc. 2016 Int. Conf. Embedded Wireless Systems and Networks*, EWSN '16, pages 137–148, 2016.

[2] N. A. Bhatti, A. A. Syed, and M. H. Alizai. Sensors with Lasers: Building a WSN Power Grid. In *Proc. 13th Int. Symp. Information Processing in Sensor Networks*, IPSN '14, pages 261–272, 2014.

[3] B. Ransford, J. Sorber, and K. Fu. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 159–170, 2011.