Towards Checkpoint Placement for Dynamic Memory Allocation in Intermittent Computing

Nicholas Shoemaker
nshoemaker@branfordschools.org
Branford High School
Branford, CT, USA

Ruzica Piskac
ruzica.piskac@yale.edu
Yale University
New Haven, CT, USA

Mark Santolucito
msantolu@barnard.edu
Barnard College, Columbia University
NYC, NY, USA

Abstract

Energy harvesting allows computational devices to run without a battery, opening new application domains of computing. Such devices work under an intermittent computing model, where the system may power cycle several times a second. To ensure progress, intermittent computing uses checkpoints, with much work being dedicated to this direction. However, no existing approaches handle programs using dynamically allocated memory in the intermittent computing model. We pose this as a challenge area, demonstrate the complexities of checkpointing in this space, and propose key characteristics of an effective solution.

CCS Concepts: • **Hardware → Power and energy**.

Keywords: intermittent computing, memory analysis

ACM Reference Format:
Nicholas Shoemaker, Ruzica Piskac, and Mark Santolucito. 2020. Towards Checkpoint Placement for Dynamic Memory Allocation in Intermittent Computing. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis (TAPAS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3427764.3428323

1 Introduction

Intermittent computing is a model of computation where the system experiences frequent power failures - possibly many times a second. A system will experience power failures as a result of using energy harvesting devices - whereby a microcontroller is powered by ambient energy from the environment (e.g. radiowaves). Although difficult to program [9], intermittent computing devices are well-suited for a number of applications where batteries does not fit the design constraints, such as space computer systems [1] and long-term environmental sensors [11].

Programs written in a continuous power model are unable to run without modification under intermittent power. This is because code keeps important state in volatile memory (RAM), which is irretrievably erased when a device loses power. This means that when a power failure occurs, all of the progress a program has made is erased, and then, when power is restored, execution starts over from the beginning again. One solution is to intersperse code with *checkpoints* [7]. A checkpoint is a procedure that saves the entire state of volatile memory to non-volatile memory, which persists even after a power failure. When the device has enough power again, the system retrieves the state saved in non-volatile memory and continues execution from that state.

While a number of checkpointing strategies have been proposed [3–5], none have support for programs with dynamic memory allocation. This is in part because reasoning about dynamic memory allocation, especially on embedded systems is a difficult problem [2]. However, memory management is important to consider as the cost of a checkpoint is directly proportional to the amount of memory the checkpoint must save. When dynamically allocated memory must be checkpointed, the fluctuation of the heap means that the cost of checkpointing will also vary greatly. This makes the heap incompatible with existing checkpointing strategies that assume a relatively stable checkpoint size.

In this extended abstract, we: 1) Describe the challenge of checkpointing programs with dynamic memory allocation; 2) Demonstrate that optimal checkpoint placement can have up to a 22% speedup in performance when compared to sub-optimal placement; 3) Outline a first approach for optimally placing checkpoints in such a way that is compatible with existing approaches checkpoint placement, and can be used in conjunction with prior work.

2 Motivating Example

To demonstrate the complexity of checkpointing in dynamically allocated code, consider the code in Fig. 1. When inserting a checkpoint, we will consider only line 5, line 10, and line 13 for this example. Placing a checkpoint at line 5 is the worst choice in the case, as we have just spent energy allocating memory, but have not done any useful computation.

TAPAS '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8189-5/20/11…$15.00
https://doi.org/10.1145/3427764.3428323

```
1  int x = 40;
2  int * y = malloc(4*sizeof(int));
3  int * z = malloc(4*sizeof(int));
4  // possible checkpoint (least optimal placement)
5  y[0] = 2;
6  z[0] = 2;
7  x = x + y[0];
8  free(y);
9  // possible checkpoint (suboptimal placement)
10 x = x + z[0];
11 free(z);
12 // possible checkpoint (optimal placement)
```

Figure 1. Code snippet of dynamic memory allocation with potential checkpoint locations.

As a result, our checkpoint will need to copy variable x, the contents of the stacks, and the empty allocated memory of y and z to non-volatile memory to preserve the state of the program. In contrast, line 13 is the best choice for this code snippet, as we have completed the necessary computations and free'd the memory. As a result, the only memory we need to copy to non-volatile memory is the variable x and anything else on stack from the context.

In considering line 10 for a checkpoint, we see that it is a suboptimal location. This is because line 12 will free memory and thus decrease the cost of our checkpoint routine. Thus, if the cost of the computation on line 11 is less than the cost savings from freeing memory on line 12, we should skip the line 10 checkpoint and instead checkpoint at line 13.

3 Impact of Checkpoint Placement

We hypothesized that placing checkpoints in areas of a program that have less dynamic memory usage will increase the overall speed of the code. In order to measure the significance of this performance gain, we created a test setup using real energy harvesting hardware.

To simulate power harvesting conditions, our experimental setup uses a TX91503 PowerSpot to send power over radio waves, and a P2110EVB energy harvesting circuit to harvest the power. A TI-MSP430FR2433 was run off the harvested energy, running code with checkpoints. We implemented a simple checkpointing procedure that saves the stack (cf. [3, 4]), as well as the heap to non-volatile memory. We added checkpoints to 3 programs with dynamically allocated arrays at both optimal and least-optimal locations. We used the standard measurement for intermittent computing: we timed how long it took for the programs to complete at different distances (less energy is harvested at further distances).

We found that there is always at least some performance gain, as shown in Fig. 2, from placing checkpoints at optimal locations based on heap usage (immediately after frees) as compared to the least optimal (immediately after mallocs).

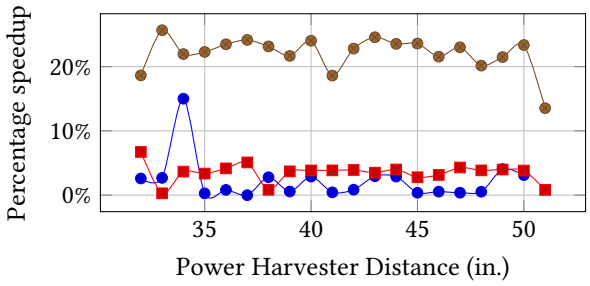

Figure 2. Performance gain between optimal and least-optimal checkpoint placement in a top-down merge sort implementation (blue), bottom-up merge sort implementation (red), and an edge case (malloc arrays and immediately free) program (brown).

Since dynamically allocated memory has a significant impact on checkpoint performance, a checkpointing strategy should take advantage of fluctuations in heap usage. We notice that freeing memory is a *net negative energy cost* computation with respect to any upcoming checkpoint. Thus, to find optimal checkpoint placements, we must balance the energy cost saved by freeing memory with the energy expended by running computations prior to freeing memory.

Determining the optimal location of checkpoints statically is understood to be a poor approach in general due to the variability of power cycles. Instead, most approaches to intermittent computing will place checkpoints at run-time, based on an adaptive, run-time analysis of the power availability. These dynamic checkpoint approaches are well-studied [5, 6], and as such, we aim to find a checkpoint placement algorithm that can compliment existing work. We believe that an effective approach should run as a static analysis procedure. This will allow the proposed solution to be combined with existing dynamic checkpoint placement algorithms [5].

To this end, our proposed checkpointing strategy aims to automatically mark sections of code as no-checkpoint zones. The intuition is that because freeing memory can significantly decrease the cost of a checkpoint, there are zones of the program immediately preceding memory frees that are always suboptimal checkpoint locations. Our compile-time analysis can then be combined with existing run-time analyses to guide the dynamic checkpointing strategies. Determining how far these no-checkpoint zones extend requires us to reason about the size of the heap throughout the program. To this end, combining formal models of intermittent computing [10] with separation logic to reason about memory usage [8] is promising direction.

Acknowledgments

The authors thank Kiwan Maeng for his help on the many challenges of intermittent computing. This work was partially funded by the National Science Foundation under Grant No. CCF-1553168 and No. CCF-1715387.

References

[1] Bradley Denby and Brandon Lucia. 2020. Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 939–954.

[2] Jiangchao Liu, Liqian Chen, and Xavier Rival. 2018. Automatic Verification of Embedded System Code Manipulating Dynamic Structures Stored in Contiguous Regions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37 (2018), 2311–2322.

[3] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices* 50, 6 (2015), 575–585.

[4] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 96.

[5] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 129–144.

[6] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks (TOSN)* 16, 1 (2020), 1–24.

[7] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 159–170.

[8] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

[9] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D Corner, and Emery D Berger. 2007. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*. 161–174.

[10] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2020. Towards a Formal Foundation of Intermittent Computing. arXiv:2007.15126 [cs.PL]

[11] K. S. Yildirim and P. Pawelczak. 2019. On Distributed Sensor Fusion in Batteryless Intermittent Networks. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. 495–501.