Worst-Case Energy Consumption Aware Compile-Time Checkpoint Placement for Energy Harvesting Systems

Bahram Yarahmadi, Erven Rohou

Compas'2019 : Parallélisme/ Architecture / Système
LIUPPA - IUT de Bayonne, France, du 24 au 28 juin 2019

Worst-Case Energy Consumption Aware Compile-Time Checkpoint Placement for Energy Harvesting Systems

Bahram Yarahmadi and Erven Rohou

Univ Rennes, Inria, CNRS, IRISA
first.last@inria.fr

Résumé
A large and increasing number of Internet-of-Things devices are not equipped with batteries and harvest energy from their environment. Many of them cannot be physically accessed once they are deployed (embedded in civil engineering structures, sent in the atmosphere or deep in the oceans). When they run out of energy, they stop executing and wait until the energy level reaches a threshold. Programming such devices is challenging in terms of ensuring memory consistency and guaranteeing forward progress. Previous work has proposed to insert checkpoints in the program so that execution can resume from well-defined locations. In this work, we propose to define these checkpoint locations based on worst-case energy consumption of code sections, with limited additional effort for programmers. As our method is based upon worst-case energy consumption, we can guarantee memory consistency and forward progress.

Mots-clés : Energy harvesting, Worst-Case Energy Consumption, Intermittently-powered MCUs

---

1. Introduction

We live in the era of Internet of things (IoT) where the world around us is surrounded with a large number of tiny objects sensing, communicating and processing data in our environment. For these tiny objects, energy provision and consumption are challenging: it is not economically viable, or even physically possible to configure them with large, heavy, and high maintenance batteries. Recently, using energy harvesting techniques as an alternative way to supply energy without resorting to batteries has been proposed. In these techniques, energy is extracted from different sources in the environment (e.g, sun light or wind) [19, 24] and stored in a buffer such as a capacitor. However, one problem with harvested energy sources is that they are all unstable. This instability of energy sources and the small amount of energy a capacitor can store make the execution of programs interrupted by power failures. As a result, tasks with long running processing time cannot be completed with a single charge of the capacitor. One way to guarantee forward progress to completion of tasks is by leveraging the idea of taking checkpoints. That is, storing all necessary volatile data such as processor state, program stack and heap into a persistent memory before energy depletion. When the energy becomes available again, all the volatile state will be copied back and the program can continue its execution.
On one hand, checkpointing volatile state of the program into the non-volatile memory available in embedded systems seems to be promising, as a program can have intermittent execution to completion.

On the other hand, incautious taking of checkpoints either makes the system not to have forward progress or to suffer from performance and energy degradation. For instance, fewer number of checkpoints than what is needed, called the optimal number of checkpoints, causes at least a section of code to consume more energy than the maximum amount of energy in the capacitor. As a result, it makes the section to be executed repeatedly without any forward progress. On the contrary, taking more checkpoints that the optimal number one, wastes the system energy for doing unnecessary work, since taking checkpoint is not without cost. Also, Ransford and Lucia [20] pinpointed that checkpointing and resuming execution may lead to program correctness violations when the program performs side-effects, such as changing non-volatile data. For example, consider a case that a checkpoint is taken and then the program reads and modifies some data in non-volatile memory. If a power failure happens before reaching the following checkpoint, the system must rollback to the previous checkpoint and re-execute the same instructions. However, the second time, the data in non-volatile memory are not correct.
The contribution of this paper is a technique to automatically insert checkpoints in the code of an intermittently-powered system, with the following properties:

- we guarantee both forward progress and program correctness;
- we limit the burden on programmers to a negligible additional effort;
- we provide a portable software solution not requiring any extra hardware support.

The rest of this paper is organized as follows. Section 2 is an overview of some related work. Section 3 presents our method. We evaluate it in Section 4. We conclude in Section 5.

2. Related work

Researchers have proposed hardware/software and software-only solutions for having forward progress as well as program correctness in energy harvesting systems.
To the best of our knowledge, Mementos [21] was the first software solution for having forward progress in energy harvesting MCUs. At compile-time, it instruments trigger points at different program locations such as loop-latches (aka tail of back-edges), and function returns. These trigger points are calls to a function that estimates the available energy at run-time by comparing the capacitor's voltage with a predefined threshold with the help of an analog-to-digital converter (ADC). If the voltage is below the threshold, Mementos checkpoints volatile state of the system onto non-volatile memory. Otherwise, the system continues its normal execution. Mementos cannot always guarantee forward progress. For instance when one iteration of the loop body consumes more energy than maximum energy in the capacitor. A driving principle of Mementos was to "reason minimally about energy at compile time, maximally at run time", because even expert programmers are not reliable when reasoning about energy. Conversely, we propose to do all the work at compile time, but also keep programmers out of the loop and rely only on automatic static analysis tools. Also, Mementos probes ADC at run-time regularly which is costly and consumes additional energy of the system.
Like Mementos, Ratchet [22] is another compile-time checkpoint placement approach. It exploits the notion of idempotency[1] for creating restartable code sections. It places checkpoints at idempotent region boundaries. However, because of the limitation in alias analysis, the number of checkpoints might be more than what is needed. Ratchet, also only works with systems

[1] A piece of code is *idempotent* if repeated subsequent invocations do not modify the state of the machine.

which are configured with one unified non-volatile memory. In contrast, our work is portable and can work with any hardware regardless of the type of the memory.
Researchers have also presented task based programming models [17, 7, 18], where a programmer is responsible for decomposing the program into tasks that execute atomically. However, in these models, the programmer must be sure that a task's energy consumption does not exceed the maximum available energy in capacitor. Otherwise, the system would face the forward progress problem and would execute the same task repeatedly. To make sure that the application have forward progress, the programmer can act conservatively and place more task boundaries into the code results in wasting more time and energy. In summary, reasoning about the number and the size of tasks is painful and error-prone for programmers. The burden will be worst when it comes to changing the code or some features of the hardware such as capacitors as the programmer must reconstruct the whole process again.
A few prior works [4, 8, 1] also consider checkpoint placement by estimating energy. However, at some point in their work, they estimate energy by profiling or measurement techniques or they did not place checkpoint based on WCEC (worst-case energy consumption). As a result, in both cases, their approaches are not safe. In contrast, our work proposes safety by leveraging WCEC. In addition, our work does not require any extra hardware feature.
Recently, a series of solutions [3, 2, 13] requiring extra hardware support, try to improve the whole process of taking checkpoint. Although these approaches perform well as they take checkpoint when it is needed, they do not have the portability of software solutions.

3. WCEC-Aware Checkpoint Placement

For finding checkpoint locations in the program, we leveraged WCEC of program sections as we believe that the properties of WCEC can specify the number of checkpoints and their location. Generally in WCEC, the goal is to have a safe as well as a tight estimation on energy consumption of a program executing on a hardware. Safety means that the actual consumption must be less than, or equal to, the estimated upper-bound, regardless of program input. Tightness means that the estimation must be as close as possible to actual WCEC. Herein, the safety property of WCEC guarantees forward progress and program correctness. For the forward progress, the safety guarantees that the energy consumption for reaching the next checkpoint is less than or equal to the energy that a capacitor can provide, since checkpoints are placed based on WCEC with the distance of capacitor's maximum energy. For program correctness, we restrict the system to continue only when the capacitor is fully charged. After a checkpoint is taken, the system enters a sleep mode until the capacitor is fully charged again. Respectively, memory will be consistent. Also, herein, the tightness of WCEC relates to the number of checkpoints relative to the optimal number. The tighter the WCEC, the lower the number of unnecessary checkpoints.
Estimating WCEC statically necessitates to have a representation of the program as well as an energy model which reflects the energy consumption of the system. For the former the CFG of the program can represent complex structures in the program such as loops, conditions and function calls. For the latter, energy models at the lower levels of the software such as ISA are more accurate as they are closer to the hardware [9]. Figure 2 (a) shows a sub CFG of a program generated from the binary representation of the program. It contains eight basic blocks. The number beside each basic block indicates the amount of energy that the basic block consumes. This number is computed based on the energy model. For example, for simple architectures by adding the amount of energy each instruction within basic block consumes. In this CFG the estimated WCEC is 209 pJ which means that for executing this CFG the available energy in

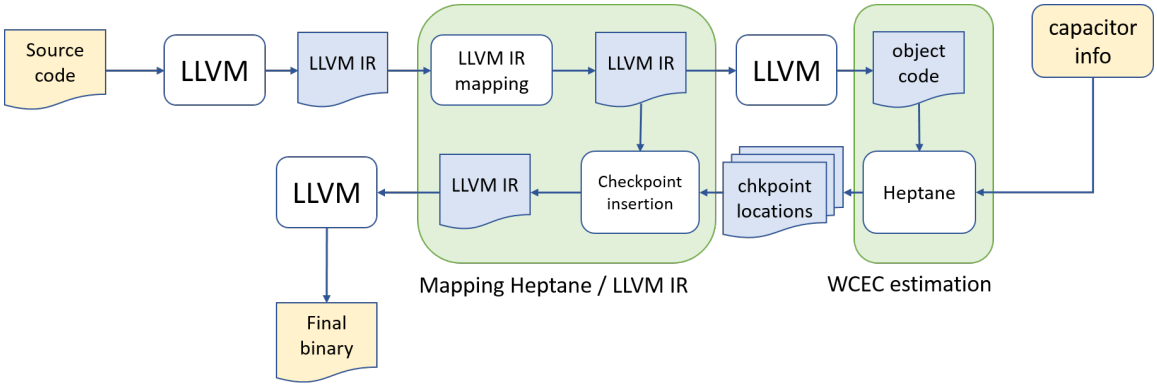


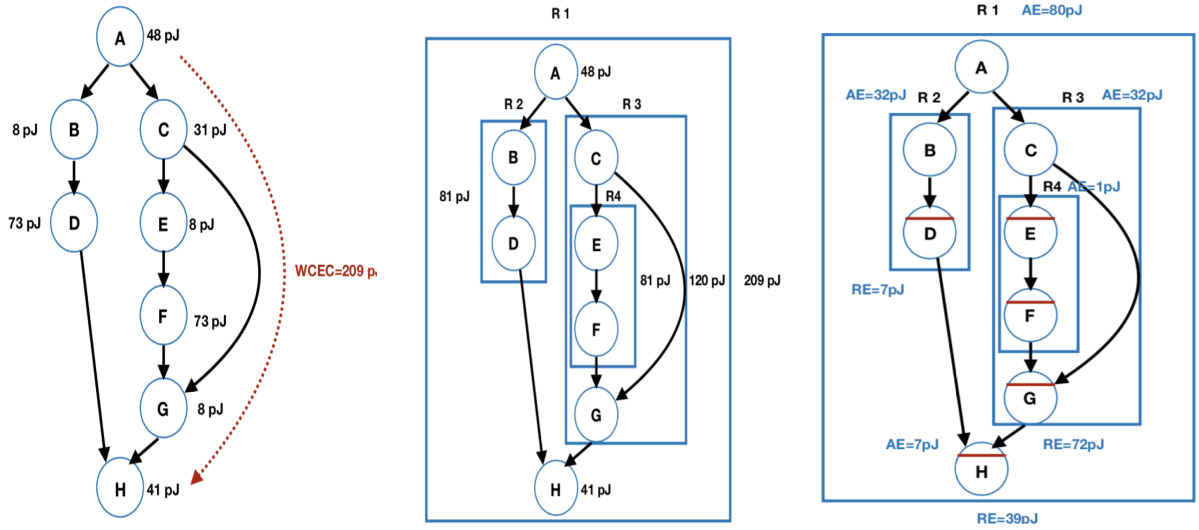
Figure 1: Overview of our flow

capacitor must not be less than 209 pJ. Otherwise, the system face power failures. In the reality, CFGs are big and the amount of estimated WCEC is always much bigger than the maximum energy a capacitor can provide. Also, due to the branches and loops, the number of paths from the start node to the end node is large. For instance, in the above mentioned simple CFG, the number of paths form node A to node H is three. This number gets bigger as the CFG gets bigger and more complex with branches and loops. Therefore, a method is needed to analyze the CFG and estimate the energy of all paths and place checkpoints when the WCEC exceeds the maximum amount of energy the capacitor can provide.

Our implementation consist of a component for estimating the WCEC (Heptane, see below), augmented with a checkpoint locating algorithm. Figure 1 shows the overview of our flow. In addition, since Heptane works in binary code and our final checkpoint placement is in LLVM IR [15], we adopted a mapping between Heptane and LLVM IR.

As shown, the input of the proposed tool-chain is the capacitor size and high-level C code annotated with loop bound information. It is worth noting that specifying loop-bound information is the only supplementary effort requested from the programmer. The output of our tool-chain is a binary code enriched with checkpoint trigger calls. Each checkpoint trigger is a call to a run-time library which is responsible for checkpointing the volatile state of the program into the non-volatile memory. The overhead and the energy cost of checkpointing itself is highly dependent on the underlying architecture. For architecture with non-volatile memory as unified memory, the cost of checkpointing is almost constant since only CPU registers must be copied. However, for systems configured with a volatile memory such as SRAM as well as a type of non-volatile memory, the cost of checkpointing is variable and dependent on program state such as the size of stack and heap, as well as the amount of live data at the time of checkpointing. In the latter case, we need to guarantee we have enough energy to perform the checkpointing in the worst case, and the location of the checkpoint matters. In the worst case the system must have enough energy to checkpoint all volatile memory. Our work can work with both types of architecture. However, in this work, we assume that there is always enough energy available for checkpointing a constant number of CPU registers in the former case or all volatile memory in the latter case , and we focus on the placement of checkpoints that guarantees correctness, and forward progress.

In the rest of this section, we explain the WCEC estimation (Heptane), our checkpoint locating which is added to Heptane as well as the mapping between Heptane and LLVM IR



(a) input CFG (8 blocks) (b) computed SESE regions (c) checkpoint locations

Figure 2: Input CFG, computed Single-Entry Single-Exit regions, and selected checkpoint locations (red lines, AE=available energy, RE=remaining energy)

3.1. WCEC Estimation

For the WCEC part, we used Heptane [12] which is a tool originally for estimating worst-case execution time (WCET) [25]. Heptane's functionality is divided into two separated components. The former is for generating control-flow graph (CFG) of the program from the object code. The latter performs two types of analysis on the generated CFG: high-level analysis and low-level analysis. The low-level analysis compute an upper-bound for each basic block in CFG by considering the cost of instructions as well as features related to micro-architecture such as cache and pipeline. Then, the high-level analysis can compute the whole program's WCET by performing Implicit Path Enumeration Technique (IPET) [16] which is based on Integer Linear Programming (ILP) formulation of the WCET calculation problem.

In this work, since our concern is energy, inspired by Wägemann et al. [23], an energy cost for each instruction is specified instead of cycle cost that Heptane considers. Due to the simplicity of the processors in the domain, that is processors without caches and branch prediction, applying complex analysis in Heptane is not necessary. Also, herein, the goal of work is not the WCEC of the whole program; instead we want to fragment the program into code sections which can be executed in one life cycle when the capacitor is fully charged. These code sections are bounded by checkpoint trigger calls. As a result, the location of these checkpoint trigger calls in the program must be identified.

3.2. Checkpoint Locating

For identifying checkpoint locations, we adopted an algorithm based on Single-Entry-Single-Exit (SESE) regions [14]. SESE regions have a single node as the entry of the region as well as a node as the exit node of the region. As such, they provide convenient placeholders for checkpoints. Also, these regions can be nested, sequentially composed or disjoint (Figure 2 (b)). The biggest SESE region is the CFG itself as it has one start node and one end node. The smallest SESE regions are basic blocks and instructions. In this work, since our granularity for checkpoint placement is basic blocks, we chose the basic block as the smallest SESE region.

The input of the algorithm (see Alg. 1 in appendix) is the CFG of the program with its corresponding SESE regions, and the capacitor size. The algorithm starts by estimating the WCEC of the biggest region and if the estimated WCEC is bigger than the available energy, it recursively estimates energy for all nested regions. For estimating the energy of region, we used partial WCET estimation (δ-WCET) proposed by Bouziane et al. [6, 5]. However, for the sake of clarity in Algorithm 1, we used (δ-WCEC) notion as here the output of the aforementioned work is partial energy consumption. For instance, in the CFG of Figure 2 (b), assume the capacitor can store 80 pJ, the algorithm first estimate the energy of region R1. Since the estimated value is bigger than 80pJ, it recursively estimates the energy of R2 and R3 with the available energy of 32pJ (80pJ - 48pJ). It continues until it reaches the smallest SESE regions which has no nesting region (basic block) and it places a checkpoint at the beginning of that basic block. The algorithm returns the amount of remaining energy. When it places a checkpoint, the return value will be the maximum amount of energy in the capacitor subtracted by the energy consumption of the basic block. It is worth noting that if a basic block consumes more energy than the maximum amount of energy in the capacitor, that basic block could easily be broken into smaller ones. However, in this work we assume that the energy consumption of each basic block is always less than the capacitor size. Figure 2 (c) shows the CFG with located checkpoints. Each located checkpoint has a corresponding LLVM IR line number which is the final output of the algorithm and WCEC section.

3.3. Mapping Between Heptane and LLVM IR

Since the analysis part is in binary code and placing checkpoint trigger calls is in LLVM IR, we leveraged a mapping between LLVM IR and binary code by using Debug Information and Source Location Information inspired from Grech et al. [10]. We created two LLVM passes. ***Source Line to LLVM IR*** traverses the LLVM IR and replaces Source Location Information with LLVM IR location information. After this pass, the binary code is generated and given to Heptane. As mentioned, the output of Heptane is a series of line numbers which specify where to place checkpoint triggers in LLVM IR. The ***Checkpoint placement*** is responsible to place checkpoint triggers based on the line numbers that Heptane produces. After this pass, the final binary code is generated and the program is ready to be executed on energy harvesting device.

4. Evaluation

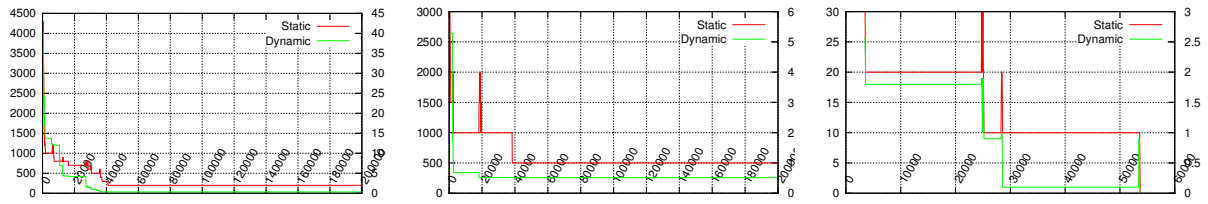


Figure 3: Static and Dynamic number of Checkpoints. The x-axis represents the capacitor size (in pJ). The left y-axis reports the dynamic (taken) number of checkpoints. The right y-axis reports the static (inserted) number of checkpoints.

For this part, we have chosen NDES which is a complex embedded code, Fast Discrete Cosine Transform (FDCT) and Cyclic Redundancy Check (CRC) from Mälardalen suite [11]. They are highly used in embedded systems and sensors. In addition, in terms of CFG complexity, they contain loops, inner-loops and function calls. Therefore, they can show the effectiveness and correctness of the proposed checkpoint placement strategy. Also, they perform a special amount of computation in the main MCU core, as in this work we are just focusing on the main CPU computation. We assigned an energy cost to each instruction for ARMv6-m ISA, derived from an actual core synthesized in 28 nm ST FDSOI. This ISA is used for a number of processors such as ARM Cortex-M0+ in low-power domains. We run the final executable generated by our tool-chain in a modified version of a cycle-accurate simulator for the mentioned ISA [22].

To show the sensitivity of our approach, we tested several benchmarks with different capacitor sizes, selected with respect to the overall WCEC of the benchmark. In Figure 3, the red line shows the number of static trigger calls inserted at compile-time; the green line shows the number of taken (executed) checkpoints at run-time (Figure 3). As expected, our strategy is sensitive to capacitor size, and the number of taken checkpoints decreases when the capacitor size increases. However, we also observe long plateaus where the number of checkpoints remains constant for a wide range of capacitor sizes (e.g. CRC between 1000 pJ and 2100 pJ). The main reason for that is the presence of loops. As long the execution of the entire loop (with worst-case trip count) requires more energy than the capacitor can provide, a checkpoint must be placed inside the loop body. Also, in some cases, because LLVM IR instructions are sometimes coarser than assembly instructions, our tool-chain cannot place the checkpoints outside the loop for a special capacitor sizes even though with a full charge of capacitor, it is possible to process the loop. This explains the occasional peaks of the curves. In the future, we will consider loop optimizations such as unrolling to reduce the number of checkpoints.

Our work in comparison to related work, namely Mementos [21] from Ransford et al. can guarantee forward progress and program correctness. Similar to our work, Mementos inserts checkpoints in the CFG. However, they only considered specific locations and opted for loop latches or function returns. Mementos checks the remaining energy (actually the voltage as a proxy for energy) only at these predefined locations. In case a loop body or a function (without loop) requires more that the capacitor provides, they cannot prevent unprotected energy depletion, and thus cannot guarantee forward progress. Also , if Mementos had the ability to modify the none-volatile memory by the program semantic, a failure may corrupt the memory and cause the program to be incorrect. In comparison to Ratchet [22], which checkpoints between every WAR dependence, our work is more efficient in terms of number of both static and dynamic checkpoints. For instance, Ratchet is forced to insert checkpoints in the example provided in this paper [1] no matter how much energy is in the capacitor size. But, as our checkpoint placement is sensitive to the capacitor size, it can place checkpoints outside the loop whenever it is possible to process a loop with one full charge of capacitor.

5. Conclusion

In this paper we introduced a compile-time checkpoint placement strategy for energy harvesting system. Our approach can guarantee to have program correctness and forward progress simultaneously. To achieve this, our toolchain inserts checkpoint trigger calls based on worst-case energy consumption of program sections. In addition, our work requires a negligible extra programming effort as well as no extra hardware support.

Bibliographie

1. Baghsorkhi (S. S.) et Margiolas (C.). – Automating efficient variable-grained resiliency for low-power IoT systems. – In *International Symposium on Code Generation and Optimization*, pp. 38–49. ACM, 2018.
2. Balsamo (D.), Weddell (A. S.), Das (A.), Arreola (A. R.), Brunelli (D.), Al-Hashimi (B. M.), Merrett (G. V.) et Benini (L.). – Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, n12, 2016, pp. 1968–1980.
3. Balsamo (D.), Weddell (A. S.), Merrett (G. V.), Al-Hashimi (B. M.), Brunelli (D.) et Benini (L.). – Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, vol. 7, n1, 2015, pp. 15–18.
4. Bhatti (N. A.) et Mottola (L.). – HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. – In *16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 209–220. IEEE, 2017.
5. Bouziane (R.), Rohou (E.) et Gamatié (A.). – Energy-efficient memory mappings based on partial WCET analysis and multi-retention time STT-RAM. – In *26th International Conference on Real-Time Networks and Systems (RTNS)*, pp. 1–11, 2018.
6. Bouziane (R.), Rohou (E.) et Gamatié (A.). – Partial worst-case execution time analysis. – In *ComPAS: Conférence d'informatique en Parallélisme, Architecture et Système*, 2018.
7. Colin (A.) et Lucia (B.). – Chain: tasks and channels for reliable intermittent programs. – In *ACM SIGPLAN Notices*volume 51, pp. 514–530. ACM, 2016.
8. Colin (A.) et Lucia (B.). – Termination checking and task decomposition for task-based intermittent programs. – In *27th International Conference on Compiler Construction*, pp. 116–127. ACM, 2018.
9. Georgiou (K.), Xavier-de Souza (S.) et Eder (K.). – The IoT energy challenge: A software perspective. *IEEE Embedded Systems Letters*, vol. 10, n3, 2018, pp. 53–56.
10. Grech (N.), Georgiou (K.), Pallister (J.), Kerrison (S.) et Eder (K.). – Static energy consumption analysis of LLVM IR programs. *Computing Research Repository, arXiv*, 2014, pp. 1–12.
11. Gustafsson (J.), Betts (A.), Ermedahl (A.) et Lisper (B.). – The Mälardalen WCET benchmarks: Past, present and future. – In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
12. Hardy (D.), Rouxel (B.) et Puaut (I.). – The Heptane static worst-case execution time estimation tool. – In *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pp. 8:12–8:12, 2017.
13. Jayakumar (H.), Raha (A.) et Raghunathan (V.). – QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. – In *27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*, pp. 330–335. IEEE, 2014.
14. Johnson (R.), Pearson (D.) et Pingali (K.). – The program structure tree: Computing control regions in linear time. – In *ACM SigPlan Notices*volume 29, pp. 171–185. ACM, 1994.
15. Lattner (C.) et Adve (V.). – LLVM: A compilation framework for lifelong program analysis & transformation. – In *International symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75. IEEE Computer Society, 2004.
16. Li (Y.-T. S.) et Malik (S.). – Performance analysis of embedded software using implicit path enumeration. – In *ACM SIGPLAN Notices*volume 30, pp. 88–98. ACM, 1995.
17. Lucia (B.) et Ransford (B.). – A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, vol. 50, n6, 2015, pp. 575–585.
18. Maeng (K.), Colin (A.) et Lucia (B.). – Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 1, 2017, p. 96.
19. Raghunathan (V.), Kansal (A.), Hsu (J.), Friedman (J.) et Srivastava (M.). – Design considerations for solar energy harvesting wireless embedded systems. – In *Proceedings of the 4th international symposium on Information processing in sensor networks*, p. 64. IEEE Press, 2005.
20. Ransford (B.) et Lucia (B.). – Nonvolatile memory is a broken time machine. – In *Workshop on Memory Systems Performance and Correctness*, p. 5. ACM, 2014.
21. Ransford (B.), Sorber (J.) et Fu (K.). – Mementos: System support for long-running computation on

RFID-scale devices. – In *ACM SIGARCH Computer Architecture News*volume 39, pp. 159–170. ACM, 2011.
22. Van Der Woude (J.) et Hicks (M.). – Intermittent computation without hardware support or programmer intervention. – In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 17–32, 2016.
23. Wägemann (P.), Distler (T.), Hönig (T.), Janker (H.), Kapitza (R.) et Schröder-Preikschat (W.). – Worst-case energy consumption analysis for energy-constrained embedded systems. – In *27th Euromicro Conference on Real-Time Systems*, pp. 105–114. IEEE, 2015.
24. Weimer (M. A.), Paing (T. S.) et Zane (R. A.). – Remote area wind energy harvesting for low-power autonomous sensors. – In *37th IEEE Power Electronics Specialists Conference*, pp. 1–5. IEEE, 2006.
25. Wilhelm (R.), Engblom (J.), Ermedahl (A.), Holsti (N.), Thesing (S.), Whalley (D.), Bernat (G.), Ferdinand (C.), Heckmann (R.), Mitra (T.) et al. – The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, n3, 2008, p. 36.

6. Appendix

Checkpoint Locating Algorithm

The function *IdentifyChKLocations* identifies checkpoint locations in a region and stores the line number where the checkpoint must be placed into *CheckpointLocations* vector. The input of the function is the CFG of the program with its corresponding identified SESE regions as well as the capacitor information. The function returns the remaining energy of the region it processes. The algorithm begins from the outermost region (Line 1). The energy of the region is estimated (Line 2). If the estimated WCEC is more than the available energy in the capacitor, the algorithm subtracts the energy of the entry node of the region (Line 3) and recursively calls the *IdentifyChKLocations* for all nesting region of the region (Line 4 and Line 5). It considers the minimum amount of remaining energy among all nested region. However, if the region does not have any nested region (a basic block), the algorithm identifies that region (the basic block) as a checkpoint location and stores the line number of the first instruction of the basic block into *CheckpointLocations* vector (Line 6). Since after each checkpoint the capacitor must be fully charged, the energy consumed by the basic block is subtracted by the maximum amount of energy in capacitor and considered as the remaining energy.

Algorithm 1 Checkpoint Locating Algorithm

Data: CFG with Identified SESE Regions
Result: Checkpoint Locations
C is the energy of capacitor
CheckpointLocations is a vector of line numbers
r is the outermost region

```
1  IdentifyChKLocations(r,C)
   function IDENTIFYCHKLOCATIONS(SESERegion R, AvailableEnergy E)
     2  e=δ-WCEC(Entry node of R,Exit node of R)
        if e > E then
           if R has nesting regions then
     3        E=E-( Energy consumption of Entry node of R )
     4        for All Region N_i Nested in R do
     5           remainingEnergy=min_i(IDENTIFYCHKLOCATIONS(N,E))
              end
           else
     6        remainingEnergy=C - e, Add this Location to CheckpointLocations
           end
        else
           remainingEnergy=E - e
        end
        return remainingenergy
   end function
```

Additional Benchmarks

Here is some additional benchmarks from Mälardalen suite [11], we evaluated our tool-chain with. As it is observed, the number of dynamic checkpoints at run-time is decreasing as the capacitor size is increasing. Also, as you can see, sometimes when the number of static checkpoints is increased , the number of dynamic checkpoints is decreased. The reason for that is when it is possible to process the whole loop with a full charge of capacitor size, our algorithm places two checkpoints right before and after loop instead of placing the checkpoint inside the loop. The first checkpoint is to have energy for processing loop and the second checkpoint is to have energy for continuing the rest of the code. The number of static checkpoints only affect on the code size. However, each checkpoint that it is taken at run-time consumes time and energy. So, it is worthy to increase the number of static checkpoints whenever it is possible to decrease the number of dynamic ones. For minmax, when the capacitor size is increased from 800 pJ, an increase in the number of static and dynamic checkpoint is observed. This is because our algorithm is biased to place checkpoints before a function call as a function might have more than one context (call site). However, in the case of minmax, all functions have only one context and it is better to process the function and place the checkpoint when it is necessary. In the future we will consider the number of contexts of a function and improve the number of checkpoints.

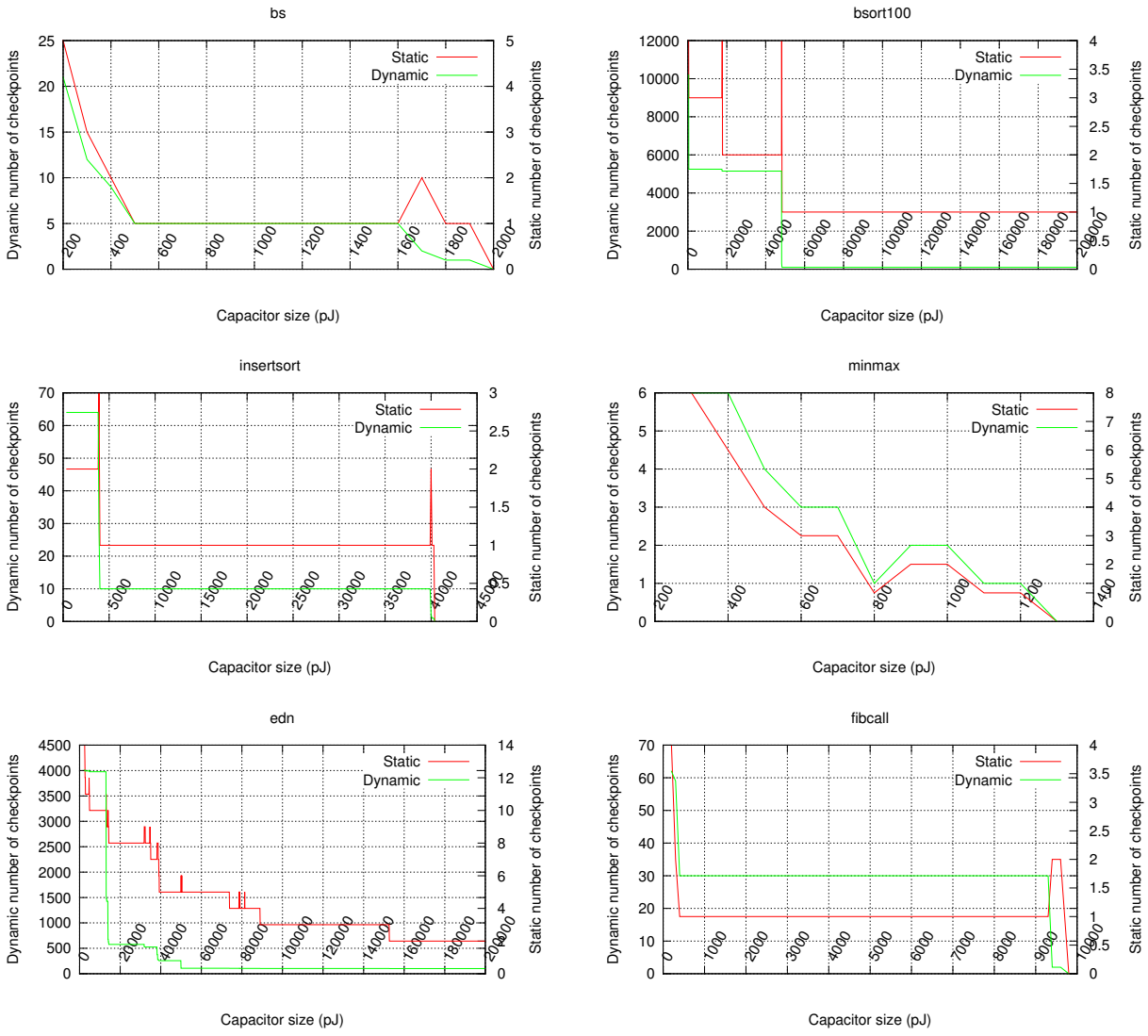


Figure 4: Additional Benchmarks