Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems

Bahram Yarahmadi, Erven Rohou

Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems

Bahram Yarahmadi[1] and Erven Rohou[1][0000−0002−8060−8360]

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

Abstract. 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 statically-computed worst-case energy consumption of code sections. We also apply classical compiler optimizations in order to decrease the required number of checkpoints at runtime. As our method is based upon worst-case energy consumption, we can guarantee memory consistency and forward progress.

Keywords: worst-case energy consumption · static analysis · energy harvesting · checkpoint

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) 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. This is a unique bug in intermittent computation jargon which is also called facing non-termination. 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 [21] 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 the following:

- we propose a static, automatic, compiler-based technique for insertion of checkpoints that guarantees correctness and termination of a program on an intermittently powered system;
- we leverage compiler optimizations to reduce the number of checkpoints;
- we limit the burden on programmers to a negligible additional effort;
- we provide a portable solution not requiring any extra hardware support.

Section 2 is an overview of related work. Section 3 presents our method. We evaluate it in Section 4. We conclude in Section 5.

2 Related work

Researchers have proposed software-only, joint hardware/software as well as fully hardware solutions for having forward progress as well as program correctness in energy harvesting systems. In this section, we will discuss them briefly.

To the best of our knowledge, Mementos [22] was the first solution addressing forward progress on a harvesting MCU. At compile-time, it inserts 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. 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.

Ratchet [23] inserts checkpoints at compile-time. It exploits the notion of idempotency[1] for creating restartable code sections. It places checkpoints at idempotent region boundaries. However, because of limitations in alias analysis, the number of checkpoints might be more than needed. Ratchet only works with systems with one unified non-volatile memory. In contrast, our work is portable and works with any hardware regardless of the type of the memory.

Researchers have also presented task based programming models [19, 8, 20], 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 [5, 9, 2] 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 insert checkpoints 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. As far as we know, there exists only one work [25] which proposed checkpointing based on WCEC. The work is a runtime kernel which schedules tasks based on the estimated WCEC. Our work differs from it in a way that it is based on the WCEC of program sections and the control-flow graph of the program. Also, our work applies classical compiler optimization in order to decrease the number of checkpoints.

Recently, a series of solutions [4, 3, 14] 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. In addition, adding or using a hardware feature increases the energy consumption of the MCU.

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

3 WCEC Aware Checkpoint Placement

3.1 Technical Background

Determining checkpoint locations is based on the control flow graph (CFG) of the program. It is similar to the computation on worst-case execution time (WCET) estimates in the real-time domain [26].

The goal is to have a safe as well as tight estimation of the energy consumption of a program executing on the hardware. Safety means that the actual consumption must be less than, or equal to, the estimate, regardless of program input and dynamic events. 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, safety guarantees that the energy consumption for reaching the next checkpoint is less than or equal to the energy a capacitor can provide, since checkpoints are placed based on WCEC with the distance of capacitor's maximum energy. We restrict the system to resume execution after a checkpoint only when the capacitor is full (it may enter a low-power mode for better efficiency). This way, we ensure that when the system wakes up from a checkpoint, it reaches the next one, where it waits for the capacitor the recharge.

This property also helps coping with the aforementioned memory consistency issue:

as re-execution do not occur, we avoid problems related to memory inconsistency or replaying side effects [21]. With the exception of the checkpoints and resuming code, the application follows its normal control flow.

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 represents complex structures 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 [10]. 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 besides each block indicates the worst-case amount of energy that the basic block consumes, computed based on the energy model (e.g. for simple architectures, by adding the amount of energy each instruction of the block consumes). In this CFG, the estimated worst-case consumption is 209 pJ.

For real-life applications, CFGs are large and the overall estimated WCEC is always much larger 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 from node A to node H is three. This number increases as the CFG gets larger and more complex with branches and loops.

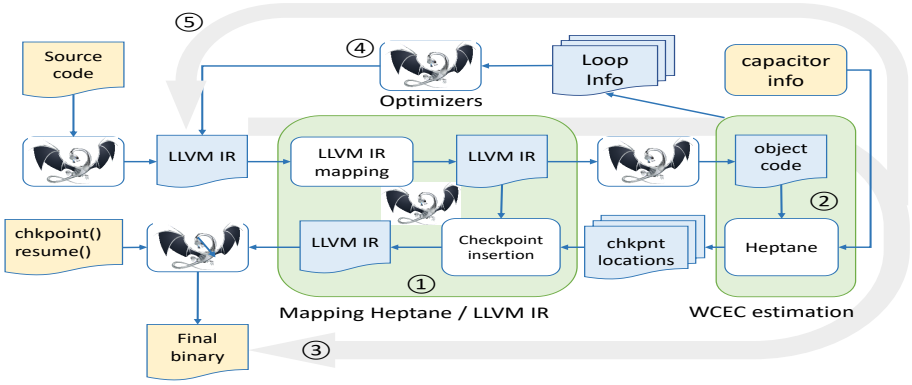


Fig. 1. Overview of our flow

As typical in the real-time domain, interrupts and preemption cannot be handled at this level. If present, they should be handled in a distinct part of the system runtime.

3.2 Approach

Problem Statement. For any given capacitor size (i.e. energy amount), our approach consists in inserting checkpoints at various places in the program such that, from any checkpoint, another checkpoint can be reached, regardless of the dynamic path taken by the application.

Solution. To make the problem tractable, we adopted an algorithm based on Single-Entry-Single-Exit (SESE) regions [15]. SESE regions may be complex, but they have a single well-defined entry as well as a single well-defined exit node. As such, they provide convenient placeholders for checkpoints. Also, these regions can be nested, sequentially composed or disjoint (see Figure 2 (b)). The largest SESE region is the CFG itself as it has one start node and one end node[2]. The smallest SESE regions are basic blocks and instructions. In our 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 Algorithm 1) is the CFG of the program with its corresponding SESE regions, and the capacitor size. The algorithm starts by estimating the WCEC of the outermost region and if the estimated WCEC is bigger than the available energy, it recursively analyzes the nested regions.

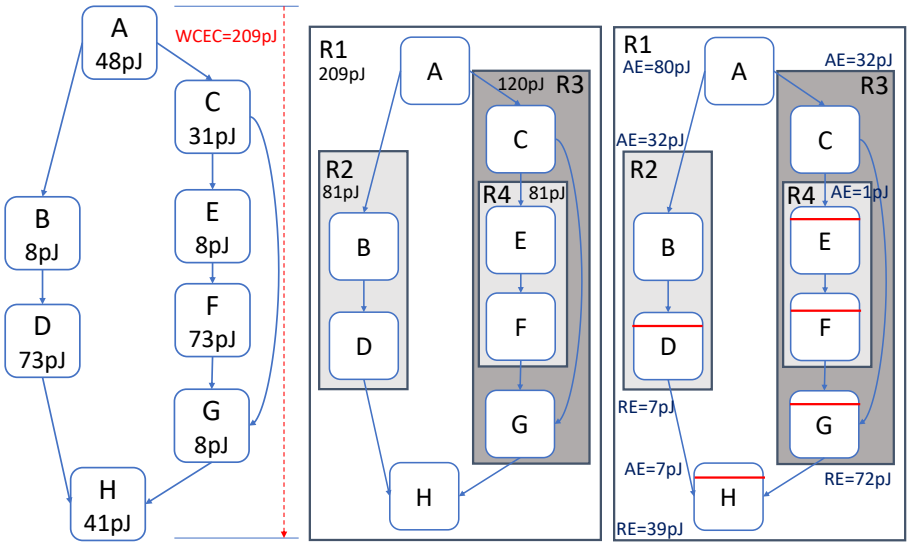
[2] Even in the presence of multiple return statements in, a function, a compiler can easily create a new block and add edges to it to guarantee a single exit node.

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

For estimating the energy of a region, we rely on partial WCET estimation (δ-WCET) proposed by Bouziane et al. [7, 6]. However, for the sake of clarity in Algorithm 1, we used energy consumption in lieu of execution time wherever relevant. For instance, consider the CFG of Figure 2 (b), and assume the capacitor can store 80 pJ. The algorithm first estimates the energy of region R1. Since the estimated value exceeds 80 pJ, it recursively estimates the energy of R2 and R3 after subtracting the cost of block A (remaining energy of 32 pJ = 80 pJ - 48 pJ). It continues until it reaches the innermost SESE regions (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 Figure 2 (c) shows the CFG with inserted checkpoints.

As shown, the input of the proposed toolchain is the capacitor size and high-level C code annotated with maximum loop bound information. It is worth noting that specifying this information is the only supplementary effort requested from the programmer, and it is typical of embedded real-time systems. 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



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

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

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 solution can deal with both types of architectures. 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.

3.3 WCEC Estimation

Our implementation consists of a component for estimating the WCEC (Heptane [13], see Section 3.3), augmented with a checkpoint insertion algorithm.

Figure 1 shows the overview of our flow. The first component (box ① in the figure) is LLVM augmented by a few passes. It first builds a mapping between LLVM IR and binary code (see Section 3.4), compiles the code and invokes our WCEC estimation tool. This tool, called Heptane (box ② in the figure) determines where checkpoints should be located based on information about the

capacitor and a power model of the architecture. This information is fed back to LLVM for actual insertion of the checkpoints and production of the binary. This process is highlighted by the arrow designated as ③. We also explored how loop optimizations can be leveraged to decrease the cost of checkpoints (Section 3.6). In this case, Heptane selects appropriate loop optimizations that are forwarded to LLVM (box ④). This results in a new version of the program in LLVM IR (arrow designated as ⑤). The process of arrow ③ is repeated.

In addition, since Heptane works in binary code and our final checkpoint placement is in LLVM IR [16], we added a mapping between Heptane and LLVM IR described in Section 3.4.

Heptane [13] is a tool originally developed for estimating worst-case execution time (WCET) [26]. Heptane's functionality is divided into two separated components: `Heptane Extract` and `Heptane Analysis`. 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) [18] 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. [24], an energy cost for each instruction is specified instead of cycle cost that Heptane normally 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. However, our approach is general and can include more complex architectures as long as an accurate (and safe) energy model is provided by manufacturer. 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.

3.4 Mapping Between Heptane and LLVM IR

Since the analysis part is performed on binary code, and placing checkpoint trigger calls is applied at LLVM IR level, we need a mapping between the two. To achieve this, inspired from Grech et al. [11], we "hijacked" the debug information mechanism that propagates source-location information into a binary: at LLVM-level, we replace source location by the line number in the LLVM representation. The regular toolchain processes it as usual, carrying our information instead of traditional source line numbers.

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.

3.5 Coping with non-termination

In our work, coping with non-termination and guaranteeing forward progress through the execution of programs is straightforward. In the WCEC analysis part, Heptane checks for every basic block whether its energy consumption exceeds the capacitor energy when it is fully charged. If a basic block consumes more energy than the maximum amount of energy in the capacitor, that basic block can easily be broken into smaller ones by the compiler. For that, Heptane reports the line numbers of the basic blocks that are too long. LLVM subsequently splits these blocks and applies LLVM IR mapping pass again. It is worth noting that this process could also be done at assembly level. However, herein for the sake of simplicity we preferred to do at LLVM IR level.

3.6 WCEC-Aware Compiler Transformations and Optimizations

Compiler optimizations heavily modify the structure of programs, hence the locations of checkpoints. As shown in the experimental section, optimizations are vital to decrease the amount of checkpointing at runtime. Our objective here is not to provide an exhaustive study of the impact of every optimizations, but rather to demonstrate the benefit of selected optimizations to the energy management of intermittently powered systems. As loops are the most time and energy consuming part of programs, we have chosen optimizations on loops. Also, placing checkpoints in loops can increase the energy consumption of the execution, making it critical to process loops with care.

Clearly, not all optimizations can be applied to all loops, since data dependences must be checked to guarantee program semantics is preserved. In this section, we briefly introduce the optimizations that we have selected. An exhaustive survey of compiler optimizations, their applicability and advantages can be found in Bacon et al. [1].

Conditional checkpoints is the simplest optimization that our toolchain applies, making checkpoints conditional on the loop iteration number. This is advantageous when a full charge of the capacitor is not enough to execute the entire loop, but we can prove that it can execute several iterations. Checkpointing at each iteration would be inefficient. This is illustrated on Figure 3 (a) and (b) respectively for the original loop, and the transformation, assuming we can execute three iterations of the loop with a single charge. This is always correct. Note that the same effect would be achieved by unrolling the loop three times and inserting a single checkpoint. Unrolling, however, increases code size and it is rarely applied on small systems with limited amount of memory.

Loop splitting divides the iteration space in two pieces (or more) with the same loop body. An example is given in Figure 3 (c). The effect is similar to conditional checkpoints, but it saves the cost of the condition, at the expense of duplicating the loop body.

Each generated loop can be executed in one charge of capacitor. Splitting is always legal.

Loop fission (aka loop distribution) also divides a loop into pieces, but takes statements apart. See the example on Figure 3 (d). Depending on the energy cost of the statements (S1 and S2 in the figure) and the value of n, it may offer more interesting tradeoffs. In this optimization new loops have less energy consumption and can be processed in one charge of capacitor. The checkpoint trigger call is also between two loops. Fission is not always legal, and depends on the type of dependence between S1 and S2, if any.

Loop interchange consists in swapping two loops in a loop nest (see Figure 3 (e) and (f) respectively for original and transformed loop nests). This is profitable when $n < m$ and the body of the nested loop can be executed m times with a single charge. This reduces the number of taken checkpoints from m to n. Interchange is not always legal, depending on dependences. It also impacts locality, however small intermittently powered devices typically do not feature a data cache. In case they do, the reduction in the number of checkpoints should be balanced with the overhead in cache misses.

Optimizations are not hindered by checkpoint insertion which is applied later in the compilation flow. It is worth noting that each optimization affects the estimated WCEC. For instance, in conditional checkpoints, a few instructions are being added which alter the WCEC. For preserving the safety of the WCEC, we run Heptane for the second time after each optimization. Also, one complication derives from the fact that loop bounds must be statically known for the tools to estimate a worst-case execution time/energy consumption. This is typically done by manually adding annotations to loops in source code, and disabling compiler optimizations to guarantee that the CFGs at source and binary levels match. Tracing loop bounds through the entire compilation flow is complex. We address a simpler problem, focusing particular loops and applying a limited repertoire of transformations. For example, the bounds of a split loop are obtained by dividing the original bound by two. Fission creates a new loop with the same bound. In the case of interchange, bounds must be interchanged as well. For a larger set of optimization, careful tracking of annotations must be enforced. However, previous work by Li et al. [17] has shown the feasibility.

To preserve the safety of the WCEC estimation, after each optimization, the WCEC estimation part of our tool-chain checks that with the insertion of the checkpoint and the transformation the new WCEC provides safety. When the safety is approved with the WCEC part, the generated binary code can be executed on the device.

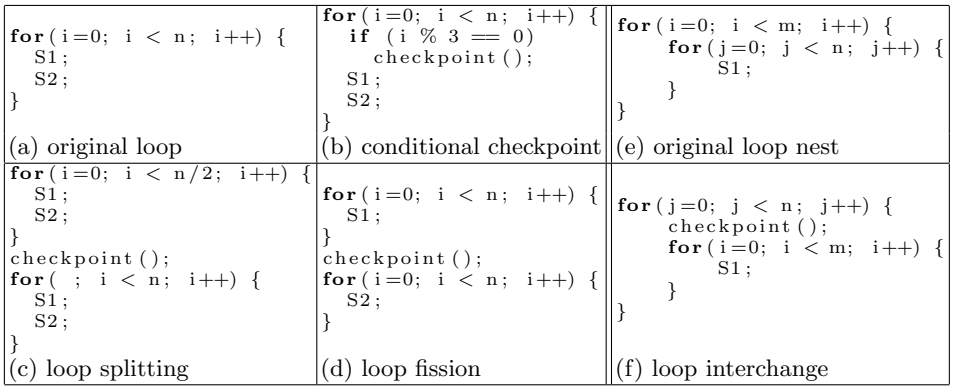


Fig. 3. Loop optimizations

Benchmark	Description
bs	Binary search for the array of 15 integer elements.
bsort100	Bubblesort program.
crc	Cyclic redundancy check computation on 40 bytes of data.
edn	Finite Impulse Response (FIR) filter calculations.
fdct	Fast Discrete Cosine Transform.
fibcall	Iterative Fibonacci, used to calculate fib(30).
insertsort	Insertion sort on a reversed array of size 10.
minmax	Simple program with infeasible paths and without loops.
ndes	Complex embedded code. Bit manipulation, shifts, array and matrix calculations
mm	rectangular matrix multiplication

Table 1. Benchmarks used for the evaluation

4 Evaluation

Settings. 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 [23].

Benchmarks. We selected nine benchmarks (Table 1) from the Mälardalen suite [12]. They are highly used in research publications on 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 significant amount of computation in the main MCU core, as this work is only focusing on the main CPU computation. We added a matrix multiplication we developed ourselves to experiment with cases not covered in the suite.

Experiments. To show the sensitivity of our approach, we tested several benchmarks with a wide range of capacitor sizes, selected with respect to the overall

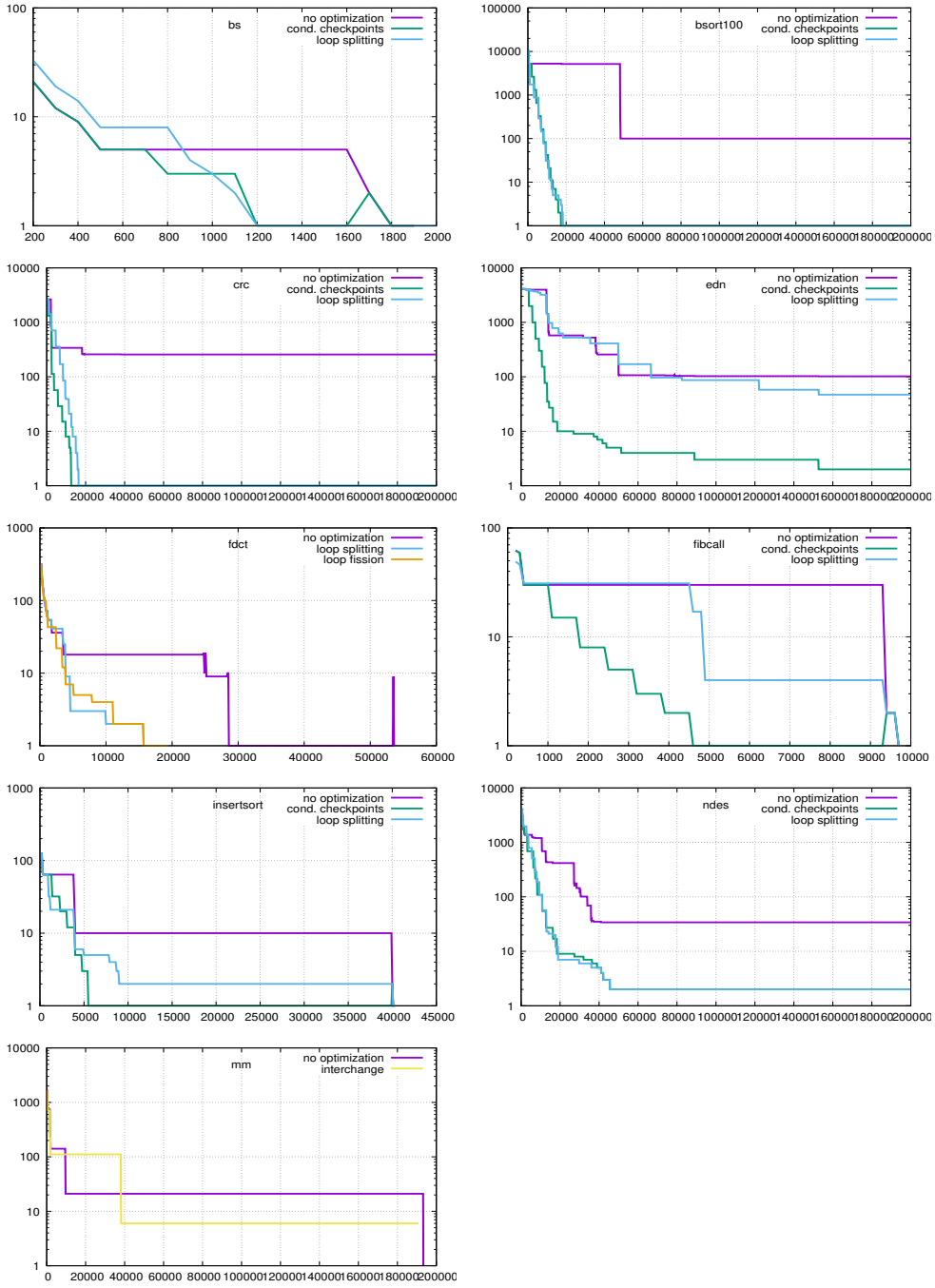


Fig. 4. Number of taken checkpoints when the capacitor size changes, without and with optimizations. The x-axis represents the capacitor size in picojoules (pJ). The y-axis represents the number of checkpoints taken at runtime (logscale).

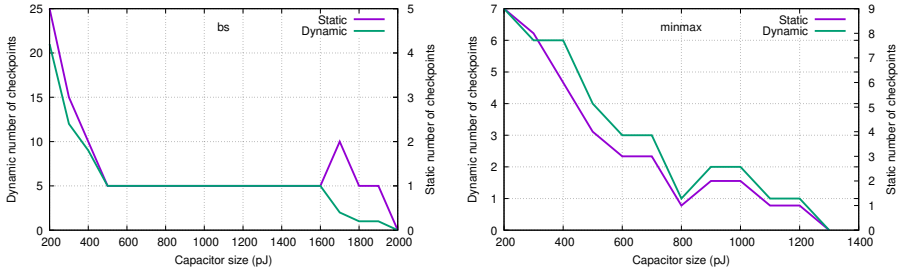


Fig. 5. Evolution of the dynamic (left axis) and static (right axis) numbers of checkpoints. No optimization is applied.

WCEC of the benchmark. We apply our algorithm to insert checkpoints, run the program in the simulator, and report the number of checkpoints taken during execution. On Figure 4, the bold purple line shows the number of checkpoints in the absence of optimizations, while other lines show the impact of individual optimizations. minmax is not reported here because it contains no loop, we discuss is below.

We explored with very small capacitor sizes, such as 200 pJ, which may not be realistic for deployed IoT systems [3]. The reasons are: 1) the Mälardalen benchmarks have rather small execution times and we still wanted to experiment with a wide range of values; 2) we wanted to test the scalability of our approach towards small as well as large values.

Number of executed checkpoints. 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. bs between 500 pJ and 1600 pJ). The main reason for that is the presence of loops. As long as 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. In some cases, it also happens that our toolchain cannot place the checkpoints exactly where Heptane suggested (i.e. just outside the loop) because LLVM IR instructions are sometimes slightly coarser than assembly instructions. This explains the occasional peaks of the curves.

Our work in comparison to related work, namely Mementos [22] 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)

[3] The available energy can be related to the capacitance C through the formula $E = \frac{1}{2}C(V_{start}^2 - V_{stop}^2)$, where V_{start} and V_{stop} represent voltages when the capacitor is full, and when the system is forced to stop because the voltage is too low.

requires more energy than the capacitor can provide, they cannot prevent unprotected energy depletion, and thus cannot guarantee forward progress. Also, if Mementos had the ability to modify the non-volatile memory by the program semantic, a failure may corrupt the memory and cause the program to be incorrect. In comparison to Ratchet [23], 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 [2] 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.

Impact of optimizations. Our results show that optimizations have the capability to reduce the number of necessary checkpoints.

First, making checkpoints conditional is always beneficial, and this can have dramatic impact. For example, unoptimized fibcall requires 30 checkpoints to complete execution, whereas conditional checkpoints can progressively reduce this number down to 15, then 8, and so on. The same behavior is visible in all benchmarks.

Loop splitting is also very effective, and in most cases similar to conditional checkpointing. When they differ, splitting is usually underperforming. This is visible in bs for small capacitor sizes, as well as fibcall and crc. Yet, in some occasions, splitting slightly outperforms conditional checkpointing: particular values in bs, or ndes between 20,000 pJ and 40,000 pJ.

Fission was applicable to fdct. It is similar in effect to splitting, slightly better or slightly worse, depending on values.

Interchange was not successfully applied to our set of benchmarks. This is why we added a simple matrix multiplication example. Our results show that interchange can reduce significantly the number of checkpoints for 141 to 111 for small sizes, and from 21 to 6 for larger sizes. For intermediate values, interchange degrades performance.

In some cases, optimizations produce worse results than unoptimized code. This can be observed with splitting in bs for small values, edn, and marginally in fibcall (31 vs. 30), as well as interchange as discussed. This behavior is not different from optimizing for performance that is tuned for average case and occasionally results in poor performance.

So far, we only experimented with applying a single optimization at a time, in order to show the potential. It is well known that optimizations interact in complex. A promising direction consists in combining many optimizations to decrease the number of checkpoints. Iterative compilation can certainly be applied to explore a region of interest of the optimization search space, and heuristics be developed in a way similar to compiling for performance.

Number of static checkpoints. Figure 5 shows the static number of checkpoints inserted along with the number of times they are taken at runtime (dynamic

number on the left axis, static on the right). We only report a limited number of benchmarks, as they all exhibit the same behavior.

Overall, the number of static checkpoints is decreasing as the number of dynamic checkpoints, as expected. However, there are exceptions, as in bs. 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 impacts 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, minmax is a very simple program, all functions have only one context. 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. Also, minmax is a benchmark without loop and it has infeasible paths which are never taken at runtime. This is the reason that the number of static checkpoints for some capacitor sizes is larger than the number of dynamic checkpoints.

5 Conclusion

We propose a compile-time checkpoint insertion strategy for intermittently powered system. Our approach simultaneously guarantees program correctness and forward progress. It does not require any additional hardware. To achieve this, our toolchain inserts checkpoint trigger calls based on worst-case energy consumption of program sections. The called function saves the state of the program to non-volatile memory before the energy depletes. In addition, we show that classical compiler optimizations can be exploited to reduce the number of checkpoints, hence the overhead.

References

1. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**(4), 345–420 (Dec 1994)
2. Baghsorkhi, S.S., Margiolas, C.: Automating efficient variable-grained resiliency for low-power IoT systems. In: CGO. pp. 38–49. ACM (2018)
3. Balsamo, D., Weddell, A.S., Das, A., Arreola, A.R., Brunelli, D., Al-Hashimi, B.M., Merrett, G.V., Benini, L.: Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. IEEE TCAD **35**(12) (2016)
4. Balsamo, D., Weddell, A.S., Merrett, G.V., Al-Hashimi, B.M., Brunelli, D., Benini, L.: Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. IEEE Embedded Systems Letters **7**(1), 15–18 (2015)

5. Bhatti, N.A., Mottola, L.: HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In: IPSN. IEEE (2017)
6. Bouziane, R., Rohou, E., Gamatié, A.: Energy-efficient memory mappings based on partial WCET analysis and multi-retention time STT-RAM. In: RTNS (2018)
7. Bouziane, R., Rohou, E., Gamatié, A.: Partial worst-case execution time analysis. In: Conférence d'informatique en Parallélisme, Architecture et Système (2018)
8. Colin, A., Lucia, B.: Chain: tasks and channels for reliable intermittent programs. In: ACM SIGPLAN Notices. vol. 51, pp. 514–530. ACM (2016)
9. Colin, A., Lucia, B.: Termination checking and task decomposition for task-based intermittent programs. In: Intl. Conf. on Compiler Construction. ACM (2018)
10. Georgiou, K., Xavier-de Souza, S., Eder, K.: The IoT energy challenge: A software perspective. IEEE Embedded Systems Letters **10**(3) (2018)
11. Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Eder, K.: Static energy consumption analysis of LLVM IR programs. arXiv (2014)
12. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks: Past, present and future. In: Intl. Workshop on Worst-Case Execution Time Analysis (2010)
13. Hardy, D., Rouxel, B., Puaut, I.: The Heptane static worst-case execution time estimation tool. In: Intl. Workshop on Worst-Case Execution Time Analysis (2017)
14. Jayakumar, H., Raha, A., 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. IEEE (2014)
15. Johnson, R., Pearson, D., Pingali, K.: The program structure tree: Computing control regions in linear time. In: ACM SigPlan Notices. vol. 29. ACM (1994)
16. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. IEEE Computer Society (2004)
17. Li, H., Puaut, I., Rohou, E.: Tracing flow information for tighter WCET estimation: Application to vectorization. In: RTCSA. IEEE (2015)
18. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: ACM SIGPLAN Notices. vol. 30. ACM (1995)
19. Lucia, B., Ransford, B.: A simpler, safer programming and execution model for intermittent systems. ACM SIGPLAN Notices **50**(6), 575–585 (2015)
20. Maeng, K., Colin, A., Lucia, B.: Alpaca: intermittent execution without checkpoints. OOPSLA **1** (2017)
21. Ransford, B., Lucia, B.: Nonvolatile memory is a broken time machine. In: Workshop on Memory Systems Performance and Correctness. ACM (2014)
22. Ransford, B., Sorber, J., Fu, K.: Mementos: System support for long-running computation on RFID-scale devices. In: ACM SIGARCH Computer Architecture News. vol. 39. ACM (2011)
23. Van Der Woude, J., Hicks, M.: Intermittent computation without hardware support or programmer intervention. In: USENIX OSDI (2016)
24. Wägemann, P., Distler, T., Hönig, T., Janker, H., Kapitza, R., Schröder-Preikschat, W.: Worst-case energy consumption analysis for energy-constrained embedded systems. In: ECRTS. IEEE (2015)
25. Wägemann, P., Distler, T., Janker, H., Raffeck, P., Sieh, V.: A kernel for energy-neutral real-time systems with mixed criticalities. In: RTAS. IEEE (2016)
26. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., et al.: The worst-case execution-time problem—overview of methods and survey of tools. ACM TECS **7**(3) (2008)