So Far So Good: Self-Adaptive Dynamic Checkpointing for Intermittent Computation based on Self-Modifying Code

Bahram Yarahmadi, Erven Rohou

So Far So Good

Self-Adaptive Dynamic Checkpointing for Intermittent Computation based on Self-Modifying Code

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

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

Abstract

Recently, different software and hardware based checkpointing strategies have been proposed to ensure forward progress toward execution for energy harvesting IoT devices. In this work, inspired by the idea used in dynamic compilers, we propose SFSG: a dynamic strategy, which shifts checkpoint placement and specialization to the runtime and takes decisions based on the past power failures and execution paths taken before each power failure. The goal of SFSG is to provide forward progress and to avoid facing non-termination without using hardware features or programmer intervention. We evaluate SFSG on a TI MSP430 device, with different types of benchmarks as well as different uninterrupted intervals, and we evaluate it in terms of the number of checkpoints and its runtime overhead.

1 Introduction

With the advent of Internet of things (IoT), there is a need to provide energy for a massive number of tiny smart devices without using large, heavy, and high maintenance batteries. One promising way is to harvest energy from the environment and store it into a capacitor which is in charge of supplying energy to the device. However, harvested energy sources are all unstable [12] making the execution of programs intermittent. That is, the program is executed as long as there is available energy in the capacitor, and crashes when it exhausts. As a result, programs 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*. This consists of storing all necessary volatile data such as processor state, program stack, and global variables into a persistent memory before energy depletion. When the energy becomes available again, all the volatile state is copied back and the program can continue its execution.

While the idea of taking checkpoints is a good solution, taking checkpoints too late, may put the system at risk of unhandled power failure. In the worst case, the program faces non-termination when the code between two checkpoints requires more energy than the capacitor can provide. This is one of the novel bugs introduced by intermittent execution.

Motivation. In this work, inspired by the idea used in dynamic compilers, we have adopted a framework including several compiler analysis and transformation passes as well as a runtime system. Our work postpones the final decisions about the location of the checkpoints to the runtime. The decisions that it takes at runtime are based on the execution path that the program has taken before power failure happened. In this way, the runtime system learns from power failures in order to place checkpoints. Our approach is self-adaptive in a sense that with the changes in program runtime behavior, the runtime system can change and specialize checkpoint locations. Moreover, our work guarantees termination by identifying those sections of program code that consume more energy than the capacitor's energy when it is fully charged. Our work does not require any special hardware features or preempt any features originally available to the programmer (peripherals of the MCU such as a timer or the ADC of the system) which may be required by the application. The concept presented here can thus be applied in a wide range of commercial and commodity devices. We illustrate it on an MSP430 board from Texas Instruments.

However, two things that complicate applying concepts of dynamic compilers for ultra-low power IoT devices are: (1) lack of time to generate code, as the typical active time in these devices are in terms of milliseconds [5]; (2) lack of sufficient memory, as usually dynamic compilers consume significant amounts of memory whereas the typical available memory for an energy harvesting device is in terms of kilobytes [10]. To cope with these limitations, we have adopted a fast with moderate memory overhead runtime technique for placing and specializing checkpoints into the code. We achieve this by applying a runtime Self-Modifying Code (SMC) technique which leverages the information provided by static analysis. This synergy – sometimes referred to as *split-compilation* [7] – has been shown to deliver good runtime results at low cost.

The contribution of this paper is SFSG[1]: an **adaptive** checkpointing scheme **at runtime** based on the current actual execution of the program, **without using any dedicated hardware feature**, and totally **transparent** to the programmer.

Paper Structure. Section 2 reviews related work. Section 3 presents the overview of SFSG. The compile-time part of SFSG is described in Section 4, while the run-time part is covered by Section 5 (trace management) and Section 6 (checkpoint management). We evaluate SFSG in Section 7. Finally, Section 8 concludes.

[1] SFSG stands for *so far, so good*: in the learning steps, we execute the program as far as we can and crash when we run out of energy. We then remember where we crashed and introduce a checkpoint a bit earlier.

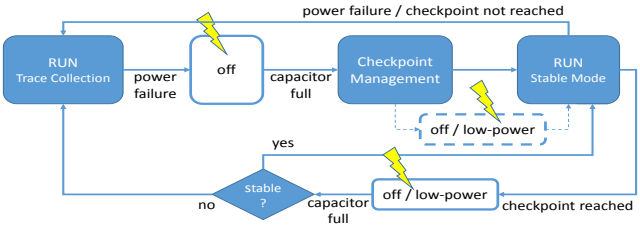

Figure 1. Flow of runtime states

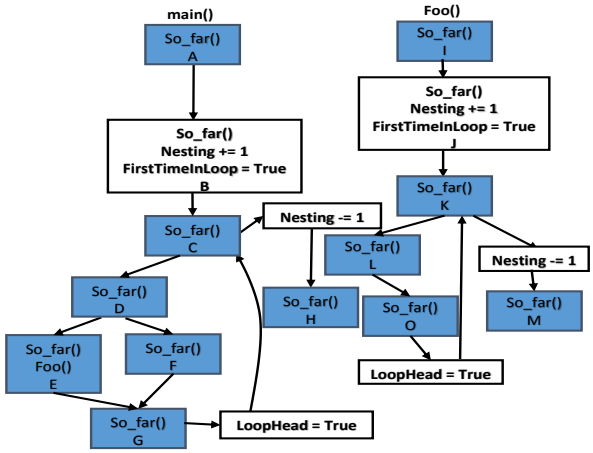

Figure 2. Static transformations of CFGs

2 Related work

Researchers have proposed different approaches from fully software solutions [2, 8, 15, 16, 20, 22] to joint hardware/-software solutions [3, 4, 9, 11, 19] as well as hardware-only solutions [21, 23] for making forward progress toward the execution of programs in energy harvesting systems. As an example of a software solution [22], a compiler can statically analyze the energy consumption of a program on a particular hardware and place checkpoints into the control-flow graph (CFG) of the program accordingly. However, to guarantee forward progress without facing non-termination, together with avoiding unnecessary checkpoints, it mandates having a safe and tight bound of the energy consumption of a program on the particular micro-controller (MCU). This is hard to achieve as the energy consumption is highly dependent on the statically unknown factors such as program's data [18] and the temperature of the deployment environment. Capacitors are also known to age [13], losing capacity over time. Systems based on static estimation of reachable checkpoints are either under-optimized in their early life, or doomed to fail when devices grow older.

In contrast to software approaches, both hardware based and joint hardware/software solutions can take checkpoints at the cost of adding additional hardware features or using hardware features of the commercial MCUs which are originally designed to be utilized by the programmer (e.g., timers, ADCs) and not by the third party system software.

3 Overview of SFSG

Intuitively, SFSG is based on the observation that an IoT program performs a series of tasks continuously for a long period of time, and control paths repeat during the execution. For instance, small IoT devices often consist of an infinite loop that senses some data from the environment, computes, possibly encrypts, and transmits the result. But repetition of control flow applies – to varying degrees – also to most applications. This is the reason behind profile-guided optimizations, but also instruction caches and branch predictors.

In a nutshell, we let the program run, only slightly modified to generate an execution trace in non-volatile memory, until energy depletes. When power resumes, and the system restarts, we know from the trace where it crashed. We insert a checkpoint slightly before, and restart execution from the beginning. Hopefully, execution reaches the checkpoint. Future re-execution will resume from this new point, thus guaranteeing forward progress. If the checkpoint is not reached, we insert it slightly earlier in the CFG and restart. At the checkpoint, we wait until the capacitor recharges completely.

The runtime system is responsible for the overall orchestration of the various steps at play. It is illustrated with a finite state machine in Figure 1. The states behave as follows.

RUN Trace Collection: In this state, while the program is executing, the trace is also being collected. These traces are stored in non-volatile memory.

off: In this state, the system is inactive or totally shutdown as a result of a power failure.

Checkpoint Management: In this state, the runtime's job is to place, modify and specialize a checkpoint call trigger based on the available collected trace in non-volatile memory.

RUN Stable Mode: In this state, the program is executing but this time without collecting trace and with the hope of reaching the checkpoint trigger call placed in state **Checkpoint Management**. However, if a power failure happens, the system goes to `RUN Trace Collection` state.

off/low-power: In this state, whether the system is in a low-power mode or totally inactive, it does not do anything and is waiting for a wake-up signal to be raised. At the time when the wake-up signal is raised, depending on the `stable` variable, the system goes to either `Run Trace Collection` or `Run Stable mode`,

4 Static Program Preparation and Transformation

The static phase transforms CFGs, extracts properties from the program and stores them alongside the binary. It also instruments the program to add control points in various places. Figure 2 shows how `main()` and `Foo()` CFGs are transformed. We leveraged the LLVM compiler [14]. Our LLVM passes perform the following steps.

1. They add preheaders to loops, if not already present.
2. They insert the special function trigger call `so_far()` at the beginning of each basic block of the program.
3. As during the execution of the program, the runtime system must be aware of the type of the basic blocks

(e.g., loop head), loops and nesting, at compile-time, we add some statement into the different locations in the program and loops:
   a. at each preheader, it increments a variable called *Nesting* and changes the value of another variable called *FirstTimeInLoop* to *true*;
   b. at each loop exit, it decrements the *Nesting* variable;
   c. at each loop-latch, it changes the value of a variable called *LoopHead* to *true*.
4. They rename the main function of the program, and substitute our own entry point to perform overall orchestration of the system at startup. They also add `so_far()`, checkpoint and restore routines into the address space. For the latter two routines, we use routines from Mementos [19].
5. They provide in non-volatile memory some information for the run-time part, for instance, the address of the beginning and the end of all basic blocks.

5 Trace Management

We redefine a trace as a sequence of basic blocks that are executed in a power cycle (before a power failure). Traces may become huge, and can easily exceed the total memory size of small IoT devices, typically in the order of kilobytes. We have devised a simple and efficient trace collection algorithm to collect the information we need within a small amount of memory.

5.1 Trace collection

The `so_far()` function which is provided by our library contains a few low-level C and assembly instructions appending the address of the first instruction of each basic block (the address of `CALL so_far()`) to a reserved area in non-volatile memory as a trace element. We demonstrate SFSG trace collection by making different power failures scenarios during the execution of the CFGs of Figure 2. The runtime system uses reserved locations in non-volatile memory for different types of information. For instance, in Figure 3 which represents the execution of `main()` function over time, it reserves two locations for trace and loop information.

The program starts its execution from basic block `A` of the CFG `main()`. At this point, the reserved areas for trace elements and loop information are empty (t_0). When `so_far()` is being executed, it appends element `A` in trace location and increments the number of trace elements (t_1). In t_2, which shows the execution after basic block `B`, the same scenario is repeated. However, this time just after the execution of `so_far()`, volatile variables' values *FirstTimeInLoop* and *Nesting* are changed. The purpose of these variables is to inform the runtime system that the next basic block is a loop head and is nested. Also, as these variables are volatile, if a power failure happens during the execution of basic block `B`, they do not have any effect on the trace. Still the runtime will know that the last execution basic block was `B` and it was outside any loop or nesting. When the basic block `C` is being executed, the runtime can identify the basic block as a loop head. It first appends element `C` to trace locations. Then, it allocates space for two pieces of information related to

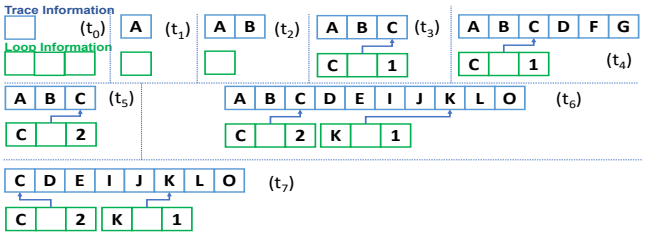

Figure 3. Trace Collection while executing basic blocks

loop information: the trace element `C`, a pointer to the trace location that contains the trace element, and the iteration of the loop (t_3). It also saves the *Nesting* variable in non-volatile memory. In this example, in the first iteration of the loop, the program takes `DFG` path and you can observe the corresponding collected trace in t_4. In the loop latch of `main()` function, the volatile *LoopHead* value will be changed to *true* and when `so_far()` in basic block `C` is being executed it starts the trace from the location of loop head `C` in loop information location and also increment the loop iteration (t_5). At runtime in the second iteration, the program takes the path `DEIJKLO` as shown in t_6.

5.2 Ephemeral tracing

Tracing is expensive, so we have designed a two-step approach to gradually limit overhead: *tracing mode* and *stable mode*.

Tracing mode is on when we discover new code, but to reach a recently placed checkpoint, SFSG temporarily turns the tracing mode off. However, after reaching the checkpoint or facing a power failure during execution, the system enters *tracing mode* again. After each iteration of the outermost loops of the program, a function (`is_stable()`) which has been inserted by the static part of the SFSG will be triggered. Typically, an IoT program contains one outermost loop that performs a series of tasks. But SFSG is not limited to this type, as it places the function for all loops with a dynamic nesting level of one. At runtime, this function checks whether at least one checkpoint has been inserted. If there is a checkpoint in the system, the system enters stable mode.

In *stable* state, we assume we have reached a steady state, and the system stops collecting traces for better performance. This is speculative, and tracing may have to be temporarily re-enabled, should a new power failure happens. Stable mode is enabled by setting a global variable (`is_stable`).

6 Checkpoint Management

We first discuss how SFSG determines the location of checkpoints, and second how insertion of checkpoint is performed.

6.1 Determining checkpoint locations

Based on our trace collection methodology discussed in the previous section, after each power failure, the runtime system has information about the last basic blocks that had been executed and the corresponding loop information (iteration and nesting level).

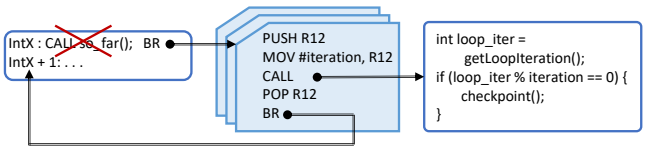

Figure 4. Checkpoint specialization with trampolines

If the selected trace element is a basic block in a loop, the runtime first (optimistically) attempts to place the checkpoint outside the loop, in the hope to execute the entire loop with a single charge of capacitor. For that purpose, the call is inserted in the loop pre-header (whose existence is guaranteed by the static compiler) as well as the loop information location. For instance, if at t_6 a power failure happens, the compiler starts with the outermost loop (`C`). The pre-header of the loop is one element before the loop head in the trace (`B`). In our trace example, after restart, the program will be re-executed from the beginning but this time at basic block `B`, a checkpoint will be taken and the system will stop executing until the capacitor is fully charged. If the program executes the same path and fails at basic block `O` again, the corresponding collected trace will be t_7. However, this time the runtime, places the checkpoint in the pre-header of basic block `K` as the pre-header of basic block `C` does not exist in the trace anymore (the beginning of the trace is the `C` element).

If (after restart) the power failure happens in the same loop again, the runtime inserts a second checkpoint in the loop body where the power failure happened. Additionally, the checkpoint is *specialized*, i.e. made conditional upon the iteration count. If a single charge of the capacitor was able to execute n iterations, we only take the checkpoint every n^{th} iterations (if $n \neq 1$). It is worth noting that each time the runtime inserts a checkpoint in the code, it saves the address of the location of the checkpoint as well as the trace to which it belongs in a reserved area in memory. If the program cannot reach the checkpoint and faces a power failure, this implies that the location of the checkpoint was not correct, or the program took a different execution path. In both cases, since the runtime saved the last checkpoint location and its trace, it can decide whether to revert the checkpoint location with a `call so_far()` instruction or not. In SFSG, if the saved trace and new trace are totally different, it reverts the checkpoint location with a `so_far()` instruction. SFSG can find a new location based on recent collected trace.

6.2 Specialized checkpoints

The process of checkpoint specialization is illustrated in Figure 4. A trampoline is created for each selected location in non-volatile memory. It is a small chunk of memory where we generate a few instructions that assign a constant to R12 (in the MSP430 ABI, R12 is used for passing the first parameter of a function) and call the actual checkpointing routine. The constant is used to specialize the checkpoint to a subset of loop iterations, as described previously. Whenever the checkpoint should not be specialized (i.e. R12=1), we bypass the trampoline and call directly into the checkpoint function. We then overwrite the call to `so_far()` by a jump to `checkpoint()`. Note that we also need to save and restore the value of R12, hence the `PUSH/POP` instruction pair.

6.3 Coping with non-termination

The energy consumption of some basic blocks might be more than the energy provided by the capacitor when it is fully charged as a result of too many instructions, or energy-consuming operations such as I/O. SFSG can identify these basic blocks when there is no collected trace after a power failure. To resolve this issue, the runtime of SFSG heuristically splits the basic block and chooses the middle of the block for placing a checkpoint. It places a branch instruction as a trampoline in the middle of the basic block by overwriting the existing instruction. The branch instruction branches into a location in NVM. The runtime then places the overwritten instruction there and places a checkpoint trigger call after that. For identifying the middle of a basic block, the runtime uses the beginning address and the end address of basic block provided by static part as well as instruction sizes. For a variable length ISA such as MSP430, we have implemented a simple and fast instruction decoder for identifying instruction sizes.

7 Evaluation

We evaluate our work on a TI MSP430FR5969 launchpad with 2 KB SRAM as volatile memory and 64 KB FRAM as non-volatile memory. The typical applications we target are meant to run forever. Total execution time makes little sense in this context. We evaluated separately the behavior of the discovery phase (early execution phase) and the stable mode (most of the execution time, after checkpoint insertions have stabilized). For the former, we studied the number of inserted checkpoints (Section 7.1), and how fast insertion stabilizes (7.2). For the latter, we measured the overhead of tracing, and the residual cost when most of the overhead is removed (Section 7.3). Finally, we also measure the code size increase (Section 7.4).

Benchmarks. We have chosen five benchmarks: A Fast Fourier Transform (FFT) and an RSA cryptography benchmark from Mementos repository [1], CRC32 (CRC error checking 32-bit), aha-mont64 (Montgomery multiplication) and ud (Simultaneous Linear Equations by LU Decomposition) from Embench [6]. They only perform computations and they do not contain any I/O. However, these benchmarks are highly used in the IoT domain. As mentioned earlier, IoT applications typically run for a long period of time. Therefore, we made each benchmark iterate a number of times to see the runtime behavior of our work. To show that SFSG makes forward progress to completion without facing non-termination in extreme cases, we have developed a benchmark which uses the timer of the system and activates LEDs periodically. We compiled each benchmark with `-O1`.

Protocol. To explore how SFSG adapts to diverse situations (different capacitors, changing environment, aging), we experimented with varying capacitor sizes. Since it is hardly practical to physically replace a capacitor a large number of

times, we developed an alternative setup. An external device is in charge of sending a reset signal to our experimental board at preconfigured time intervals. This simulates a power failure followed a restart when the capacitor is full (we do not take into account the energy harvesting time, which heavily depends on external factors, such as the technology used and ambient conditions). We considered intervals varying from 0.1 s to 2 s. We have chosen an ideal checkpointing strategy as an Oracle to compare SFSG with. For that, we utilized a timer in our experimental board raising a backup signal just before the reset signal being sent from the external device. Our Oracle is very similar to on-demand checkpointing systems [3, 4, 11] which proved to have low checkpointing overhead as they checkpoint immediately before power failure [17]. However, in reality, these systems utilize a hardware voltage monitoring system which has minor impact on the overall performance. This makes our Oracle to be superior to them but infeasible in reality.

7.1 Number of Checkpoints

The number of taken checkpoints at runtime, for different interval sizes, are shown in Figure 5. As it is observed, the number of overall checkpoints decreases when the interval size increases. We can also observe outliers. The reason for that is SFSG is a greedy approach based on a heuristic. A slight variation in interval size may result in a checkpoint placed in different basic blocks, possibly crossing a loop boundary. Local variations are thus expected. Yet, we also observe that these outliers remain rare, showing that our heuristic is fairly robust to changes in interval sizes.

Most checkpoints are able to complete before power failure happens. When power failure happens during the checkpointing operation, we say the checkpoint was unsuccessful. This case is extremely rare. It occurs once in fft, twice in ud, and five times in mont64, with the smallest capacitor sizes (under 260 ms uninterrupted execution time).

To test that our basic block splitting strategy works correctly, we changed the code of our test benchmark. We called two external functions consecutively. Each of which is responsible for activating a LED for about 160 microseconds, and we set the interval of the system to 200 microseconds. SFSG could successfully break the basic block and have forward progress to completion of the benchmark.

7.2 Stabilization of checkpoint insertion

Figure 6 shows the behavior of two representative benchmarks at runtime, shortly after startup. As shown, benchmarks take different amounts of time before they stabilize. With longer intervals, the initial performance of the execution suffers from the longer execution paths until failure followed by longer re-execution. However, this is temporary. This penalty is amortized by the fewer number of checkpoints in a long term of execution. In contrast to the longer intervals, shorter intervals go to the stable mode faster, as they suffer from failures earlier. However, they continue their execution with a higher number of checkpoints. As further discussed below (see Section 7.3), when the system is collecting a trace, it suffers from the high overhead of writing to the NVM. In stable mode, this overhead is reduced as in each `so_far()`, no writing to NVM happens. However, at each `so_far()` point some extra instructions still need to be executed (such as reading the status from NVM, comparing...) Stabilizing faster means that this overhead is eliminated earlier.

Benchmark	Trace collection	Stable mode	minimal overhead
fft	4.0×	2.0×	+9 %
rsa	8.3×	3.4×	+27 %
crc32	5.3×	2.5×	+8 %
ud	3.4×	1.8×	+9 %
aha-mont64	2.1×	1.4×	+2 %
test	1.0×	1.0×	+0 %

Table 1. Overhead compared to Pure C.

7.3 Tracing Overhead

Table 1 shows the impact of the runtime execution modes compared to Pure C[2]. As expected, trace collection mode has a severe overhead in the execution of the program, degrading performance up to 8× in the worst case. However, the trace collection step is temporary and it is the penalty that the system pays for locating the checkpoints when discovering the code. The overhead of this mode is dependent on the benchmarks and their number and size of basic blocks. For instance, rsa has a large number of basic blocks with the small code size as well as loop nesting. This causes this benchmark to have a high overhead when it is in a tracing mode. In contrast to rsa, test has a small number of basic block with two levels of loop nesting. The program is waiting most of the time for pseudo-peripheral operations that take significant time. As a result, collecting traces has a marginal overhead on the execution.

In stable mode, the system does not have the high overhead of the trace collection. However, there is still a function call to `so_far()`, and a few instructions to read a status variable from NVM, compare its value, and return to the caller. This makes the execution still slower than the Pure C, but cuts the overhead of trace collection in half, resulting in slowdowns from 1.3× to 3.4× (excluding test).

To better observe the overhead of `so_far()` function calls when the system is in stable mode, we have replaced every `so_far()` in the code by NOPs. As shown in the last column of Table 1, in benchmarks with a large number of basic blocks and less time consuming instructions, `so_far()` has non-negligible overhead. As a future research direction to reduce the overhead of `so_far()`, the static part of SFSG can decide not to place `so_far()` on some basic blocks. For instance, those that have a small number of instructions without using any peripherals. However, it must ensure that the system will make forward progress. The runtime system could also decide to dynamically overwrite the calls by NOPs when the system is deemed stable enough, with a rollback mechanism in case tracing must be enabled again.

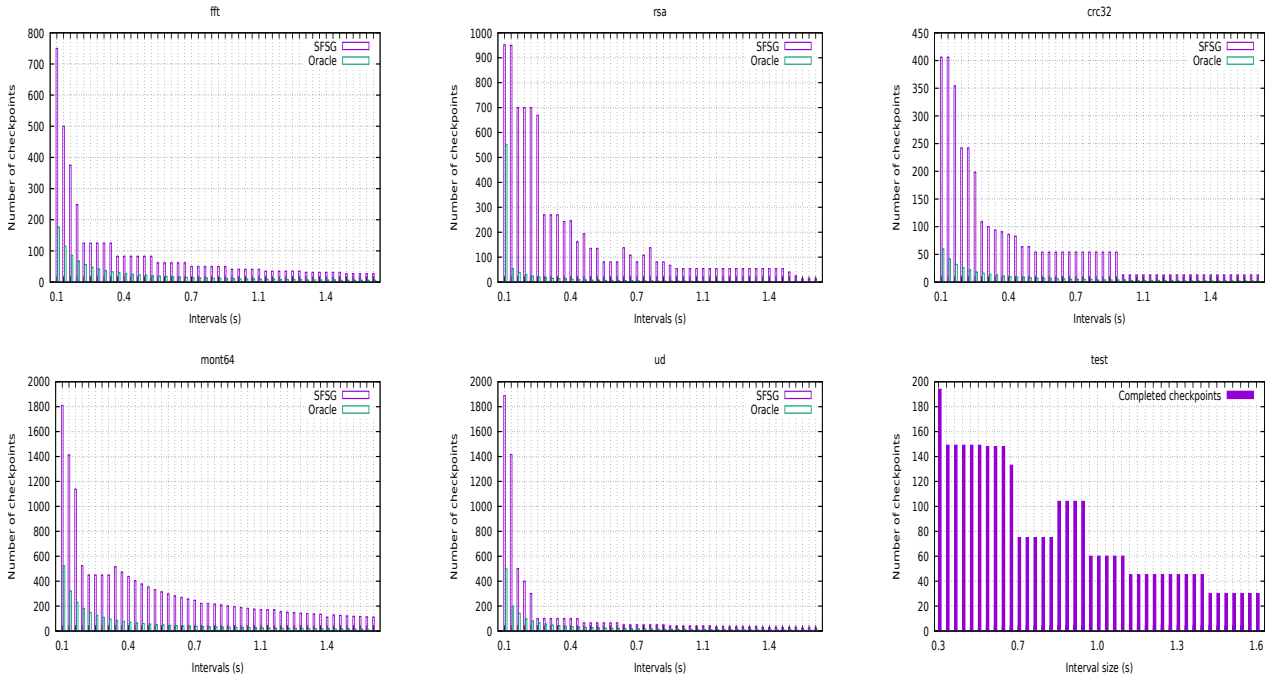

Figure 5. Dynamic number of checkpoints taken, for different uninterrupted interval sizes

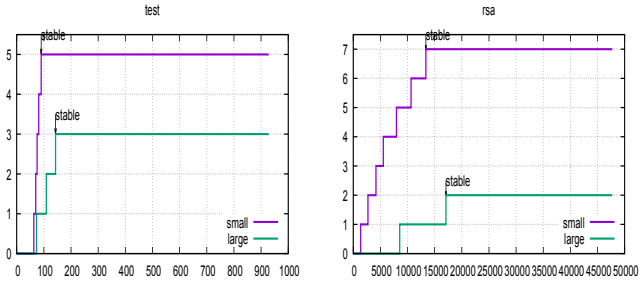

Figure 6. Behavior of benchmarks in early execution steps after startups. Two interval sizes are shown (small is 325 ms and large is 2 s). The x-axis represents the number of executed basic blocks. The y-axis represents the static number of checkpoints inserted so far (ie, the number of times SFSG entered the checkpoint management state). The moment the system was considered stable is shown.

Benchmark	Pure C	SFSG	Benchmark	Pure C	SFSG
fft	3640	8762	ud	8188	13616
rsa	9366	16752	aha-mont64	8282	13394
crc32	2800	7556	test	1076	6414

Table 2. Code Size in Bytes

7.4 Code Size

In terms of code size, we compare our work with pure code that is generated by clang. The code size of our work includes checkpoint and restore routines from Mementos, `so_far()` routine, our runtime system, traces and addresses of basic blocks. Table 2 reports the code size, defined as the number of bytes transmitted to the board. The difference between the size of the code of our work with Pure C code generated by clang is significant in relative value, but is a mere 5 KB on average. The checkpoint and restore routine contribute a large amount of the increase of the device text sections. However, these two routines contain the basic features that a software approach must have in order to be able to survive power failures, and they are similar to other solutions proposed by related work. We also measured the size of the dynamically created objects during execution (traces, trampolines) in the worst case. Thanks to our algorithm and the existence of the loops as well as the benchmarks in IoT domain, it is on average 150 bytes across all benchmarks with different intervals.

8 Conclusion

We introduced SFSG: an adaptive checkpointing strategy for intermittently powered systems. Our approach makes decisions about locations of the checkpoint routines and specialize them at runtime based on the actual execution of the program in the deployment environment. SFSG automatically discovers the parts of the code that consume more energy

[2]Note that the performance of *Pure C* is an optimistic upper bound: no checkpoint is inserted, and benchmarks would not survive a power depletion.

than the capacitor of the MCU can provide and inserts checkpoints to guarantee termination. In addition, SFSG requires no extra programming effort as well as no extra hardware support.

References

[1] [n. d.]. . https://github.com/CMUAbstract/mementos.

[2] Sara S Baghsorkhi and Christos Margiolas. 2018. Automating efficient variable-grained resiliency for low-power IoT systems. In *2018 International Symposium on Code Generation and Optimization*. 38–49.

[3] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 1968–1980.

[4] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2014. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2014), 15–18.

[5] Michael Buettner, Richa Prasad, Alanson Sample, Daniel Yeager, Ben Greenstein, Joshua R Smith, and David Wetherall. 2008. RFID sensor networks with the Intel WISP. In *6th ACM conference on Embedded network sensor systems*.

[6] Andrew Burgess, Ashley Whetter, George Field, Graham Markall, Hendrik Oosenbrug, James Pallister, Jeremy Bennett, Neville Grech Pierre Langlois, and Simon Cook. [n. d.]. *Embench™: A Modern Embedded Benchmark Suite*.

[7] Albert Cohen and Erven Rohou. 2010. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *DAC*. IEEE.

[8] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *ACM SIGPLAN OOPSLA*.

[9] Matthew Hicks. 2017. Clank: Architectural support for intermittent computation. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 228–240.

[10] Texas Instruments. [n. d.]. *MSP430FR59694*. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[11] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. IEEE.

[12] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B Srivastava. 2007. Power management in energy harvesting sensor networks. *ACM TECS* 6, 4 (2007).

[13] Paul Kreczanik, Pascal Venet, Alaa Hijazi, and Guy Clerc. 2013. Study of supercapacitor aging and lifetime estimation according to voltage, temperature, and RMS current. *IEEE Transactions on Industrial Electronics* 61, 9 (2013), 4895–4902.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. IEEE.

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

[16] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2019. Alpaca: intermittent execution without checkpoints. *Preprint arXiv:1909.06951* (2019).

[17] Kiwan Maeng and Brandon Lucia. 2019. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[18] Jeremy Morse, Steve Kerrison, and Kerstin Eder. 2018. On the limitations of analyzing worst-case dynamic energy of processing. *ACM Transactions on Embedded Computing Systems (TECS)* 17, 3 (2018), 1–22.

[19] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: system support for long-running computation on RFID-scale devices. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 159–170.

[20] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *12th USENIX OSDI*.

[21] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. 2012. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *ESSCIRC*. IEEE, 149–152.

[22] Bahram Yarahmadi and Erven Rohou. 2020. Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (LNCS)*. Samos (virtual), Greece.

[23] Wing-kei Yu, Shantanu Rajwade, Sung-En Wang, Bob Lian, G Edward Suh, and Edwin Kan. 2011. A non-volatile microcontroller with integrated floating-gate transistors. In *IEEE/IFIP DSN-W*. IEEE.