HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing

Naveed Anwar Bhatti
Politecnico di Milano, Italy
naveedanwar.bhatti@polimi.it

Luca Mottola
Politecnico di Milano, Italy and SICS Swedish ICT
luca.mottola@polimi.it

ABSTRACT

We present code instrumentation strategies to allow transiently-powered embedded sensing devices efficiently checkpoint the system's state before energy is exhausted. Our solution, called HarvOS, operates at compile-time with limited developer intervention based on the control-flow graph of a program, while adapting to varying levels of remaining energy and possible program executions at run-time. In addition, the underlying design rationale allows the system to spare the energy-intensive probing of the energy buffer whenever possible. Compared to existing approaches, our evaluation indicates that HarvOS allows transiently-powered devices to complete a given workload with 68% fewer checkpoints, on average. Moreover, our performance in the number of required checkpoints rests only 19% far from that of an "oracle" that represents an *ideal* solution, yet unfeasible in practice, that knows *exactly* the last point in time when to checkpoint.

CCS CONCEPTS

•Computer systems organization →Embedded systems; *Sensor networks;*

KEYWORDS

Embedded Systems, Sensor Networks, Checkpointing, Transiently-powered computing

ACM Reference format:
Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of The 16th ACM/IEEE International Conference on Information Processing in Sensor Networks, Pittsburgh, PA USA, April 2017 (IPSN 2017),* 12 pages.
DOI: http://dx.doi.org/10.1145/3055031.3055082

1 INTRODUCTION

Advances in energy harvesting and wireless energy transfer are redefining the scope and extent of the energy constraints in embedded sensing [8]. It now becomes conceivable to power not just RFID-scale devices out of harvested or wirelessly-transferred energy, but also more powerful devices operating in sophisticated applications such as smart buildings, factory automation, and mobile health [8]. However, energy provisioning from ambient energy harvesting or wireless energy transfer is generally erratic. Thus, devices need to cope with highly variable, yet unpredictable energy supplies across both space and time, and be prepared to survive periods of energy unavailability.

The problem is exacerbated as the complexity of applications grows. Many modern applications are effectively stateful [8]. In these settings, for example, whenever actuation becomes part of the application logic, actuators must retain their operating settings after a power failure to safely resume their functionality. Even in stateless implementations, the application processing deployed on embedded devices might not execute entirely on a single charge of the limited energy buffers typically employed. For example, accelerometer sensors may need to apply complex signal processing algorithms before reporting the data, which typically requires a few seconds of intense MCU utilization.

One way to enable the operation of such transiently-powered devices is to efficiently checkpoint the system's state on non-volatile memory [9, 20] whenever energy is about to be exhausted. This allows a device to resume operation from the saved state as soon as energy is newly available. *When* and *how* to perform the checkpoint, which is an energy-expensive operation per se, is crucial. Doing so too early would essentially correspond to a waste of energy that could be usefully employed in further computations. In contrast, excessively postponing a checkpoint may yield a situation where insufficient energy is left to complete the operation. Because of the unpredictable supply of energy from the environment and the varying run-time execution of programs, striking an efficient trade-off is challenging.

We present code instrumentation strategies to place calls to *trigger* functions that, based on the current system state, decide whether to perform the checkpoint before continuing the execution [20]. Different from existing approaches, we look at the control-flow graph (CFG) of a program and place triggers according to different strategies depending on the programming constructs, for example, branching statements as opposed to loops. Simultaneously, we aim at reducing the size of the checkpoint itself by placing triggers where the size of the allocated memory is reduced. The decision on whether to checkpoint is based on available energy as well as the worst-case estimation of the energy required to reach the next trigger call.

Such a scheme, which we call HarvOS, completely operates at compile-time and dynamically adapts to varying levels of remaining energy at run-time, while capturing the actual program execution through the CFG. The underlying design rationale also allows the system to spare energy-intensive probing of the energy buffer, for example, through ADCs, whenever possible.

HarvOS is largely independent of programming language, OS, and underlying platform. It is generally applicable to imperative programming languages. The execution of checkpoints is transparent to the OS as long as a way to make these operations atomic is somehow provided. Our solution applies both to platforms where traditional volatile memory is employed for normal processing and a separate non-volatile memory is reserved for checkpoints, and to platforms where non-volatile memory is used in place of volatile one; for example, when FRAM chips replace normal SRAM chips.

Our evaluation considers modern 32-bit MCUs and three increasingly complex benchmark codes commonly employed in embedded sensing. We sweep the possible executions of programs against varying size of the underlying energy buffers to measure the performance of HarvOS against existing approaches. The results we collect indicate that, for example, HarvOS allows a device to complete a given workload with 68% fewer checkpoints, on average compared to existing approaches. Moreover, such a performance rests 19% far from that of an "oracle" that represents an *ideal* solution, yet unfeasible in practice, that knows *exactly* the last point in time when a checkpoint is required.

The benefits are not, however, limited to the number of required checkpoints. Our evaluation also shows that, because checkpoints in HarvOS happen much closer to the last practical point in time when the system should take a checkpoint, we can also reduce the processing that would go wasted as its results would not become part of any checkpoint. We further demonstrate that, unlike existing approaches, our performance is largely robust against different program structures. Ultimately, this means that energy utilization is improved in a larger set of applications, as it is employed more for useful computations than for checkpointing.

The rest of the paper unfolds as follows. Section 2 places our work in context. Section 3 describes the design rationale and the foundations of our approach. Section 4 describes the compile-time rules we apply to decide on the placement of trigger calls depending on the program structure. Experimental results are reported in Section 5. We end the paper with brief concluding remarks in Section 6.

2 RELATED WORK

Relatively little research exists on enabling transiently-powered computing on embedded devices. Recent work comprehensively describes existing approaches and quantitatively compares them against each other [6]. Here we focus on the aspects most relevant for code instrumentation. We recognize two classes of such approaches.

One class is based on separate memory areas for normal computations and for checkpointing. Examples are MementOS [20] and Hibernus [7]. The MementOS prototype uses flash memory for checkpointing. Trigger calls are executed periodically or placed using a *loop-latch* or *function-return* strategy. The former places trigger calls at the end of loop iterations; the latter places trigger calls at function return points. These are the locations where one may expect the stack to store less data, which would then reduce the size and energy cost of moving data to flash. The decision to checkpoint is based on a voltage threshold obtained through repeated emulation experiments that eventually determine a single program-wide threshold based on average run-time behavior and user-supplied energy traces.

Hibernus [7] uses FRAM instead of flash memory, and triggers a checkpoint based on a hardware interrupt firing if the operating voltage drops below a threshold. Because of the higher energy efficiency of FRAM compared to flash memory, the latter can be statically defined because Hibernus can afford to copy the entire RAM segment independent of the current memory occupation. The energy to perform such a fixed-cost checkpoint is stored in a separate decoupling capacitor, in turn driven by an external voltage regulator. In contrast, one of our goals is to enable efficient checkpointing without requiring hardware modification. Furthermore, FRAM chips are still limited in overall size. It is then difficult to store multiple checkpoints; for example, to ensure that at least a complete consistent checkpoint is always available.

The other class of solutions employ non-volatile memory, especially FRAM, as the only memory space. This means FRAM is used both for normal computations and for saving the system's state in periods of energy unavailability. The advantage is that application data already resides on non-volatile memory, so only registers and program counter need to be saved when checkpointing. QuickRecall [16] is an example in this class. These solutions are especially indicated for scenarios characterized by very short energy bursts, as checkpoints can happen quickly. However, they suffer from an increase of energy consumption during normal computations due to the use of FRAM in place of SRAM, and from potential data consistency issues that require specialized compiler techniques [18].

HarvOS is independent of the underlying memory architecture, and applies to both classes of approaches with only minor changes. The trigger placement rules we describe next may replace or complement the heuristics or periodic trigger calls employed in the aforementioned systems. Our design is rooted in the unbalance between normal computation and the energy-hungry operation of checkpointing, and seeks to reduce the overhead of the latter.

3 OVERVIEW

Calls to trigger functions placed anywhere in the code essentially represent an overhead compared to the normal computation. In existing systems, two operations are performed every time the execution encounters a trigger call. First, the system verifies some condition that indicates whether it is time to checkpoint. MementOS, for example, uses a voltage threshold as explained in Section 2. If the condition is verified, the checkpoint takes place. The energy cost of checking whether a checkpoint is necessary is normally constant.

The energy cost of the actual checkpoint, on the other hand, depends on the underlying memory architecture. For platforms that only employ a single non-volatile memory area [16], the size of checkpoints is fixed and independent of where the checkpoint takes place throughout the program execution: only registers and program counter need to be saved. Thus, the energy cost of checkpointing is fixed.

In platforms employing separate memory areas for normal computations and for checkpointing [7, 20], the entire allocated memory needs to be saved, including stack and heap, at the time of checkpointing. As a result, the size and therefore the energy cost of checkpointing depend on where in the program the checkpoint takes place, making this energy cost typically proportional to the size of the allocated memory [9]. For example, the higher the stack at that point in the execution, the larger is the energy cost of checkpointing.

Challenge. Our objective is to minimize the energy overhead due to checkpointing operations. This means *i)* to minimize the number of trigger calls that are uselessly executed, that is, to verify no checkpoint is needed, and *ii)* to postpone the actual checkpoint to a moment where the available energy is strictly sufficient to that end, that is, one can not perform further computations without jeopardizing the ability to checkpoint later.

The two needs are at odds with each other. Postponing the checkpoint, in fact, requires to frequently check how close is the execution to when no sufficient energy is left to perform the checkpoint. However, trigger calls need to probe the energy buffer, for example, through ADC operations. Therefore, frequently performing this operation may become prohibitive because of the energy cost. The negative effects are not limited to energy consumption. Trigger calls might, in addition, change the execution timings. Using resource-constrained devices, this may introduce subtle software bugs [22].

Rationale. To optimize the point in time when the actual checkpoint takes places and its energy cost, we rely on compile-time information on memory allocation patterns. Static code analysis techniques exist that can return accurate information on the evolution of the stack and, in many cases, of the heap as well [4, 14, 15]. The latter techniques especially apply when the size of heap-allocated data structures is known at compile-time; for example, whenever objects are dynamically allocated in languages such as C++.

Similar to existing works [7, 16, 20], we focus on supporting transiently-powered computing for the main MCU. Other components on the device, such as sensors or radios, may operate through separate energy buffers [13] or techniques such as radio backscattering [17]. The former technique effectively decouples the energy management of peripherals from that of the MCU, which is in charge of driving the entire system and thus requires ad-hoc techniques to operate across periods of energy unavailability. The latter techniques enable networking among embedded devices and between embedded devices and surroundings infrastructure through energy-neutral operations. These ensure that the amount of energy consumed for transmissions does not exceed the harvested RF energy.

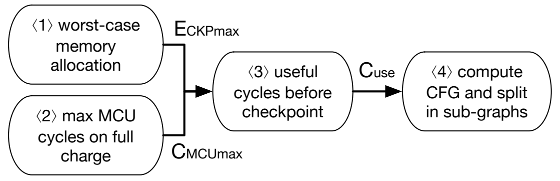

⟨1⟩ worst-case memory allocation
⟨2⟩ max MCU cycles on full charge
ECKPmax
CMCUmax
⟨3⟩ useful cycles before checkpoint
Cuse
⟨4⟩ compute CFG and split in sub-graphs

Figure 1: Compile-time operation of HarvOS.

Operation. Figure 1 illustrates the compile-time operation of HarvOS. Given a program, at step ⟨1⟩ we estimate the worst-case memory usage throughout the code and use this to obtain an estimate of the highest energy cost E_{CKPmax} for checkpointing at any point in a program's execution. The latter step is simple; for example, as the energy consumption of flash chips obeys to specific trends dictated by the manufacturing characteristics.

In case the underlying platform employs non-volatile memory also for normal computations, the energy cost E_{CKPmax} is constant; application data already resides on stable storage, and only the content of registers and program counter needs to be saved. In the following, we consider the more complex case of variable energy cost for checkpointing operations, germane to platforms that employ separate memory areas for normal computations and for checkpointing.

At step ⟨2⟩, we calculate the maximum number of cycles C_{MCUmax} we can execute whenever the device wakes up with a freshly charged energy buffer supplying energy $E_{wake-up}$. Many transiently-powered devices include a wake-up circuit that boots the device when the voltage level of the on-board capacitor surpasses a certain threshold. Knowing this value, computing C_{MCUmax} is also simple, according to an MCU's datasheet. In doing so, we consider the maximum power consumption of the MCU. This likely underestimates the value of C_{MCUmax}, and yet allows us to reason in a worst-case setting that shields us from unexpected power failures.

Steps ⟨1⟩ and ⟨2⟩ above are independent of each other. Their outputs are fed as input to step ⟨3⟩ where we compute the number of *useful* cycles C_{use} the MCU can execute in a worst-case scenario where: *i)* the device starts afresh with energy $E_{wake-up}$, *ii)* it does not receive any additional energy contribution from the environment afterwards, and *iii)* it needs to spend E_{CKPmax} right before dying to checkpoint the system's state. The number C_{use} of cycles are therefore those the MCU can execute with an amount of energy $E_{wake-up} - E_{CKPmax}$, and can be computed similarly to step ⟨2⟩. These cycles are, in practice, those the MCU can execute to make progress in the program.

Next, in step ⟨4⟩ we compute the CFG of the program and associate every block in the graph to the number of cycles required to execute it with the target MCU. A combination

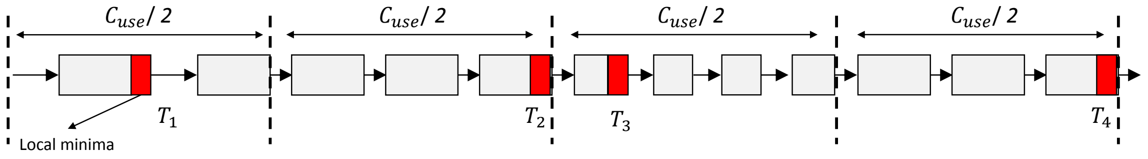


Figure 2: Splitting the CFG in sub-graphs whose required number of MCU cycles is at most $C_{use}/2$. *The picture considers a linear CFG for simplicity.*

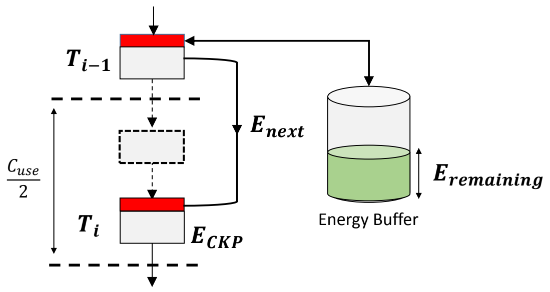


Figure 3: Decision logic to take a checkpoint. *At the T_{i-1}-th trigger call, the system checks if sufficient energy remains to reach the next trigger call at T_i and to checkpoint at T_i. If so, the execution continues. If not, a checkpoint takes place at T_{i-1}.*

of mature code inspection and emulation tools, such as Understand (www.scitools.com), Emul8 (emul8.org), and Kiel uVision (www2.keil.com) can be used to this end. We then split the CFG in sub-graphs whose total stretch in number of cycles is at most $C_{use}/2$, as intuitively shown in Figure 2. The picture considers a linear CFG for simplicity; we discuss the general case next.

Within each sub-graph, we identify the block corresponding to the minimum size of allocated memory, and place a trigger call right at the end of it. This means we aim at possibly checkpointing the system's state whenever the cost of the checkpoint operation is reduced, as the amount of data to copy over stable storage is minimal within a sub-graph. In doing so, we do not add any instrumentation other than the trigger calls themselves. As a result of these procedures, provided the code can execute entirely on a single charge, that is, the C_{use} cycles are sufficient to cover the entire execution, no trigger calls are placed anywhere in the code, which remains unaltered.

Step ⟨4⟩ explained above is crucial in the general case. Because of our placement strategy, the maximum number of MCU cycles separating any two trigger calls, for example, T_3 and T_4 in Figure 2, is bound to be less than C_{use}. The extreme case is where the T_{i-1}-th call is at the start of a sub-graph and T_i-th call is at the end of following sub-graph; in this case as well, the cycle distance is at most C_{use}. Thus, even in a worst-case situation, a device that starts afresh with energy $E_{wake-up}$ from the location of a previous checkpoint can reach the next trigger call with sufficient energy to complete the checkpoint before dying again.

As a result of the placement logic and based on the information collected up to step ⟨4⟩, at every trigger call the system can take an informed decision on whether to checkpoint. Say E_{next} is the energy to execute the required MCU cycles from the T_{i-1}-th call to the T_i-th call, whereas $E_{CKP(i)}$ is the energy required to checkpoint the system's state against the size of the allocated memory at the T_i-th call, as intuitively depicted in Figure 3. A checkpoint at the T_{i-1}-th call is required if

$$E_{remaining} \leq E_{next} + E_{CKP(i)} \tag{1}$$

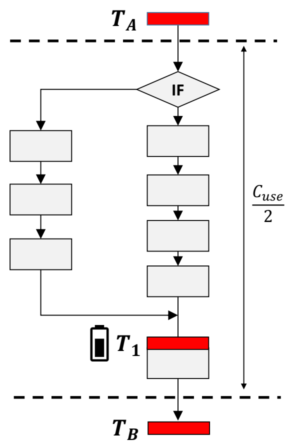
where $E_{remaining}$ is the energy left in the buffer when executing the T_{i-1}-th trigger call.

The condition in equation (1) essentially checks if the remaining energy is sufficient to reach the next trigger call *and* to checkpoint there. This reasoning assumes that the environment provisions no additional energy between T_{i-1} and T_i, that is, we are operating in a worst-case situation in terms of energy provisioning. At run-time, we can obtain the value of $E_{remaining}$ through software-based techniques [10, 21] or hardware solutions [12, 19] with negligible overhead.

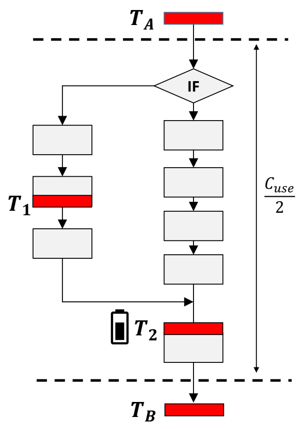
The ability to reason on whether the system can reach the *next* trigger call is one of the key of traits of our approach, and a major source of improvements compared to previous work, as we discuss in Section 5.

Generalization. CFGs are generally not linear as the example of Figure 2. On the contrary, they show complex structures reflecting the variety of available programming constructs, such as branching statements, loops, and function calls. This means there may be multiple places in a sub-graph corresponding to the minimum allocated memory, as a function of different execution paths. Moreover, embedded devices often operate in an interrupt-driven manner, that is, the execution through a CFG may be arbitrarily preempted and temporarily re-directed through the CFG of interrupt handlers. The latter case does not appear to be taken into explicit account in existing systems [7, 20].

To address these issues, the next section describes a set of trigger placement rules that, depending on the program structure, dictate where to place the trigger call and what to consider as the E_{next} energy to reach the next trigger call. We identify *branches*, *loops*, *function calls* and *interrupt handlers* as the fundamental structures of the CFG, and design specific rules for them. The complete set of rules is *recursively* applied until an elementary block in the CFG is reached. The rules also determine the conditions when probing the energy buffer for the value of $E_{remaining}$ is strictly needed, or $E_{remaining}$



(a) The location of minimum allocated memory is found outside of the branching statement.



(b) The location of minimum allocated memory is found in either of the two branches.

Figure 4: Placement rules for branching statements fully included in a single sub-graph. *The battery icon indicates where probing the energy buffer is mandatory.*

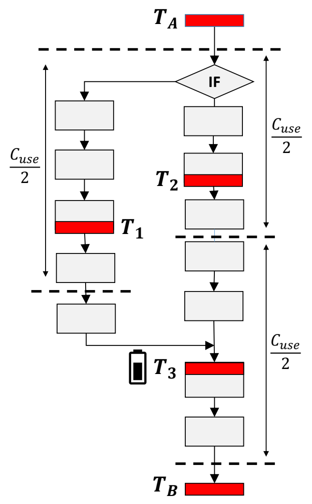


Figure 5: Placement rules for branching statements executing across multiple sub-graphs. *The battery icon indicates where probing the energy buffer is mandatory.*

can be inferred from compile-time information. The latter situation allows the system to spare operations that may be energy-expensive per se, such as probing ADCs.

4 PLACEMENT RULES

We illustrate the set of rules used to place trigger calls for arbitrary program structures. In conceiving these rules, our reasoning is based on a worst-case analysis among the possible program executions. In addition, interrupt handlers require special care, as it is generally impossible to predict the point in time when they preempt the execution.

4.1 Branching

The challenge here is to account for the lack of compile-time information on what path is taken at run-time. To address this, one may decide to instrument the code to trace the actual execution. Doing so, however, would add further overhead and greatly complicate the instrumentation strategy; as the complexity of the code grows, the number of possible paths increases exponentially. We rather adopt a worst-case approach and avoid any further code instrumentation besides the trigger calls. We demonstrate in Section 5 that this is not necessarily detrimental to performance.

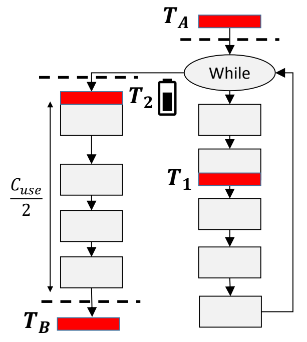
In general, what rule to apply depends on whether branching is fully included in a single sub-graph or not.

Branching in a single sub-graph. Figure 4 shows the situation. We call T_A (T_B) the last (first) trigger call in the preceding (following) sub-graph. We consider two cases:

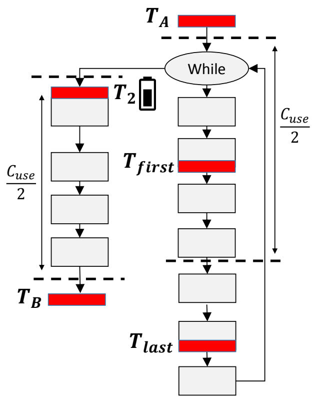
(1) The minimum amount of allocated memory is outside the branching statement, for example, at location T_1 in Figure 4a. If so, at trigger call T_A we consider the E_{next} value corresponding to the most energy-consuming branch. This means that, if the environment provisions no additional energy since T_A and the least energy-consuming branch is taken, at T_1 we are going to find a higher value for $E_{remaining}$ than we expect. This is the reason why at T_1 we are forced to probe the energy buffer to find out the exact value for $E_{remaining}$ before taking a decision to checkpoint.

(2) The minimum amount of allocated memory is found in either of the two branches, say at location T_1 in Figure 4b. According to Section 3, we place a trigger call at T_1. However, an issue arises if the *other* path is taken, where no trigger calls are placed. To cater for this, we place a trigger call right outside the branching statement, say at location T_2 in Figure 4b. The E_{next} value at trigger call T_A is set to the most energy-consuming path between those leading to either T_1 or T_2. At T_1 we can spare probing the energy buffer because, provided only one execution path exists, the energy necessary from T_A to T_1 is fixed. Differently, we still need to probe the energy buffer at T_2. In fact, without additional instrumentation, we cannot determine what path is taken at run-time.

Branching across multiple sub-graphs. Figure 5 illustrates a general example. Following the sub-graph where the last trigger call is T_A, different sub-graphs may correspond to different paths. We may thus identify a block in the CFG with the minimum amount of allocated memory in every involved sub-graph. The E_{next} value at trigger call T_A is then set to the most energy-consuming path between the one leading to either T_1 or T_2. If a single execution path exists, the trigger call along this path does not require probing again the energy buffer, as the energy required from T_A up to the



(a) The loop executes in a single sub-graph.



(b) The loop executes across multiple sub-graphs.

Figure 6: Placement rules for loops. *The battery icon indicates where probing the energy buffer is mandatory.*

trigger call is fixed. The same does not hold for the path requiring the least energy.

Based on a similar reasoning, the E_{next} value at trigger call T_1 or T_2 is set to the energy required to reach T_3. At the latter, however, we necessarily need to probe the energy buffer; again, we cannot determine what path the execution is coming from. Cases may exist where combining these rules recursively might introduce a slight overhead. For example, if multiple branch statements are nested inside each other, all of them are forced to probe the energy buffer at the end. A single probe operation may be, however, sufficient.

4.2 Loops

When the number of iterations is known or can be statically determined, placing trigger calls when a sub-graph includes loop statements is not an issue. One may first exercise existing loop unrolling schemes [11]; then apply the remainder of the techniques in this section to the resulting CFG.

Whenever the number of iterations is determined by run-time information, however, we are faced with two challenges. First, we need to decide whether the last (or single) trigger call inside the loop should consider as the E_{next} value the energy to reach the first trigger call *inside* the same loop, that is, the loop continues with the next iteration, or rather the energy to reach the first trigger call *outside* the loop, that is, the execution exits the loop. Second, whenever the latter happens, it is impossible to know how many iterations were executed without additional instrumentation, for example, in the form of counters, which would increase the overhead.

The lack of run-time information at compile-time prevents us from taking an accurate decision here. However, loops that depend on run-time information are likely to yield some number of iterations. Again, we need to distinguish whether the loop is fully included in a single sub-graph or not.

Loop in a single sub-graph. Figure 6a illustrates the situation. Consider T_A (T_B) is the last (first) trigger call in the preceding (following) sub-graph. Inside the loop, T_1 indicates the trigger call corresponding to the location of minimum allocated memory.

The E_{next} value at T_A is set to the energy to reach T_1, which is fixed if a single execution path exists; therefore, the trigger call at T_1 may not need to probe the energy buffer. This already implicitly considers the case that the execution enters the loop at least once, which may be considered as the most frequent case.

Because of the above observation, at T_1 we consider the E_{next} value to reach the next trigger call as the one corresponding to the execution continuing with the next iteration. In Figure 6a, this corresponds to the energy to execute from T_1 back to T_1. In this case, if the code inside the loop shows a single execution path, we can spare probing the energy buffer, as the energy to reach T_1 from T_1 itself is fixed.

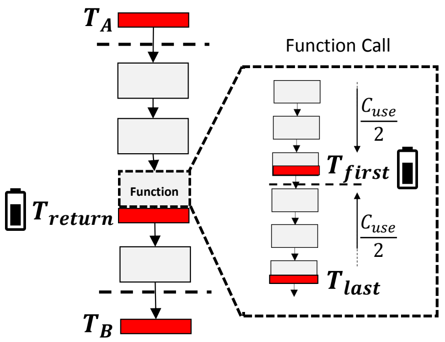
Note that the other option here would be to consider the E_{next} value corresponding to the execution exiting the loop, that is, the energy necessary to reach the first trigger call outside the loop. If this was higher than the one to reach T_1 again, the system would likely over-checkpoint without any need. Every time the execution reaches T_1 with little energy, the system would detect there is no sufficient energy to reach the first trigger call outside of the loop and it would checkpoint. However, the energy may be sufficient for another iteration of the loop if the individual iterations are less energy-expensive than reaching the first trigger call outside of the loop. We expect this to be the most frequent case.

Finally, we place another trigger call, indicated with T_2 in Figure 6a, right outside the loop. This is necessary because we have no information on how many loop iterations executed before exiting. The last time the trigger call at T_1 is executed before the loop breaks may not checkpoint, as the system thinks one more iteration is possible. As this reasoning is no longer applicable when the execution exits the loop, at T_2 we are forced to probe the energy buffer to possibly checkpoint. The E_{next} value at T_2 considers the energy to reach T_B, that is, the first trigger call in the following sub-graph.

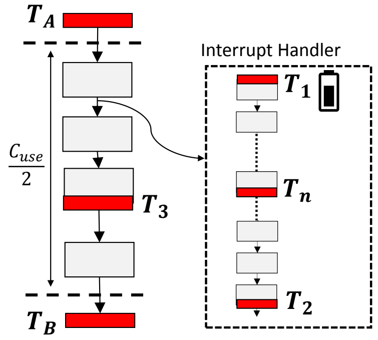
Loops across multiple sub-graphs. Figure 6b illustrates a general example. We handle this case as an extension of the previous one, with the added complexity that multiple trigger calls are necessarily included in the loop body because it spans multiple sub-graphs.

In this case, the last trigger call in the previous sub-graph considers the E_{next} value to reach the first trigger call inside the loop, indicated with T_{first} in Figure 6b. Inside the loop body, the last trigger call, that is, T_{last} in Figure 6b, considers the energy value E_{next} corresponding to continuing with a further loop iteration, which leads to T_{first}. Again in the case of a single execution path, the E_{next} value at T_{last} to reach T_{first} is fixed, so we can spare probing the energy buffer. This again considers the case of the loop continuing with the following iteration as the most probable.

Right outside the loop, we still need to place a further trigger call T_2, required for the exact same reasons as the case of Figure 6a.



(a) Function calls.



(b) Interrupt handlers.

Figure 7: Placement rules for function calls and interrupt handlers. *The battery icon indicates where probing the energy buffer is mandatory.*

4.3 Function Calls and Interrupt Handlers

Placement rules for function calls and interrupt handlers follow a similar rationale as the previous cases.

Function calls. Whenever a given execution path includes a function call, the situation is equivalent to "inlining" the CFG of the function within the CFG of the caller, as shown in Figure 7a.

If the execution of a function call spans multiple sub-graphs, at least one trigger call is placed at a local minimum of allocated memory inside the function. The energy value E_{next} at the last trigger call in the previous sub-graph considers the energy to reach the first trigger call inside the function, that is, T_{first} in Figure 7a.

The issue here is that same function may be called at multiple places in the code. Without resorting to additional code instrumentation, it is impossible to differentiate these cases. Therefore, at T_{first} we necessarily must probe the energy buffer, as the function's execution is unaware of where the caller code issued the call; thus, we cannot know uniquely what is the amount of energy spent since an arbitrary T_A.

Because of the same reason, we need to place a further trigger call right after the function returns, indicated with T_{return} in Figure 7a. Chances are that the local minimum in allocated memory within the sub-graph is found right when the function returns, that is, the time when the function's activation record—including the memory allocated for local variables and to resume execution of the main code—is de-allocated. The E_{next} value at the last trigger call inside a function considers the energy spent to reach T_{return}.

Interrupt handlers. The case of interrupt handlers adds another challenge to those for function calls. Without additional instrumentation, interrupt handlers have no information on what was the situation in the execution the handler preempted. Consider Figure 7b as an example. The trigger call at T_A decides not to checkpoint as the condition in equation (1) indicates the execution may reach T_3. An interrupt handler preempts the execution between T_A and T_3. Coping with this case requires two rules in addition to those normally applied to the code in the interrupt handler:

(1) When it starts, the interrupt handler has no information on what decision was taken at T_A. Therefore, we place a trigger call right at the beginning of the interrupt handler to verify that, based on remaining energy, the next trigger call within the interrupt handler is reachable. Probing the energy buffer is thus mandatory at T_1.
(2) When the interrupt handler finishes, it has no information on where the next trigger call is located in the main code. It may happen that the execution at T_A was actually expecting to checkpoint at T_3, as we predicted we could reach T_3 with just the right amount of energy. As the interrupt handler consumes some energy per se, T_3 may be unreachable now, and we need to checkpoint before returning to the main code. To capture these situations, every trigger call raises a flag if it expects to checkpoint at the immediately following trigger call. In this example, T_A would raise the flag. The flag prompts an additional trigger call at the end of the interrupt handler, shown as T_2 in Figure 7b to checkpoint.

Finally, we need a second flag to indicate that the main code was preempted. The trigger call at T_3 may be one that does not require probing the energy buffer, according to normal placement rules. This decision must be superseded if an interrupt handler executed in between, whose energy cost is generally not known.

5 EVALUATION

We assess the effectiveness of HarvOS along multiple dimensions. In the following, Section 5.1 describes the experimental settings, whereas Section 5.2 reports on the results. Our key findings are summarized as follows:

- HarvOS allows transiently-powered devices to complete a fixed workload with with 68% fewer checkpoints, on average compared to existing approaches;
- HarvOS performance rests 19% far from that of an "oracle" that, while unpractical in reality, knows exactly the last point in time when to checkpoint;

- compared to existing approaches, HarvOS reduces the amount of MCU processing whose results do not eventually become part of a checkpoint;
- compared to existing approaches, HarvOS allows transiently-powered devices to complete the same fixed workload using smaller energy buffers;
- HarvOS performance is robust against implementations of different complexity and structure, unlike existing approaches.

The following sections provide quantitative evidence of these findings.

5.1 Settings

Benchmarks and setup. We consider publicly available C implementations of Kalman filter [3], finite impulse response (FIR) filter [2], and Advanced Encryption Standard (AES) [1] with key length of 256 bits. Kalman filters are often used in embedded sensing to process accelerometer values; for example, to predict future trends. FIR filters are equally used in embedded sensing to filter out noise, especially when the input signal includes multi-rate components. AES is a block-cipher algorithm widely employed in embedded systems.

The implementations we consider include a variety of branching statements, loops, and function calls; therefore, they exercise most of the trigger placement rules described in Section 4. These implementations also exhibit different degrees of complexity, with the Kalman filter code being the simplest, the FIR filter code being the longest in number of lines of C code, and the AES implementation being the one structurally most complex. Overall, the benchmarks are arguably on par, and sometimes more complex, than those considered in existing literature [6, 20]

The Kalman filter code executes a fixed workload of 1000 iterations using 48 bytes of dummy data as input, emulating the use of Kalman filter to process consecutive acceleration readings. Instead, we consider a single iteration of both the FIR filter and AES implementations as they perform enough processing in a single iteration to raise the problem of resets with limited energy buffers. We feed the FIR filter code with 116 bytes of dummy data, whereas AES processes 100 bytes of dummy data for both encryption and decryption. We consider these sizes as they are comparable to the size of a radio packet.

These benchmark codes do not take decisions based on sensed values or perform actuation. To introduce some degree of unpredictability, we experiment with a further custom version of the Kalman filter code, where we artificially modify the executions so that there is a 75% probability that a dummy interrupt handler worth 10000 MCU cycles preempts the execution. The latter version serves to measure the performance of the rules in Section 4.3 in an extreme case.

We consider an ARM Cortex M3 MCU aboard an ST Nucleo L152RE board, equipped with a standard flash chip as stable storage. The board is not specifically designed for transiently-powered operation; however, here we are only interested in the MCU, which represents state-of-the-art technology and was recently employed in designs with power consumption comparable to earlier 16-bit platforms [5]. To this end, ST Nucleo boards crucially provide a range of hooks useful to monitor the MCU execution and isolate its power consumption. Using the ST-Link in-circuit debugger, Kiel uVision, and a Tektronix TBS 1072B oscilloscope, we can obtain very fine-grained information on the real hardware execution, including all the information required at compile-time as described in Section 3. This is key to the accuracy of the results.

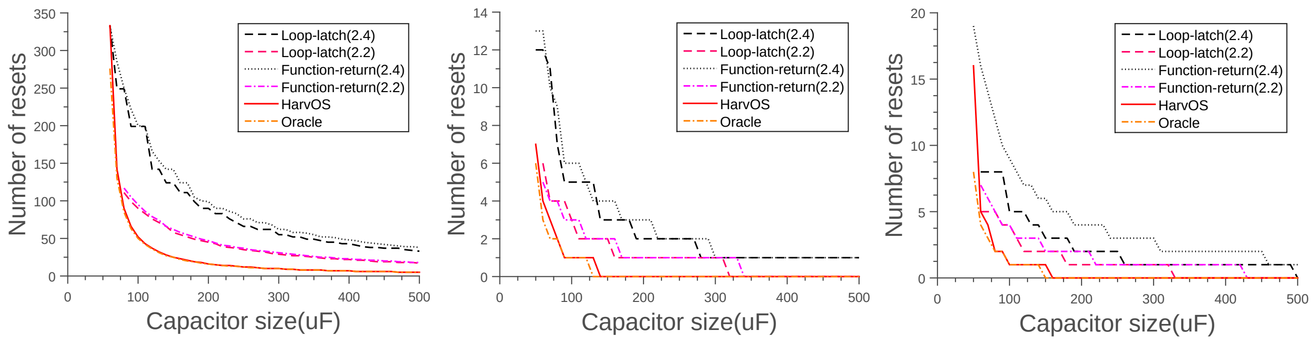
Metrics. Based on the information gathered with the setup above, we consider a variable size for a capacitor used as energy buffer that we assume to fully recharge every time is exhausted. Then, similar to existing works [6, 20], we synthetically emulate the execution of the code and compute three key metrics:

(1) The *number of times the MCU resets* because the capacitor needs to recharge to complete the fixed workload. This figure is inversely proportional to the effectiveness of a given instrumentation strategy. Given a fixed workload, the more the MCU needs to reboot, the more the checkpointing operation is subtracting energy from useful computations. Lower values are thus better.
(2) The *number of wasted cycles* because the MCU exhausts the energy before reaching the next trigger call. This figure is again inversely proportional to the effectiveness of a certain solution. The higher is this figure, the more a given instrumentation strategy is failing in accurately identifying the last useful moment where to possibly checkpoint. Lower values are thus again better.
(3) The *minimum size of the energy buffer* that allows the MCU to complete the workload under a given instrumentation strategy. For example, depending on how the trigger calls are placed, with small capacitor sizes, the execution may not reach even the first trigger call. This means checkpointing never happens and the system is stuck in a live-lock situation, rebooting every time from the initial state.

Baselines. We consider the *loop-latch* and *function-return* strategies of MementOS, described in Section 2, as representative of current approaches.

Note that the voltage threshold MementOS considers to decide whether to checkpoint is the result of repeated emulation experiments ran with a specific application code and user-supplied energy traces [20]. HarvOS does not require users to supply similar traces, which may be hard to obtain in the first place. Our solution is rather based on a worst-case analysis and assumes that the environment provides no additional energy once the device can reboot.

Considering this specific energy supply pattern also in the evaluation, as we do, does not impact the results. Should the environment provide new energy after the device reboots, equation (1) in Section 3 would capture the new supply of energy as part of $E_{remaining}$. Similarly, the new supply of



(a) Kalman filter. **(b) FIR filter.** **(c) AES.**

Figure 8: Number of resets necessary to complete a fixed workload. HarvOS *improves by a 69% factor on average, with a peak improvement of 80% compared to MementOS, while performing close to the oracle.*

energy would affect the execution of MementOS as soon as it is sufficient to move the operating voltage above the threshold.

To study the performance of MementOS in a manner orthogonal to the availability of energy traces, in our experiments we manually vary the value of MementOS voltage threshold. This way, we are likely to cover also the specific setting that MementOS would identify given a specific application code, and orthogonally to energy traces. We experiment with multiple such thresholds, always above the minimum voltage that still allows the system to write onto the flash chip. Settings of or below 2.2V, in particular, were never seen for MementOS in related literature [6, 20], and are likely to play favorably to it.

In addition, we apply a brute-force search on all possible executions of the code to identify an *oracle* that, by predicting how the execution is going to unfold in the future, knows the last practical point in time when to checkpoint. This is not feasible in reality; to make it work in a concrete execution, one would theoretically need to place a trigger call after every instruction in the code, yielding an unbearable overhead.

5.2 Results

Number of resets. Figure 8 plots the results we obtain in the number of MCU resets for a fixed workload, with no preemption due to interrupt handlers. As expected, bigger capacitor sizes generally correspond to fewer resets, in that individual executions progress farther on a single charge.

Compared to either of the MementOS strategies, HarvOS completes the fixed workload with 69% fewer resets on average, with a peak improvement of 80% fewer resets. On the other hand, certain configurations exist, especially for the FIR filter code in Figure 8b and for the AES implementation in Figure 8c, where the performance is the same. Yet, HarvOS never performs worse than MementOS in our experiments.

Moreover, in most cases the performance of our solution rests very close to the *oracle*. Notably for the Kalman filter code, the performance is often almost the same. This demonstrates that the rationale explained in Section 3 strikes an effective trade-off between opposite needs, ultimately performing similarly to an optimal solution that is, however, unfeasible in practice.

Comparing Figure 8a, obtained using the Kalman filter code, with Figure 8b that shows the performance with the FIR filter code as well as Figure 8c that depicts the performance of the AES implementation, one may note that the performance of HarvOS is robust against diverse benchmark codes. The Kalman filter code is 1053 lines of C code and includes a few function calls at the outmost level of the code structure; the size of a checkpoint is 442 bytes. The AES implementation is 2848 lines of C code and it includes two loops and multiple function calls at the outmost level of the code structure; the size of a checkpoint is 624 bytes. The FIR code is 14986 lines of C code as it includes several digital signal processing functions part of a larger library, but includes two loops with fewer function calls than AES at the outmost level of code structure; the size of a checkpoint is 568 bytes.

MementOS, on the other hand, shows a different behavior in terms of how the program structure affects the performance. For a certain voltage threshold, the performance of both MementOS strategies is very similar for the Kalman filter code in Figure 8a and for the FIR filter code in Figure 8b. In contrast, *loop-latch* performs distinctively better than *function-return* for the AES implementation, as shown in Figure 8c. In fact, one should generally not only emulate different voltage thresholds for configuring MementOS, but also try with different instrumentation strategies to find the best performing configuration [20]. The process may then become laborious.

To complement these results, Figure 9 depicts the number of resets necessary to complete the Kalman filter workload with a 75% probability that the main code is preempted by a dummy interrupt handler. As MementOS does not explicitly account for interrupt handlers, we can only compare against the oracle here. Compared to Figure 8a, the performance is remarkably similar. The strategy described in Section 4.3, as a result, bears minimal additional overhead.

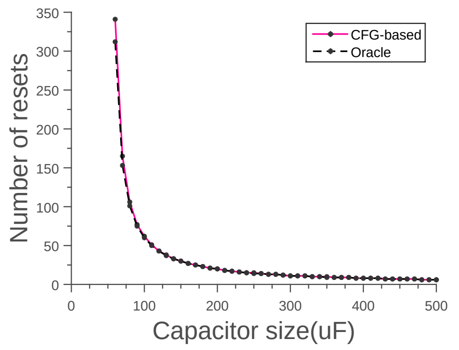


Figure 9: Number of resets necessary to complete a fixed workload with the implementation of Kalman filter in preemptable executions. *We emulate a 75% probability of preemption by an interrupt handler. The performance is close to that of Figure 8a with no interrupts. The rules of Section 4.3 bear minimal additional impact.*

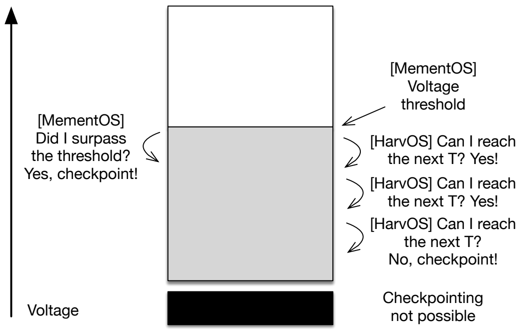

[MementOS] Voltage threshold
[MementOS] Did I surpass the threshold? Yes, checkpoint!
[HarvOS] Can I reach the next T? Yes!
[HarvOS] Can I reach the next T? Yes!
[HarvOS] Can I reach the next T? No, checkpoint!
Voltage
Checkpointing not possible

Figure 10: Graphically comparing the behavior of MementOS against HarvOS. *The voltage threshold in MementOS defines a "grey area" that corresponds to a mandatory checkpoint as soon as it is entered. While this threshold applies globally to the whole program and may penalize specific executions by checkpointing too early,* HarvOS *can postpone the decision to checkpoint based on the specific situation.*

Explaining the improvements. The gains over MementOS are intuitively explained in Figure 10. MementOS employs a *single* voltage threshold in all of its trigger calls. Such threshold defines a "grey region" of energy supplies that, during the off-line emulation experiments, were found *at least once* to *immediately* require a checkpoint, or the device would die before reaching the next trigger call with sufficient energy to checkpoint. As soon as MementOS enters the grey region and a trigger call is executed, a checkpoint takes place without checking whether it can reach the next trigger call given the available energy.

In contrast, our solution works in a localized fashion. Once we enter in what MementOS would consider the grey area, every trigger call results in a checkpoint *only if* the available energy is insufficient, in the worst-case, to reach the next trigger call and to checkpoint there. This is, in essence, what equation (1) in Section 3 stipulates. This reasoning allows us to postpone checkpointing in time and space, essentially "digging" down into the grey area as long as possible.

Wasted cycles. Figure 11 shows the number of wasted cycles against the capacitor size. As expected in light of the discussion of Figure 10, at higher threshold voltages, MementOS wastes a higher number of cycles because checkpoints tend to happen too early, leaving unused energy in the buffer.

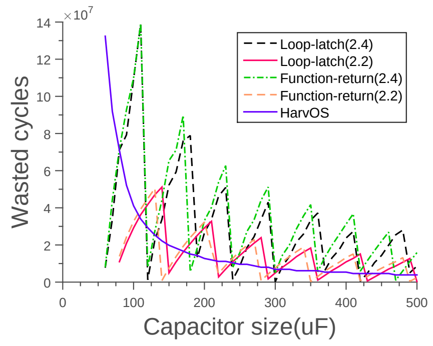
Moreover, the staircase pattern in MementOS, especially visible for the Kalman filter code in Figure 11a and the FIR filter code in Figure 11b, appears because both its strategies are too coarse-grained. Either the trigger call is located at the "right" place, and so there are only a few wasted cycles, or it is located at "wrong" place and so a lot of unused energy remains in the buffer. In fact, MementOS placement strategies only look at the structure of the code and not at its energy consumption patterns.

Differently, with smaller capacitors, HarvOS results in a higher number of wasted cycles than MementOS because equation (1) in Section 3 often finds $E_{remaining}$ insufficient to reach the next trigger call *in the worst case.* The value of $E_{remaining}$ is, in fact, upper-bound by the size of the capacitor. The worst-case analysis we apply is here counter-productive, in that it tends to be excessively pessimistic. The larger is the capacitor, however, the less this issue affects the operation of HarvOS, yielding increasingly precise checkpoint decisions that reduce the number of wasted cycles.

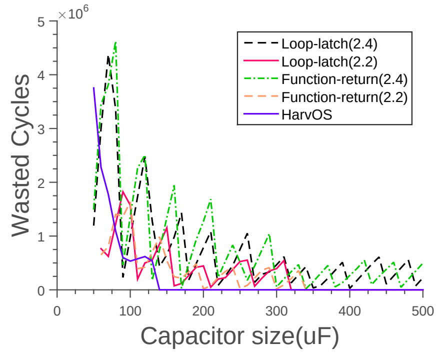
Minimum size of energy buffer. Figure 12 reports the minimum capacitor size required to complete the three benchmarks, against different threshold voltages for MementOS.

When operated under lower threshold voltages, MementOS finishes the execution only with bigger capacitors. This is essentially a result of its logic for placing triggers and of basing the decision to checkpoint on voltage levels. Without reasoning on the energy necessary to reach the next trigger call, MementOS may place trigger calls too far from each other. Under lower threshold voltages, the execution may then continue "blindly" up to a point when insufficient energy is left to complete the checkpoint, namely it is too late to checkpoint. To remedy this, a bigger capacitor is needed.

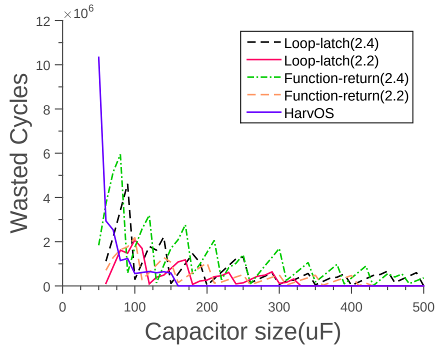
In contrast, our solution allows one to employ smaller capacitors. The splitting of the CFG in sub-graphs whose stretch is at most equal to the number of cycles the MCU can execute in a worst-case situation, as explained in Section 3, rules out the possibility of placing trigger calls too far apart. In fact, our performance in Figure 12 is independent of the voltage threshold used by MementOS. At each trigger call, we again decide whether to continue the execution or to checkpoint based on the ability to reach the next trigger call with sufficient energy to checkpoint there, as equation (1) indicates. The ability to complete a given workload with smaller capacitors may be particularly beneficial in applications requiring quick recharge times.



(a) Kalman filter.

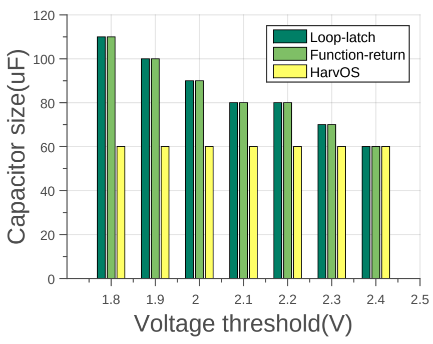


(b) FIR filter.

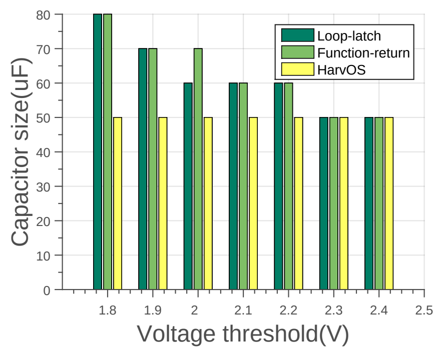


(c) AES.

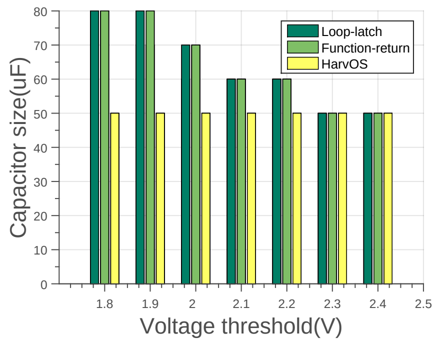
Figure 11: Number of wasted cycles. *Both MementOS strategies are too coarse-grained and oblivious of energy consumption patterns. Differently, basing the checkpointing decisions on the ability to reach the next trigger call yields increasingly precise decisions with bigger capacitors.*



(a) Kalman filter.



(b) FIR filter.



(c) AES.

Figure 12: Minimum capacitor size required to complete the three benchmarks. *MementOS places trigger calls too far from each other, and thus requires bigger capacitors to complete the workload. Our placement rules allows the system to complete the workload with smaller capacitors.*

6 CONCLUSION

HarvOS operates at compile-time based on the control-flow graph (CFG) of the program. Trigger calls are placed by looking at the worst-case energy cost required to reach the next trigger call and depending on the program structure as represented in the CFG. The information collected at compile-time also enables to spare the energy-intensive probing of the energy buffers whenever possible. The combination of these techniques allows the system to take informed decisions at every trigger call on whether to continue with the normal execution or to rather checkpoint. Our evaluation of the approach, based on three diverse benchmarks, indicates that our techniques allow transiently-powered devices to complete a given workload with 68% fewer checkpoints, compared to existing literature. Our performance also rests only 19% far from that of an "oracle" that would know exactly the last point in time when a checkpoint is required. The performance of HarvOS may further improve by removing, whenever possible, the worst-case assumption at the cost of additional code instrumentation. We consider exploring the trade-off between simplicity of instrumentation and the additional information it may provide as a direction for future work.

Acknowledgments. This work was partly supported through the Google Faculty Award program, by the Swedish Energy Agency, and by projects "Zero-energy Buildings in Smart Urban Districts" (EEB), "ICT Solutions to Support Logistics and Transport Processes" (ITS), and "Smart Living Technologies" (SHELL) of the Italian Ministry for University and Research.

REFERENCES

[1] 2016. AES C Implementation for Mbed platform. goo.gl/PBjhoF. (2016).
[2] 2016. FIR Filter C Implementation for Mbed platform. goo.gl/yFyUyX. (2016). Accessed: 10-13-2016.
[3] 2016. Kalman Filter C Implementation for Mbed platform. goo.gl/ikzYFt. (2016). Accessed: 30-9-2016.
[4] 2016. mBed OS energy profiler. goo.gl/dghhd4. (2016). Accessed: 11-10-2016.
[5] Michael Andersen and others. 2016. System design for a synergistic, low power mote/BLE embedded platform. In *IPSN*.
[6] Alberto Arreola and others. 2015. Approaches to Transient Computing for Energy Harvesting Systems: A Quantitative Evaluation.

In *ENSSYS*.
[7] Domenico Balsamo and others. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2015).
[8] Naveed Anwar Bhatti and others. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM TOSN* 12 (2016). Issue 3.
[9] Naveed Anwar Bhatti and Luca Mottola. 2016. Efficient state retention for transiently-powered embedded sensing. In *EWSN*.
[10] Bernhard Buchli and others. 2013. Battery state-of-charge approximation for energy harvesting embedded systems. In *EWSN*.
[11] Jack Davidson and Sanjay Jinturkar. 1995. Improving Instruction level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation. In *MICRO*.
[12] Prabal Dutta and others. 2008. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN*.
[13] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *SENSYS*.
[14] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *POPL*.
[15] Martin Hofmann and Steffen Jost. 2006. Type-based amortised heap-space analysis. In *ESOP*.
[16] Hrishikesh Jayakumar and others. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *VLSI Design*.
[17] Vincent Liu and others. 2013. Ambient Backscatter: Wireless Communication out of Thin Air. In *SIGCOMM*.
[18] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *PLDI*.
[19] Saman Naderiparizi and others. 2016. μMonitor: In-situ Energy Monitoring with Microwatt Power Consumption. In *RFID*.
[20] Benjamin Ransford and others. 2011. MementOS: System Support for Long-running Computation on RFID-scale Devices. In *ASPLOS*.
[21] Philipp Sommer and others. 2016. Information Bang for the Energy Buck: Towards Energy-and Mobility-Aware Tracking. In *EWSN*.
[22] Jing Yang and others. 2007. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *SENSYS*.