Efficient State Retention 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 state retention techniques to support embedded sensing applications on 32-bit microcontrollers whose energy provisioning is assisted through ambient harvesting or wireless energy transfer. As energy availability is likely erratic in these settings, applications may be unpredictably interrupted. To behave dependably, applications should resume from where they left as soon as energy is newly available. We investigate the fundamental building block necessary to this end, and conceive three mechanisms to checkpoint and restore a device's state on stable storage quickly and in an energy-efficient manner. The problem is unique in many regards; for example, because of the distinctive performance vs. energy trade-offs of modern 32-bit microcontrollers and the peculiar characteristics of current flash chips. Our results, obtained from real experiments using two different platforms, crucially indicate that there is no "one-size-fits-all" solution. The performance depends on factors such as the amount of data to handle, how in memory the data is laid out, as well as an application's read/write patterns.

1 Introduction

Progresses in micro electro-mechanical systems are redefining the scope and extent of the energy constraints in networked embedded sensing. Technologies to harvest energy from the ambient can integrate with embedded devices to refill their energy buffers. A variety of these technologies appeared that apply to, for example, light and vibrations, while matching the physical constraints of the devices [14, 16, 20]. Wireless energy transfer complements these techniques by enabling opportunistic recharges. Several techniques recently appeared that enable practical wireless energy transfer at scales suitable for embedded sensing [11, 23, 24].

These technologies, however, can rarely ensure a predictable supply of energy. Computing under such transient energy conditions becomes a challenge. Devices experience frequent shutdowns, to later reboot as soon as energy is newly available. In the mean time, applications lose their state. This translates into a lack of dependable behavior and a waste of resources, including energy, as applications need to re-initialize, re-acquire state, and perform re-synchronization with other nearby devices. As a result, even if an application ultimately manages to make some progress, the overall system performance inevitably suffers.

Meanwhile, embedded sensing systems are increasingly built around modern 32-bit microcontrollers (MCUs), such as those of the ARM Cortex-M series [3]. These provide increased computing power and larger amounts of memory compared to earlier 16-bit MCUs, at a modest increase in energy consumption. These features enable employing more sophisticated algorithms and programming techniques, facilitating more demanding embedded sensing applications in several that require dependable behaviors, including wireless control [1, 7] and Internet-connected sensing [2].

In this context, we aim at allowing an application's processing to cross the boundaries of periods of energy unavailability. We wish to do so without resorting to hardware modifications that may greatly impact costs, especially at scale. Solutions to similar issues exist, for example, in the domain of computational RFIDs [28, 29], whose applications and hardware characteristics are, however, sharply different from the platforms above. As further elaborated in Section 2, the net result is that existing solutions are hardly applicable.

In this paper, we study the fundamental building block to reach the goal, and investigate efficient system support to checkpoint an application's state on stable storage, where it can be later retrieved to re-start the application from where it left. Two requirements are key for these functionality:

1. they must be *energy-efficient* not to affect the duration of the next computing cycle; indeed, the energy spent in checkpointing and restoring is subtracted to the energy budget for computing and communicating.
2. they need to execute *quickly* to minimally perturb the system; as the time taken to complete the routines grows, applications may be increasingly affected as they are often not designed to be preempted.

As described in Section 3, the checkpoint and restore routines we design are made available to programmers through a single pair of **`checkpoint()`** and **`restore()`** func-

International Conference on Embedded Wireless Systems and Networks (EWSN) 2016
15–17 February, Graz, Austria

ISBN: 978-0-9949886-0-7

tions. Key to their efficiency is the way the state information is organized on stable storage. Embedded devices are indeed typically equipped with flash chips as stable storage, which are energy-hungry and offer peculiar modes to perform read and write operations. Section 4 describes three dedicated storage modes that exploit different facets of how data is laid out on modern 32-bit MCUs and of the energy consumption characteristics of current flash chips.

We study the trade-offs among the three schemes and two baselines taken from the literature through real experiments using two different platforms. Our results, reported in Section 5, provide evidence of several trade-offs that depend, for example, on the amount of data to handle and an application's read/write patterns. Section 6 discusses these trade-offs based on our results, and provides insights on what kind of application may benefit most from what storage mode.

Section 7 ends the paper with an outlook on future work and brief concluding remarks.

2 Background

Our work targets modern embedded platforms, whose characteristics depart from traditional mote-class devices. Differently, existing software techniques for state retention on transiently-powered devices mostly target computational RFIDs, whose programming techniques and resource constraints do not match those of the aforementioned platforms.

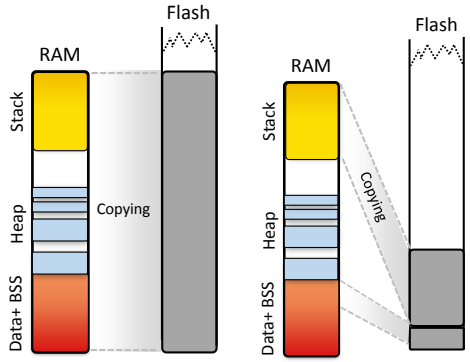
2.1 Target Platforms

We consider 32-bit MCUs of the ARM Cortex-M series as representatives of modern embedded sensing platforms. This specific breed of MCU is gaining momentum [2], due to excellent performance vs. energy consumption trade-offs.

ARM Cortex-M. We use two STM32 Cortex-M prototyping boards, one ST Nucleo L152RE board equipped with a Cortex-M3 MCU, and one ST Nucleo F091RC board equipped with a Cortex-M0 MCU. The two boards represent, in a sense, opposite extremes within the Cortex-M family. The Cortex-M3 board offers higher processing power, 80 KBytes of RAM space, and maximum energy consumption of 0.365 mA/MHz. Differently, the Cortex-M0 board has more limited processing capabilities, 32 KBytes of RAM space, and maximum energy consumption of 0.31 mA/MHz.

The Cortex-M design provides sixteen core registers. The first thirteen registers (R0-R12) are 32-bit General-Purpose Registers (GPRs) for data processing. The *Stack Pointer* register (SP, R13) tracks the address of the last stack allocation in RAM. The *Link Register* (LR, R14) holds the address of the return instruction when a function call completes, whereas the *Program Counter* (PC, R15) holds the address of the currently executing instruction.

The characteristics of Cortex-M MCUs as well as the availability of dedicated development environments [2, 4] and efficient compilers [17] are impacting existing embedded programming techniques. A paradigmatic example is the use of heap memory. Traditionally discouraged because of overhead and lack of predictable behavior, it is increasingly gaining adoption [9, 18]. Besides providing better programming flexibility, heap memory allows developers to employ sophisticated programming languages and techniques, such as object orientation with polymorphic data types and exception handling [22]. Moreover, it facilitates porting existing libraries, such as STL containers, to embedded systems [18].



(a) Full RAM [27]. (b) Stack-based [28].

Figure 1. Existing checkpointing techniques.

Flash memory. Representative of existing platforms is also the kind of stable storage aboard both boards we use. The MCU is connected to a NAND-type flash memory chip through a dedicated instruction bus, optimized for smaller chip size and low energy cost per bit.

This kind of flash memories are divided into *sector*s, which are then sub-divided into *page*s. The two units determine the read/write modes. The flash chip on the Cortex M3 board requires to write half of the *page* size at a time, whereas the flash chip on the Cortex M0 board permits writes of a 32-bit word in a single turn. This complicates saving arbitrary amounts of data on the flash. Moreover, the written data cannot be modified in-place as in RAM; data needs to be erased before re-writing. The unit size of an erase operation is, however, different than the unit size for writes, which further complicates matters. For example, the Cortex M0 board requires the erase of an entire 2 KByte sector at a time, possibly to modify a single bit in a sector.

These aspects combine with the peculiar energy consumption of flash chips: write and erase operations are slow and extremely energy-hungry, whereas read operations takes significantly shorter time and consumes less energy. To put things in perspective, the flash chip of the Cortex-M3 board draws 11.1 uA/MHz, that is, orders of magnitude more than any other peripheral on the board.

2.2 Prior Art

Checkpointing and restoring the system's state is not a new concept. In database systems, for example, these mechanisms are used for ensuring the consistency of concurrent transactions on replicated databases [10]. In distributed debugging, checkpointing aids identifying root causes by providing the input to re-play concurrently executing processes [21]. Checkpoints are also used for ensuring fault-tolerance in redundant real-time embedded systems, such as those interconnected via wired buses [8].

Checkpoint and restore techniques are often reported for testing and experimentation using mote-class devices. For example, Osterlind et al. [27] present a checkpointing scheme similar to the one in Figure 1(a), where the entire

RAM space is transferred to stable storage. The objective is to facilitate transferring network state between testbeds and simulations, thus achieving increased repeatability. Their technique targets TMote Sky nodes. However, dumping the entire RAM space onto stable storage is likely inefficient, as the procedure also includes empty areas of memory that do not need to be saved. Moreover, the work of Osterlind et al. [27] does not necessarily support resuming the execution from the point in the code where the last checkpoint is taken, which is however required in our setting.

Chen et al. [13] augment the TinyOS operating system with mechanisms to checkpoint and restore selected components upon recognizing state inconsistencies. The mechanisms to trigger the checkpoints are generally application-specific, and meant to describe the conditions that indicate data faults. In our setting, the motivation for checkpoint and restore is different; it originates from a lack of the energy necessary to continue the computation, rather than data faults. As a result, we do not aim at checkpointing selected components, but the entire application state so that a device can survive periods when it completely shuts down.

Existing works closest to ours target computational RFIDs equipped with 16-bit MCUs and small amounts of memory, such as the WISP mote [12]. For these platforms, MementOS [28] allows programmers to inserts "trigger points" to save programmer-selected parts of the BSS or DATA sections and the stack onto stable storage, as shown in Figure 1(b). As it only handles contiguous areas of memory, the processing is quite simple. It is, however, inapplicable to our case. For example, we are to include also heap memory as part of checkpoint and restore. This creates issues such as how to cope with fragmentation in the heap, which are specific to the setting we consider in this work.

To ameliorate the energy overhead of flash memory, Quickrecall [19] resorts to hardware modifications by replacing traditional SRAM with non-volatile ferroelectric RAM chips. However, ferroelectric RAM is currently significantly less dense and more expensive compared to normal SRAM, which makes it less desirable for high-performance embedded devices, such as those built with Cortex-M MCUs. Furthermore, memory-mapped FRAM will create data inconsistencies among non-volatile data variables. Lucia et al. [25] ensure volatile and non-volatile data consistency by asking the programmer to manually place the calls to the checkpoint routines within the code. In this work, we intend to use non-volatile memory as a support, not a replacement of RAM. We only use non-volatile memory for storing and retrieving checkpoints, rather than supporting general computations.

3 Fundamental Operation

We describe the choice of the minimum state information required for correctly enabling a subsequent restore, as well as the fundamental operation of the checkpoint and restore routines. The former are independent of how the state information is mapped to stable storage. We deal with this aspect in Section 4, by presenting three different storage modes.

Our target platforms employ a plain memory map. The program data is divided into five segments: DATA, BSS, heap, stack, and TEXT. The DATA segment includes initialized global and static variables, and is typically located at the starting address of the RAM. The BSS segment is located adjacent to the DATA segment and includes uninitialized global variables. At the end of the BSS segment starts the heap segment, dedicated to dynamic data. The stack segments starts from the bottom address of the main memory and grows towards the heap. The TEXT segment resides in a flash-type memory and holds the program instructions.

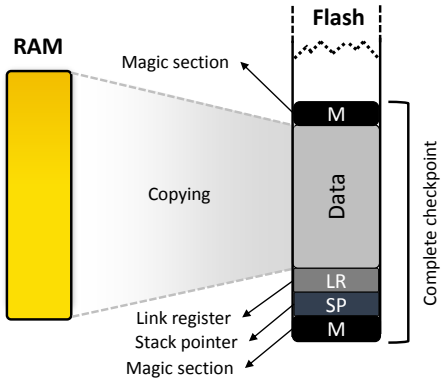


Figure 2. Fundamental operation during checkpoint.

In principle, the minimum state information for later restoring the device's state includes: *i)* the values of all GPRs, *ii)* the content of the RAM, including stack, heap, and the BSS and DATA segments, and *iii)* the values of stack pointer (SP), program counter (PC), and link register (LR). To checkpoint the device's state, we initially push the values of all GPRs onto the stack through an assembly function—this is the only device-specific step in the whole procedure. Next, we proceed backward from the last address of the stable storage, as in Figure 2. In doing so, we:

1. save a "magic section" [28] on the last address of the stable storage, which includes a randomly generated number and the size of the RAM data we need to store—this information is used to ensure a checkpoint is complete when restoring, as explained next;
2. save the current values of stack pointer and link register: as described later, these two are sufficient to resume the computation using the checkpoint information;
3. copy to stable storage the RAM data, including stack, heap, the BSS and DATA segments, as well as the values of the GPRs we copied to RAM earlier;
4. save the same "magic section" again and pop the GPR values from the stack back into their respective registers, so the program can resume its normal execution.

These operations are made available through a single C function **`checkpoint()`** that takes the value of the stack pointer (SP) as an argument[1]. The reason why the function requires this value is because the call to the function itself affects the stack pointer. However, a checkpoint must resume the computation right after the call to **`checkpoint()`**, that is, in a situation where the stack pointer holds the same value

[1]For programming convenience, this information is provided through a C macro.

as before the call. Because of this, it is also not necessary to save the value of the program counter (PC). The link register (LR), which holds the return address of the call to **`checkpoint()`**, carries precisely the point in the program where we wish to resume the computation after restoring.

To restore the device's state, we provide a symmetric C function **`restore()`**, which is to be called immediately after the device starts running the **`main()`** function. The key functionality is to ensure that only complete checkpoints are restored. It may indeed happen that **`checkpoint()`** is called when the energy left on the device is insufficient to complete the operation, and the device turns off before the function finishes. In these circumstances, the data on the stable storage cannot be used to resume the computation: a partial restore may ultimately bring the device to an inconsistent state that prevents any other progress.

To address this issue, the **`restore()`** function proceeds in the opposite way compared to **`checkpoint()`**. It first reads the magic section. Based on this, it calculates the size of the whole checkpoint and retrieves the other copy of the magic section at the end of the checkpoint. If the two copies of the random number in the magic section are equal, it means the checkpoint data is complete, that is, the **`checkpoint()`** function correctly reached the end of its processing. Only in this case, **`restore()`** proceeds by reading the data from the checkpoint to re-populate the RAM space and to update the stack pointer (SP) as well as link register (LR). Setting the latter to the instruction immediately following the call to **`checkpoint()`** makes the program resume as if the computation was never interrupted.

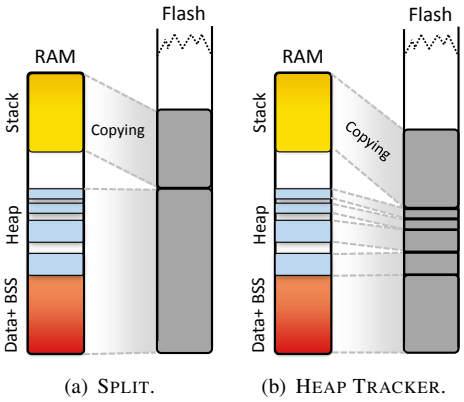
4 Storage Modes

The crucial aspect determining the performance of the checkpoint and restore routines is the organization of the state information on stable storage. We design three storage modes, described later and illustrated in Figure 3. In Section 5 we report on extensive experimental results revealing several performance trade-offs.

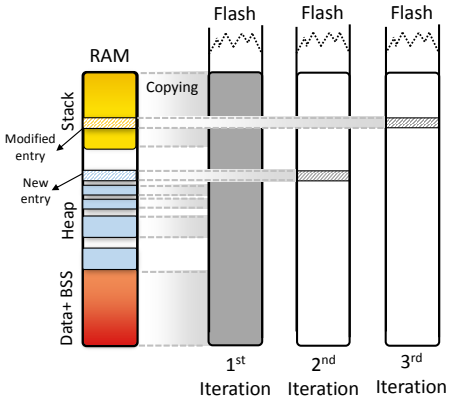
4.1 Split

To include the heap segment in the checkpoint, the most natural optimization over copying the whole RAM space [27] is to split the operation between the stack, heap, and the BSS and DATA segments. This allows one not to write to stable storage the unused memory space between the end of the heap and the top of the stack, avoiding unnecessary energy-hungry write operations.

Based on this reasoning, as shown in Figure 3(a), the SPLIT mode processes the stack information and the rest of the memory segments separately. First, it copies the whole stack segment to stable storage. This is possible because, as explained in Section 3, the checkpoint routine receives the current value of the stack pointer as input. Next, we copy the DATA, BSS, and heap segments as a whole to stable storage. To this end, the checkpoint routine needs to know the highest allocated address in the heap segment. We gain this information by wrapping **`malloc()`** and **`free()`** with the functionality to keep track of this address as memory is allocated and deallocated during the application's lifetime. The checkpoint routine then simply copies everything below this address up to the RAM start address.



(a) SPLIT. (b) HEAP TRACKER.



(c) COPY-IF-CHANGE.

Figure 3. Storage modes.

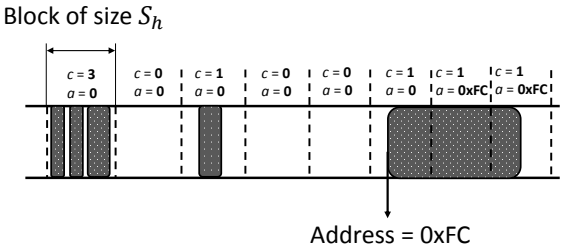


Figure 4. Example configuration in HEAP TRACKER when allocated memory crosses multiple blocks.

Trade-offs. The processing required by SPLIT is extremely simple. Moreover, the additional state information to be kept is minimal: it solely amounts to keeping track of the highest memory address allocated in the heap. On the other hand, the custom **`malloc()`** and **`free()`** system functions introduce some slight processing overhead. In addition, space is still wasted on stable storage if the heap segment is fragmented. Again, unnecessary writes may be detrimental to the system's lifetime and, particularly, to the ability of the checkpoint routine to correctly complete.

4.2 Heap Tracker

To overcome the potential energy waste due to writes of fragmented areas of the heap, we must achieve higher granularity in keeping track of allocated and deallocated memory.

This is, however, challenging in the general case. It is indeed quite complex to predict allocation and deallocation operations in the heap, or to forecast the size of the allocated or deallocated chunks of memory.

To address these issues, we conceive a simple, yet effective scheme called HEAP TRACKER, intuitively illustrated in Figure 3(b). We split the heap segment in blocks of size S_h, and create a supporting data structure m with M/S_h entries, M being the maximum size of the heap. Each entry $m[i]$ carries two pieces of information: a 1 byte integer counter c and a memory address a.

The counter c records the number of memory chunks allocated in the i-th heap block. The checkpoint routine checks this information before copying the block to stable storage, and performs the operation only if c is greater than zero. A counter is necessary, rather than simply a flag, because in the general case S_h may be larger than the size of the allocated chunks of memory, so a single block may accommodate multiple allocations. The counter for every block is incremented or decremented upon allocating or deallocating memory within the block's boundaries, again through proper wrappers to `malloc()` and `free()`.

The address a serves the cases where allocations and deallocations cross multiple blocks. For example, the right part of Figure 4 shows a case where a chunk of memory is allocated across three blocks. In this situation, the counters of all affected blocks are to be incremented upon allocating memory, and vice-versa when deallocating. The operation is simple in the former case, but special care needs to be taken when deallocating. Indeed, unless one modifies the internal implementation of `malloc()` and `free()`, which we would rather avoid for better portability, it is difficult to know the size of the deallocated memory when `free()`-ing. To address this, the memory address a is set to a value corresponding to the one returned by the original `malloc()` when allocating the crossing chunk. In a sense, it indicates where the crossing chunk starts out of the current block. This way, the wrapper for `free()` can recognize the situation based on the function's input argument, and proceeds decrementing the counter for all blocks where a matches.

Note that, as the data structure m is in the DATA segment, it implicitly becomes part of the checkpoint. The restore routine uses this information to reconstruct the heap, including the fragmented areas, before resuming the computation.

Trade-offs. The choice of the value for S_h greatly impacts the performance of HEAP TRACKER. Larger values for S_h decrease the size of the supporting data structure, thus alleviating the memory overhead due to additional state information. However, the achieved granularity may still cause some un-allocated space to be written to stable storage if memory is allocated in chunks smaller than S_h. Conversely, smaller values for S_h ameliorate this issue, but increase the size of the additional state information required by HEAP TRACKER. This makes the checkpoints larger, and thus increases the energy required when writing to stable storage.

Orthogonal to this trade-off is the fact that if the block size S_h does not align with the smallest writeable unit on stable storage, the same heap block may require multiple writes on stable storage unit, as discussed in Section 2.1, causing unnecessary energy overhead. Among these conflicting requirements, we choose to optimize the energy spent in writing the memory blocks to stable storage, and set S_h equal to the size of the smallest writeable unit of stable storage. Based on the maximum heap size allowed by the compiler we use, this creates an overhead of 3200 (1280) Bytes for the Cortex M3 (M0) board, which is at most 4% of the available RAM.

4.3 Copy-If-Change

A different take at the problem is to try and understand whether a write to stable storage is needed at all. It may indeed be the case that the previous checkpoint already includes the same information, thus re-writing to stable storage is unnecessary. This reasoning finds justification in some of the characteristics of modern flash chips, as discussed in Section 2.1, where read operations are often more quick and energy-efficient than writes. Thus, trading the energy necessary to read from the previous checkpoint to possibly avoid a write may be beneficial overall.

To leverage this aspect, COPY-IF-CHANGE splits the *entire* RAM space in blocks of size S_c again equal to the size of the smallest writable unit on stable storage. As illustrated in Figure 3(c), for each such block, COPY-IF-CHANGE first reads the corresponding memory block from the previous checkpoint if available, and compares that with the current content of the RAM. If the two differ, the block is updated on stable storage; otherwise, we proceed to the next block. In the first iteration, COPY-IF-CHANGE considers the previous checkpoint as empty, thus all blocks are updated.

Trade-offs. COPY-IF-CHANGE evidently incurs high overhead for the first checkpoint, as all the blocks appear as modified and need to be copied to stable storage. Conversely, the fewer modifications to the RAM, the more efficient the mode becomes, as more energy-hungry write operations are avoided. As experimentally verified in Section 5 and unlike the previous two modes, the energy performance of COPY-IF-CHANGE is also simple to predict, as it shows a basic relation with the number of modified memory blocks.

5 Evaluation

We discuss the experimental results we obtain by comparing the performance of the storage modes in Section 4 against each other, as well as with *i)* a mode equivalent to that of Österlind et al. [27] called FULL, whereby the entire RAM space is copied to stable storage regardless of how memory is occupied, and *ii)* a mode akin to MementOS [28] called STACK, whereby only the BSS, DATA, and stack segments are copied to stable storage.

The results we present next indicate that no single solution outperforms all others in all settings. Thus, the choice of what storage mode to employ depends on the application's characteristics. We provide an overarching discussion of these aspects, including examples of target applications for each storage mode, in Section 6.

Metrics and setup. We consider two metrics based on the requirements we elicit in the Introduction: *i)* the *energy consumption* for the checkpoint and restore routines, and *ii)* the *time* to perform the routines since the time of the call to the corresponding C function. Note that the impact of restore

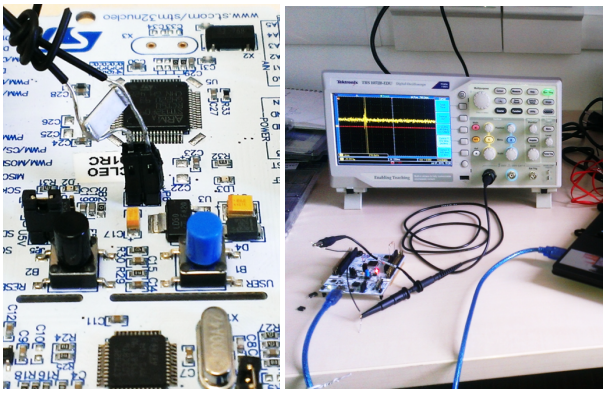


Figure 5. The 1Ω resistor in series with the IDD connector aboard the ST Nucleo boards and the measurement setup used in the evaluation.

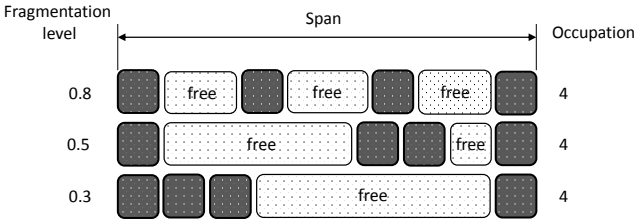


Figure 6. Memory configurations representing the same *span* and *occupation*, but different *fragmentation*.

operations on the overall system performance is generally much smaller than checkpoint ones. This is essentially because: *i)* as already mentioned, reading from flash memories is both faster and consumes less energy than writes, *ii)* restore operations generally happen with the node charged, as opposed to checkpoints.

The metrics are a function of both the energy spent to operate on the flash chip and by the MCU for processing. To compute them, we place a 1Ω resistor in series with a dedicated connector provided by the ST Nucleo boards, shown in Figure 5. A Tektronix TBS 1072B oscilloscope tracks the current flowing through the resistor. This allows us to accurately record both the energy absorbed and the time taken during checkpoint or restore. We set $S_h = 128$ Bytes for HEAP TRACKER, which corresponds to the smallest writeable unit on the flash chip of the Cortex M3 board. All values we present next are averages and error bars obtained over at least ten repetitions.

Memory configuration. Conceiving a thorough set of inputs for measuring the performance of the checkpoint and restore routines is only deceptively simple. Their functioning is indeed determined by the content of the memory when running the checkpoint, which is arbitrary. Quantitatively characterizing the relevant aspects for the DATA and BSS segments might not be difficult, as the data therein is necessarily contiguous and their size is known at compile time. This is not so for the stack, and especially for the heap.

We thus consider the model represented in Figure 6 to synthetically characterize the inputs to our experiments, and accordingly define three metrics that apply to either the stack or the heap segment:

1. the *span* indicates the memory interval from the first allocated chunk to the last one, that is, what portion of RAM space is covered by either segment.
2. the *occupation* measures the net amount of data found in memory within a given *span*. This corresponds to the *span* only for the stack, as the memory allocation is contiguous; the same does not hold for the heap as chunks of unallocated memory may be present.
3. for the heap, the *fragmentation* measures how allocated and unallocated memory chunks are distributed within the *span*; we quantitatively characterize this as

$$fragmentation(x) = 1 - \frac{x \times (\#\text{free chunks of size } x)}{(\text{total free bytes})} \quad (1)$$

where x is the size of the largest allocated memory chunk at the time of taking the measure.

Equation (1) evaluates to 0.0 in configurations where it is possible to allocate the maximum possible number of objects of size x, that is, the memory is not fragmented. Differently, it evaluates to 1.0 when it is impossible to allocate any chunk of size x, that is, memory is extremely fragmented.

Note that, for the heap, these metrics are orthogonal. For example, the same span may correspond to different occupations. Given a value of span and occupation, different configurations may yield different levels of fragmentation depending on the distribution of the allocated memory chunks, as illustrated in Figure 6.

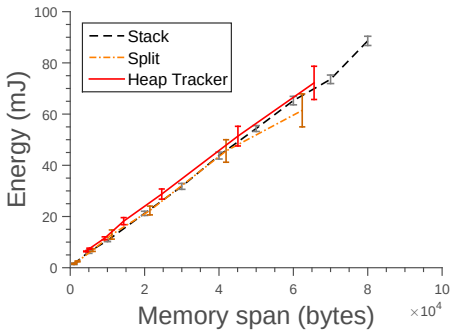
5.1 Contiguous Data

We investigate the performance of the different storage modes when RAM data is allocated in a contiguous manner. This is the case of applications whose memory demands are mostly known beforehand; in these cases, programmers tend to pre-allocate the necessary data structures. Differently, the case of non-contiguous data and general fragmentation are investigated in the following.

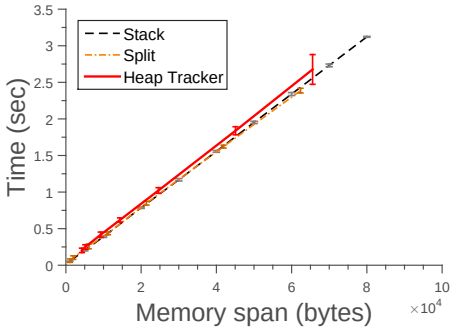
Setting. We vary together the *span* and *occupation* of different RAM segments to understand how these affect the performance. We test values within the limits allowed by either physical memory or the compiler we use, and operate differently depending on the storage mode.

For the STACK mode, we artificially increase the *span* of the stack by growing the size of local variables in a dummy function. As STACK does not consider the heap, its manipulation is indeed immaterial. For both SPLIT and HEAP TRACKER, we artificially increase the *span* and *occupation* of the heap by growing dynamically-allocated dummy structures, and keep the stack segment to the minimum. This is to investigate the performance as the varies; if it is empty, both SPLIT and HEAP TRACKER behave equivalent to STACK. For COPY-IF-CHANGE, we initially consider the first iteration, whereby all blocks are found to be different from the previous (empty) checkpoint, and later study the case of a varying number of blocks requiring an update. The FULL mode covers the entire RAM space anyways.

Results. Figure 7 summarizes the results obtained with the Cortex M3 board in energy and time, using STACK, SPLIT, and HEAP TRACKER. Overall, the values are quite limited.



(a) Average energy consumption.



(b) Average time taken.

Figure 7. Cortex M3: performance of the *checkpoint* routine with increasing *span* of contiguous RAM data. Heap fragmentation is 0. STACK *and* SPLIT *show similar performance in this setting, whereas* HEAP TRACKER *suffers from the overhead of additional support data without being able to take advantage of it.*

A checkpoint that covers the *entire* RAM space is completed in slightly more than 3 secs, arguably resulting in a moderate disruption of the application processing. In all modes, writes to the flash chip dominate both energy and time; thus the two figures are highly correlated, as seen by comparing Figure 7(a) and 7(b).

All modes in Figure 7 show a linear increase in both metrics as the memory *span* grows. STACK and SPLIT follow each other closely, as their working principles are the same, that is, they copy memory segments as a whole. Differently, the performance of HEAP TRACKER is slightly, but constantly worse than STACK and SPLIT. This quantifies the trade-offs discussed in Section 4.2, as it represents the price for: *i)* storing the support data structure that indicates what memory blocks are occupied in addition to the application data, and *ii)* performing additional processing on such data structure during checkpoint. In absolute terms, the overhead is limited in a worst-case situation for HEAP TRACKER: the contiguous memory allocation prevents it from leveraging the ability to avoid copying some blocks if they do not cover chunks of allocated memory.

The energy performance of COPY-IF-CHANGE for the initial iteration and of FULL—not shown in the charts as they are independent of the memory *span*—is 94.5±1.34 *mJ* and 85.2±4.0 *mJ*, respectively. We are indeed considering a worst case also for COPY-IF-CHANGE, as all memory blocks

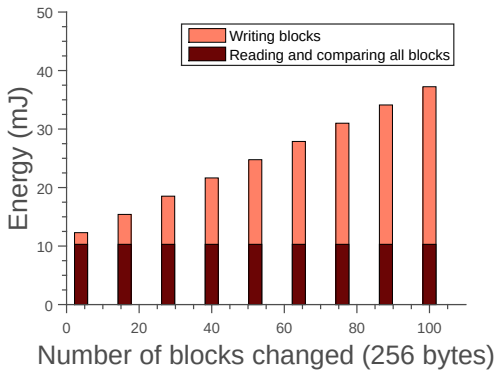
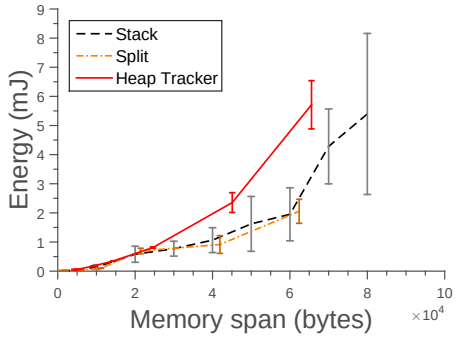
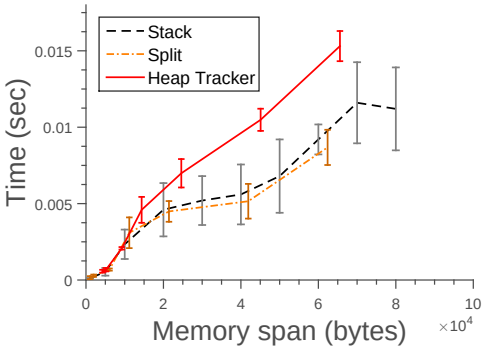


Figure 8. Cortex M3: energy consumption in COPY-IF-CHANGE against the number of blocks that need to be updated on flash. *The performance is a function of a fixed overhead for reading and comparing all blocks, plus the variable cost of re-writing those that are found modified.*



(a) Average energy consumption.



(b) Average time taken.

Figure 9. Cortex M3: performance of the *restore* routine with increasing *span* of contiguous RAM data. The heap fragmentation is 0. *Because of the small absolute values, the overhead of restoring the support data and reconstructing the heap becomes more visible for* HEAP TRACKER *as compared to* STACK *and* SPLIT.

are detected to be different from the previous (empty) checkpoint. In this case, COPY-IF-CHANGE also copies the entire RAM space. Unlike FULL, however, COPY-IF-CHANGE also needs to read all blocks from the previous checkpoint before deciding whether to update.

The performance of COPY-IF-CHANGE in the general

case is rather illustrated in Figure 8, where we artificially create a situation with a varying number of modified blocks in RAM, which thus require an update on flash when checkpointing. The performance corresponding to every value on the X-axis is a combination of fixed overhead caused by reading and comparing *all* the blocks, plus erasing and re-writing those that are found modified.

Figure 9 plots the performance of the restore routine in energy and time for the Cortex M3 board. Because of the small values at hand, the overhead of HEAP TRACKER due to additional data structures and further processing becomes more visible. During the restore routine, the latter processing might be non-trivial, as we need to carefully reconstruct the layout of the heap using the information in the support data structure. For the same reason, the measures come close to the granularity of our equipment, hence higher variability is observed. COPY-IF-CHANGE restore performance, not shown in Figure 9, is constant and worse than any other mode. This is because COPY-IF-CHANGE does not leverage any information about what blocks need to be restored, thus it always re-writes the whole RAM.

We draw similar conclusions also for the Cortex M0 board. The experiments leading to Figure 7 and 9, however, do not exercise the ability of HEAP TRACKER to avoid writing some memory blocks in case of heap fragmentation. We investigate this setting next.

5.2 Non-contiguous Data

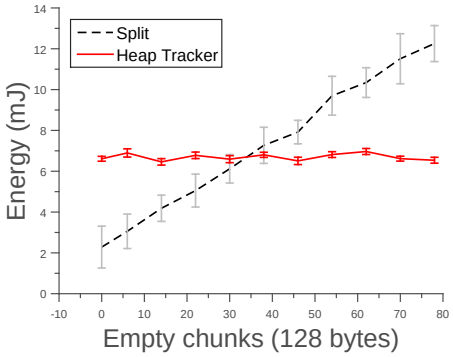
To investigate the other extreme compared to the case above, we concentrate on how SPLIT and HEAP TRACKER handle the case of non-contiguous data in the heap.

Setting. We employ a configuration with a fixed memory *occupation* and a varying *span*. This models the case where data is continuously allocated and deallocated in ways that prevent the program to use previous areas of the memory. To this end, we artificially create a situation with two 128-byte chunks of dynamically allocated memory separated by a variable number of unallocated chunks of the same size. All other memory segments, including the stack, are kept to the minimum. We only consider SPLIT and HEAP TRACKER because this setting bears no influence on FULL and STACK. COPY-IF-CHANGE, on the other hand, would show almost constant performance, as it would update at most two blocks.

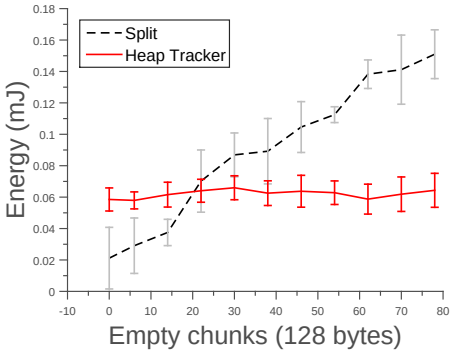
Results. Figure 10 shows the energy and time performance for the Cortex M3 board as the *span* increases with a growing number of unallocated chunks.

As for energy consumption during checkpoint, shown in Figure 10(a), the performance of HEAP TRACKER is about constant: by tracking what blocks cover chunks of allocated memory, the net amount of bytes written to stable storage remains the same regardless of the unallocated chunks. SPLIT, however, is unable to recognize the situation. It copies an increasingly higher amount of data to the flash chip as the *span* increases, because the highest memory address allocated on the heap continues to grow.

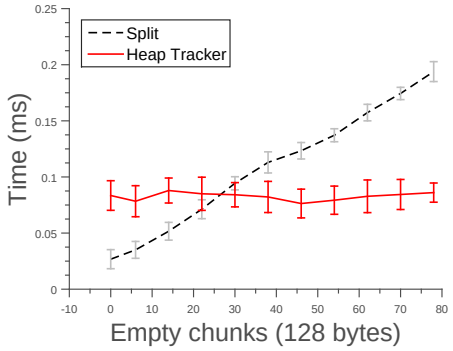
Nevertheless, the constant performance of HEAP TRACKER is worse than SPLIT as long as the size of the unallocated memory chunks collectively equals the size of the support data structure used by HEAP TRACKER to map out



(a) Average energy consumption during checkpoint.



(b) Average energy consumption during restore.

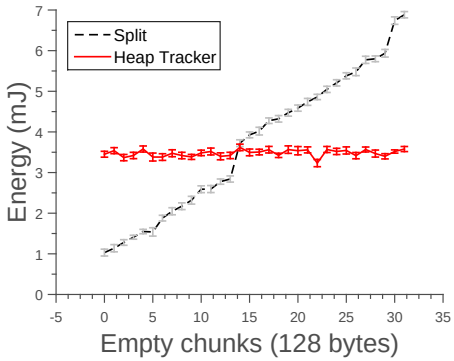


(c) Average time taken during restore.

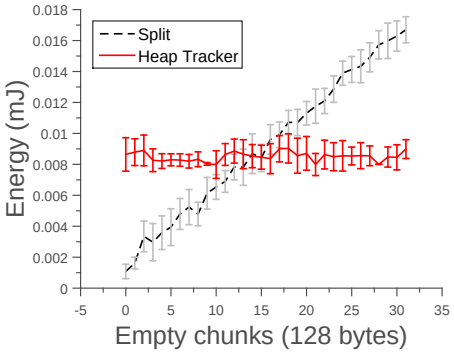
Figure 10. Cortex M3: performance in the case of non-contiguous data in the heap. HEAP TRACKER *starts paying off in energy and time as soon as the number of memory blocks it can avoid writing to flash equals the size of the support data structure used to track the heap.*

the heap. This occurs around 35 unallocated chunks; as soon as this grows larger, HEAP TRACKER shows overall better performance than SPLIT. In other words, HEAP TRACKER starts to pay off whenever the number of blocks it can avoid writing to flash counter-balances the overhead due to the support data structure. Similar considerations also apply to time, whose plot we omit for brevity.

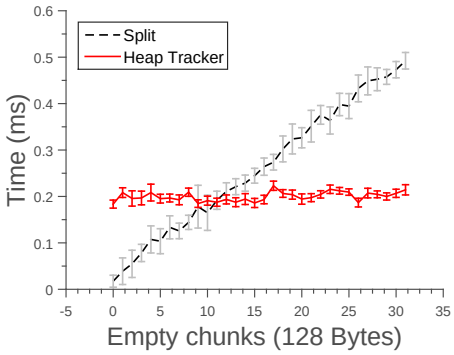
Even though the trends remain similar, the break-even point occurs earlier for the restore routine: around 20 unallocated chunks, as shown in Figure 10(b). This is an effect of the cheaper cost, in terms of energy consumption, of read



(a) Average energy consumption during checkpoint.



(b) Average energy consumption during restore.



(c) Average time taken during restore.

Figure 11. Cortex M0: performance in the case of non-contiguous data in the heap. *Even though processing using a Cortex M0 is cheaper energy-wise, but also slower compared to a Cortex M3, the performance is mainly determined by the flash chip.*

operations from flash compared to writes. In this respect, the Cortex M3 imposes a further cost: as shown in Figure 10(c), the break-even point for time no longer corresponds to the one for energy during restore. The added overhead is imputable to the processing required to reconstruct the heap based on information in the support data structure, which in the case of the Cortex M3 becomes appreciable.

Different, and sometimes opposite consideration apply to the results obtained from the Cortex M0 board, because of the different combination of MCU and flash chip, as visible in Figure 11. The flash chip on the Cortex M0 board requires erasing an entire 2 KByte segment before a new write can occur on the same segment. Figure 11(a) indeed indicates two steep increases in energy consumption between 14-15 and 30-31 chunks, corresponding to when a whole flash segment needs to be erased before performing a write.

Moreover, the flash chip largely determines the restore performance. Processing on a Cortex M0 is cheaper energy-wise than on a Cortex M3, but also slower. Despite this, comparing Figure 11(b) with Figure 11(c) indicates that the break-even point between SPLIT and HEAP TRACKER occurs earlier when considering time as opposed to energy, unlike what we observe in Figure 10(b) and 10(c) for the Cortex M3. Thus, reconstructing the heap using a Cortex M0 does not impose a significant time overhead due to processing, even though the MCU is slower than a Cortex M3. The overall performance is determined by the flash chip.

The results above all consider cases of 0% fragmentation. We study next the case of varying degrees of fragmentation.

5.3 Fragmented Data

We aim at realistically creating different levels of fragmentation in the heap to study how SPLIT, HEAP TRACKER, and COPY-IF-CHANGE handle the situation.

Setting. We borrow the definition of data structures from the CTP [5] protocol, from the custom design logging, and from a link estimator table [6] to emulate a scenario where differently-sized data items are continuously allocated and deallocated in the heap. Figure 12 exemplifies the first few iterations in these experiments. Memory *occupation* remains constant, as the sum of allocated memory chunks is always the same. Both the memory *span* and *fragmentation* change at every iteration. The former grows monotonically, whereas the latter yields seemingly casual values.

Such a setting replicates—on a smaller scale—the evolution of heap memory when using general purpose libraries of commonly used data structures. The only difference compared to reality is that we force the *span* to continue grow to sweep this parameter as well, whereas normally the heap manager would eventually start re-using previously deallocated memory chunks. All other memory segments, including the stack, are kept to the minimum possible.

Results. Figure 13 plots the results of energy consumption for the checkpoint routine, against a varying memory *span*. Similar overall trends are observed also for time.

The results for COPY-IF-CHANGE on the Cortex M3 board are shown separately in Figure 13(a) for better clarity. The performance is highly oscillating as it depends on how the changes in RAM align with the blocks on the flash chip. Initially, the trend is increasing because the allocated chunks are still close to each other in the first few iterations, and so they "move" within the same block that constantly needs to be dumped on flash. As the allocated blocks spread out, situations where an entire block is not impacted by changes increasingly occur, eventually determining oscillations within a specific interval. Similar trends are also seen for the Cortex M0 board, yet the 2 KByte granularity of page erases changes the scale of the oscillations.

Figure 13(b) plots the energy consumption for SPLIT and HEAP TRACKER on the Cortex M3 board. The absolute

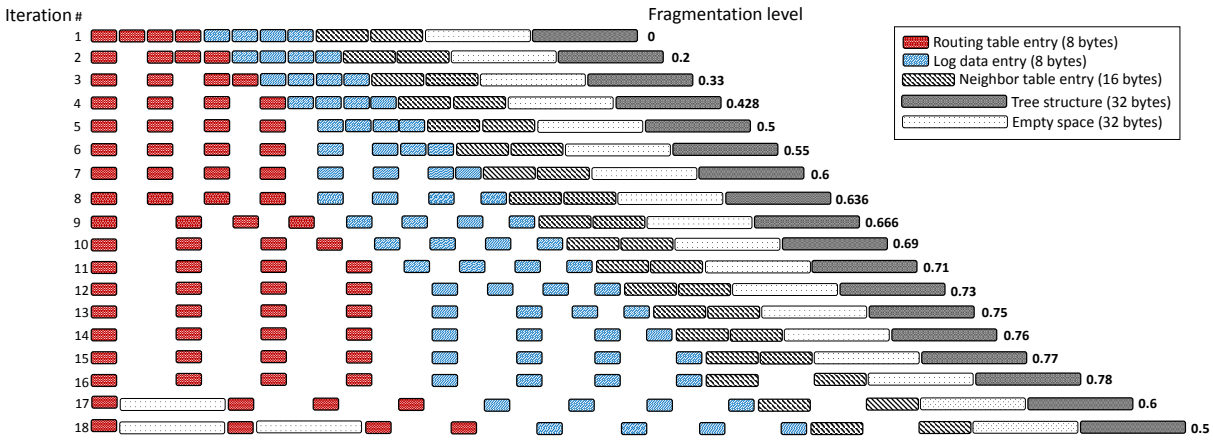


Figure 12. Evolution of the memory configurations as both *span* and *fragmentation* continue to change.

numbers indicate that both outperform COPY-IF-CHANGE within the 8 KByte maximum *span* we test, which we maintain to be a reasonable limit considering the intended use of the heap on this class of MCUs [9, 18]. Comparing SPLIT with HEAP TRACKER, as discussed in Section 5.1, the latter provides benefits as soon as the number of blocks it can skip writing on flash counterbalances the added overhead due to storing information to track the heap. Due to fragmentation in these experiments, HEAP TRACKER skips some blocks as soon as the size of unallocated memory chunks grows larger than S_h. Because of this, the break-even point with SPLIT occurs around a *span* of 5 KBytes. Different than Section 5.1, however, the performance of HEAP TRACKER is not always constant: the initial increase in energy consumption is due to the same reason as in Figure 13(a).

The results for the Cortex M0 board, shown in Figure 13(c), show similar trends as the corresponding results for the Cortex M3 board in Figure 13(b). Again, the different combination of MCU and flash chip makes the break-even point between SPLIT and HEAP TRACKER occur earlier, whereas the steep increase in memory consumption around 4 KBytes of memory *span* is again due to the page erase mode on the specific flash chip.

Figure 14 shows the results of these same experiments from the perspective of the *fragmentation* level rather than the memory *span*. We only consider SPLIT and HEAP TRACKER here, as COPY-IF-CHANGE is not directly affected by this dimension. The key observation based on comparing Figure 14(a) with 14(b) is that for low levels of fragmentation, HEAP TRACKER often outperforms SPLIT, whereas the opposite always holds for high levels of fragmentation. The explanation is that, also based on Figure 12, with constant *occupation* low levels of fragmentation are more likely to manifest as the *span* increases. Whenever this happens, however, we already observed that SPLIT incurs in high costs as it is unable to optimize the writes to flash based on unallocated memory chunks. The same applies to the Cortex M0 board, despite the different hardware.

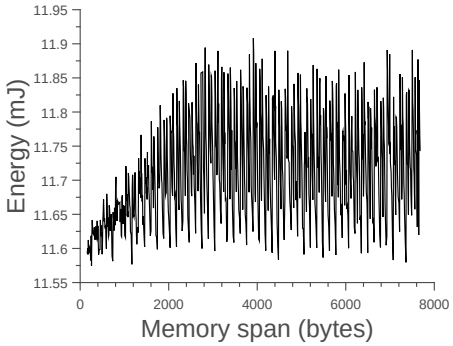
6 Discussion

The results we collect crucially indicate that no single storage mode is efficient in all situations. We discuss next these insights and attempt at identifying the application's characteristics that determine the recommended mode. Our conclusions are summarized in Table 1.

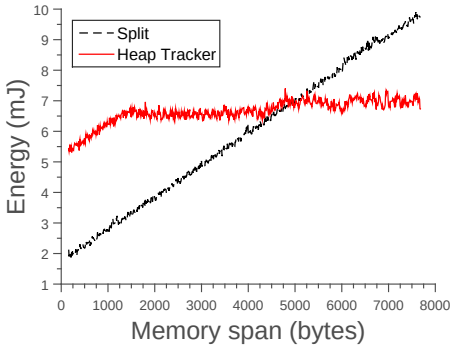
The role of memory *span*. Section 5.1 indicates that, independent of the *span*, using COPY-IF-CHANGE the overhead for reading a block from the previous checkpoint to understand whether an update is needed is quite limited. This is due to the characteristics of flash chips, where read operations are more energy-efficient than writes. Differently, the performance of all other storage modes drastically grows as the *span* increases, regardless of the need to update the previous checkpoint. As an example, comparing Figure 7 with 8 indicates that updating 25.6 KBytes of memory, that is, about one third of the entire RAM space on our Cortex M3 board, using COPY-IF-CHANGE roughly costs the same energy as using SPLIT with an *overall span* of 32 KBytes.

This observation makes COPY-IF-CHANGE attractive for applications characterized by a large memory *span* and a limited number of updates between consecutive checkpoints. This is the case, for example, of applications possibly required to run in a disconnected fashion. Under these circumstances, a node accumulates data until some form of opportunistic connection is established and data is offloaded. The data is often appended at the end of buffers while performing few changes on other data structures [26]: a pattern particularly suited to the way COPY-IF-CHANGE operates. Based on the same considerations, running SPLIT or HEAP TRACKER where COPY-IF-CHANGE is preferred would likely be inefficient. The former save all data regardless of changes and their performance worsen as the span grows, as already shown in Figure 7.

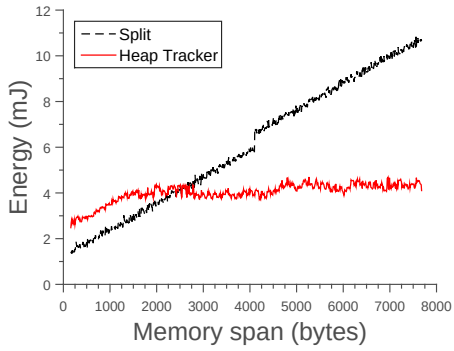
Opposite considerations apply to applications characterized by small memory *span* and frequent updates to data structures. This is the case, for example, of applications mostly concerned with routing packets on behalf of other nodes [15], where a small set of data structures is continuously updated as the wireless topology changes and protocols need to adapt. Most of our results indicate that, if the *span* is limited and regardless of *occupation* and *fragmentation*, SPLIT outperforms all other modes. This is, in essence, a result of its simple operation. Compared to COPY-IF-CHANGE, SPLIT does not pay the cost of initially reading all blocks from the previous checkpoint; compared to HEAP



(a) Cortex M3: COPY-IF-CHANGE.



(b) Cortex M3: SPLIT and HEAP TRACKER.

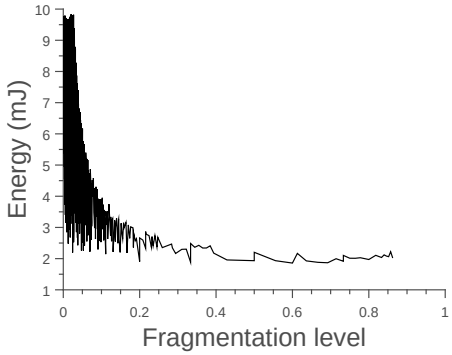


(c) Cortex M0: SPLIT and HEAP TRACKER.

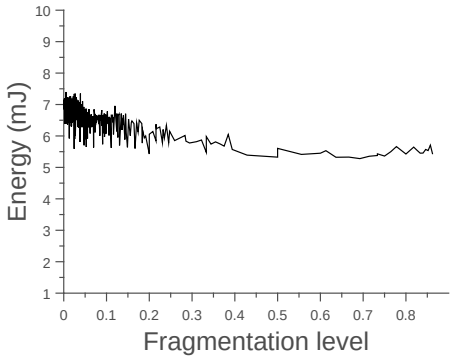
Figure 13. Energy consumption performance with varying *span* in case of fragmented memory, as shown in Figure 12. *For* COPY-IF-CHANGE*, the performance oscillates depending on how the changes in RAM align with the blocks on the flash chip.* SPLIT *and* HEAP TRACKER *provide different trade-offs depending on the memory* span.

TRACKER, it does not suffer from the overhead of support data structures. As an example, Figure 13 indicates that checkpointing around 3 KBytes of heap data using SPLIT costs 40% less energy than using HEAP TRACKER, and one third of that with COPY-IF-CHANGE.

The role of *fragmentation*. Section 5.2 and 5.3 point to a fundamental trade-off between SPLIT and HEAP TRACKER that concerns cases where the memory *span* grows without the *occupation* necessarily following. In these circumstances, HEAP TRACKER outperforms SPLIT provided the



(a) SPLIT.



(b) HEAP TRACKER.

Figure 14. Cortex M3: energy performance against different *fragmentation* levels. HEAP TRACKER *outperforms* SPLIT *for low levels of* fragmentation*, whereas the opposite holds for high levels of* fragmentation.

overhead of the support data structure equals the savings due to avoiding the write of some blocks on flash. The latter occurs when unallocated memory chunks are large enough to cover multiples of S_h, which we set to the smallest writable unit on the flash chip for better energy saving.

These memory configurations correspond to low levels of *fragmentation*. As an example, Figure 14 often demonstrates better performance for HEAP TRACKER up to 0.1 fragmentation. As a result, whenever the memory *span* is not too limited, and yet it is still not comparable to the entire RAM space, the choice of SPLIT or HEAP TRACKER ultimately depends on the expected levels of fragmentation. If the size of data structures in an application's implementation is close

Table 1. Summary of insights from the experimental results and mapping to example target applications.

Span	Fragmentation	Recommended mode	Example target
Large	-	COPY-IF-CHANGE	Disconnected operation
Small	-	SPLIT	Networking support
Intermediate	Low	HEAP TRACKER	Process control
	High	SPLIT	Remote sensing

to S_h, low levels of *fragmentation* are likely and thus HEAP TRACKER is recommended. This may be the case of control applications, where the representation of the process state is typically rendered with complex data structures [30]. Differently, if high levels of *fragmentation* are expected, SPLIT is to be favored. This would be the case of Internet-connected sensing [2], where small data items are acquired for the time necessary to perform some simple processing before sending the data out towards a long-distance destination.

7 Outlook and Conclusion

Our work here is foundational: we aim at developing the basic building blocks for state retention. In doing so, the contribution we present certainly has limitations. For example, we use synthetic settings rather than deploying real applications for evaluating the performance. Our methodology creates a controlled environment that offers repeatability and allows us to uniformly sweep the parameter space, at the cost of reduced realism. In contrast, concrete applications would introduce several sources of randomness, such as the unpredictable evolution of application state and the unequal harvesting performance across devices.

Necessary to enable an assessment in real applications is also to decide on the location of `checkpoint()` calls in the code. In principle, they should be placed at a point where the application makes a progress worth to be saved and the remaining energy is sufficient to complete the checkpoint. How to generalize such a notion is both challenging and orthogonal to increasing the efficiency of the individual checkpoint and restore operations, which is the goal we set forth in this work. We are currently investigating this problem with the goal of automatically deciding on the placement of `checkpoint()` calls, for example, based on control flow graph information, as opposed to manual placement of checkpoints by programmer [25].

In conclusion, we presented techniques to checkpoint and restore a device's state on stable storage, catering for scenarios where devices opportunistically harvest energy from the ambient or are provided with wireless energy transfer mechanisms. Our work aims at reducing the time for these operations and at minimizing their energy cost. We target modern 32-bit MCUs and currently available flash chips, making the checkpoint and restore routines available to programmers through a pair of simple C functions. The three storage modes we designed in support expose different trade-offs that depend on the memory *span*, its *occupation*, the possible *fragmentation*, and the read/write patterns in memory. The experimental results we gathered allowed us to quantity these trade-offs and discern the application's characteristics that would make one storage mode preferable over another.

Acknowledgments. We thank Marco Sampietro for the insightful comments and support to the experimental evaluation. This work was partly supported by the Cluster 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.

8 References

[1] Honeywell Inc: Honeywell Process Solutions—White Paper. tinyurl.com/honeywell-whitepaper.
[2] mbed. tinyurl.com/pkgoy6d, 2015.
[3] STM32 32-bit ARM Cortex MCUs. tinyurl.com/STM32bitMCU, 2015.
[4] STM32 Cube. tinyurl.com/STM32CubeMX, 2015.
[5] TinyOS CTP tree routing . tinyurl.com/TreeRouting, 2015.
[6] TinyOS Link Estimator. tinyurl.com/LinkEstimator, 2015.
[7] Y. Agarwal et al. Duty-cycling Buildings Aggressively: The Next Frontier in HVAC Control. In *IPSN*, 2011.
[8] L. Almeida, P. Pedreiras, and J. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Trans. on Industrial Electronics*, 49(6), 2002.
[9] M. Barr and A. Massa. *Programming Embedded Systems*. O'Relly Media, 2006.
[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[11] N. A. Bhatti et al. Sensors with Lasers: Building a WSN Power Grid. In *IPSN*, 2014.
[12] M. Buettner et al. RFID Sensor Networks with the Intel WISP. In *SENSYS*, 2008.
[13] Y. Chen et al. Surviving sensor network software faults. In *SOSP*, 2009.
[14] D. Dondi et al. A WSN System Powered by Vibrations to Improve Safety of Machinery with Trailer. In *IEEE Sensors*, 2012.
[15] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *SENSYS*, 2009.
[16] A. Hande, T. Polk, W. Walker, and D. Bhatia. Indoor Solar Energy Harvesting for Sensor Network Router Nodes . *Microprocessors and Microsystems*, 31(6), 2007.
[17] IAR Systems. IAR Embedded Workbench Cortex M Edition. tinyurl.com/arm-m-workbench, 2015.
[18] IAR Systems. Mastering Stack and Heap for System Reliability. tinyurl.com/iar-stack-heap, 2015.
[19] H. Jayakumar, A. Raha, and V. Raghunathan. QuickRecall: A Low Overhead HW/SW Approach for Enabling Computations Across Power Cycles in Transiently-powered Computers. In *International Conference on Embedded Systems and VLSI Design*, 2014.
[20] S. Kahrobaee and M. C. Vuran. Vibration energy harvesting for wireless underground sensor networks. In *ICC*, 2013.
[21] R. Koo and S. Toueg. Checkpointing and Rollback-recovery for Distributed Systems. In *Proceedings of ACM Fall Joint Computer Conference*, 1986.
[22] E. Lee. Cyber Physical Systems: Design Challenges. In *IEEE ISORC*, 2008.
[23] K. Li, H. Luan, and C.-C. Shen. Qi-ferry: Energy-constrained Wireless Charging in Wireless Sensor Networks. In *WCNC*, 2012.
[24] P. Liu et al. eLighthouse: Enhance Solar Power Coverage in Renewable Sensor Networks. *IJDSN*, 2013.
[25] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *PLDI*, 2005.
[26] L. Mottola. Programming Storage-centric Sensor Networks with Squirrel. In *IPSN*, 2010.
[27] F. Österlind et al. Sensornet Checkpointing: Enabling Repeatability in Testbeds and Realism in Simulations. In *EWSN*, 2009.
[28] B. Ransford et al. MementOS: System Support for Long-running Computation on RFID-scale Devices. *SIGARCH*, 39, 2011.
[29] P. Zhang, D. Ganesan, and B. Lu. QuarkOS: Pushing the Operating Limits of Micro-powered Sensors. In *Proceedings of the USENIX Conference on Hot Topics in Operating Systems*, 2013.
[30] J. Zou et al. Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems. In *RTAS*, 2009.