Efficient Intermittent Computing with Differential Checkpointing

Saad Ahmed
LUMS, Pakistan
16030047@lums.edu.pk

Naveed Anwar Bhatti
RI.SE SICS Swedish
naveed.bhatti@ri.se

Muhammad Hamad Alizai
LUMS, Pakistan
hamad.alizai@lums.edu.pk

Junaid Haroon Siddiqui
LUMS, Pakistan
junaid.siddiqui@lums.edu.pk

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

Abstract

Embedded devices running on ambient energy perform computations intermittently, depending upon energy availability. System support ensures forward progress of programs through state checkpointing in non-volatile memory. Checkpointing is, however, expensive in energy and adds to execution times. To reduce this overhead, we present DICE, a system design that efficiently achieves differential checkpointing in intermittent computing. Distinctive traits of DICE are its software-only nature and its ability to only operate in volatile main memory to determine differentials. DICE works with arbitrary programs using automatic code instrumentation, thus requiring no programmer intervention, and can be integrated with both reactive (Hibernus) or proactive (MementOS, HarvOS) checkpointing systems. By reducing the cost of checkpoints, performance markedly improves. For example, using DICE, Hibernus requires one order of magnitude shorter time to complete a fixed workload in real-world settings.

CCS Concepts • **Computer systems organization → Embedded software**.

Keywords transiently powered computers, intermittent computing, differential checkpointing

ACM Reference Format:
Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3316482.3326357

LCTES '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6724-0/19/06…$15.00
https://doi.org/10.1145/3316482.3326357

1 Introduction

Energy harvesting allows embedded devices to mitigate, if not to eliminate, their dependency on traditional batteries. However, energy harvesting is generally highly variable across space and time [7]. This trait clashes with the increasing push to realize tiny devices enabling pervasive deployments. Energy storage facilities, such as capacitors, are used to ameliorate fluctuations in energy supplies and need to be miniaturized as well, as they often represent a dominating factor in size. System shutdowns due to energy depletion are thus difficult to avoid. Computing then becomes *intermittent* [41, 46]: periods of normal computation and periods of energy harvesting come to be unpredictably interleaved [26].

Problem. System support exists to enable intermittent computing, employing a form of *checkpointing* to let the program cross periods of energy unavailability [5, 40]. This consists in replicating the application state over non-volatile memory (NVM) in anticipation of power failures, where it is retrieved back once the system resumes with sufficient energy.

Due to the characteristics of NVM, checkpoints are extremely costly in energy and time. When using flash memories, for example, the energy cost is orders of magnitude larger than most system operations [6, 34]. FRAM improves these figures; still, checkpoints often represent the dominating factor in an application's energy and time profile [5, 8]. As the cost of checkpoint is subtracted from the energy for useful computations, taming this overhead is crucial.

Contribution. To reduce the energy cost and additional execution times of checkpoints, we design DICE (DIfferential ChEckpointing), a system that limits the checkpoint operation to a slice of NVM data, namely, the differential between the previous checkpoint data and the volatile application state at the time of checkpointing.

To that end, as described in Sec. 3, we identify an efficient design point that integrates three contributions:

1) unlike previous attempts [1, 6] that access NVM to compute differentials, DICE maintains differential information *only in main memory* and access to NVM is limited to updating existing checkpoint data; we achieve this through an automatic code instrumentation step.
2) DICE capitalizes on the different memory write patterns by employing *different techniques* to track changes in long-lived global variables as opposed to short-lived variables local to functions or the heap; the code instrumentation step identifies these patterns and accordingly selects the most appropriate tracking technique.
3) in the absence of hardware support to track changes in main memory, which is too energy-hungry for intermittently-powered devices, our design is entirely implemented in software and ensures *functional correctness* by prudently opting for worst-case assumptions in tracking memory changes; we demonstrate, however, that such a choice is not detrimental to performance.

We design DICE as a plug-in complement to existing system support. This adds a further challenge. Systems such as Hibernus [4, 5] operate in a *reactive* manner: an interrupt is fired that may preempt the application at *any* point in time. Differently, systems such as MementOS [40] and HarvOS [8] place explicit function calls to *proactively* decide whether to checkpoint. Knowledge of where a checkpoint takes place influences what differentials need to be considered, how to track them, and how to configure the system parameters triggering a checkpoint. Sec. 4 details the code instrumentation of DICE, together with the different techniques we employ to support *reactive* and *proactive* systems.

Benefits. DICE reduces the amount of data to be written on NVM by orders of magnitude with Hibernus, and by a fraction of the original size with MementOS or HarvOS.

This bears beneficial cascading effects on a number of other key performance metrics. It reduces the peak energy demand during checkpoints and shifts the energy budget from checkpoints to useful computations. Reducing the peak demand enables a reduction of *up to one-eighth* in the size of energy buffer necessary for completing a given workload, cutting charging times and enabling smaller device footprints. This is crucial in application domains such as biomedical wearables [11] and implants [3]. Furthermore, DICE yields up to *one order of magnitude* fewer checkpoints to complete a workload. Sparing checkpoints lets the system progress farther on a single charge, cutting down the time to complete a workload up to *one order of magnitude*.

Following implementation details in Sec. 5, our quantitative assessment in this respect is two-pronged. Sec. 6 reports on the performance of DICE based on three benchmarks across three existing systems (i.e., Hibernus, MementOS, and HarvOS), two hardware platforms, and synthetic power profiles that allow fine-grained control on executions and accurate interpretation of results. Sec. 7 investigates the impact of DICE using power traces obtained from highly diverse harvesting sources. We show that the results hold across these different power traces, demonstrating the general applicability of DICE and its performance impact.

2 Background and Related Work

We elaborate on how intermittent computing shapes the problem we tackle in unseen ways; then proceed with discussing relevant works in this area.

2.1 Mainstream Computing

The performance trade-offs in mainstream computing are generally different compared to ours. Energy is not a concern, whereas execution speed is key, being it a function of stable storage operations or message exchanges on a network. Systems are thus optimized to perform as fast as possible, not to save energy by reducing NVM operations, as we do.

Differential checkpointing is widely used in tackling various challenges in the domain of virtual machines and multiprocessor operating systems. Here, checkpoints are mainly used for fault tolerance and load balancing. Systems use specialized hardware support to compute differentials [12, 36, 38], including memory management units (MMUs), which would be too energy-hungry for intermittently-powered devices. Moreover, updates happen at page granularity, say 4 KBytes. This is a tiny fraction of main memory in a mainstream computing system, but a large chunk of it in an intermittently-powered one, thus motivating different techniques.

Checkpointing in databases and distributed systems [21, 32, 39] is different in nature. Here, checkpoints are used to ensure consistency across data replicas and against concurrently-running transactions. Moreover, differential checkpointing does not require any tracking of changes in application state, neither in hardware nor in software, because the data to be checkpointed is explicitly provided by the application.

Differential checkpoints are also investigated in autonomic systems to create self-healing software. Enabling this behavior requires language facilities rarely available in embedded systems, let apart intermittently-powered ones. For example, Fuad et al. [13] rely on Java reflection, whereas Java is generally too heavyweight for intermittently-powered devices.

Our compile-time approach shares some of the design rationale with that of Netzer and Weaver [33], who however target debugging long-running programs, which is a different problem. Further, our techniques are thought to benefit from the properties of proactive checkpointing and to ensure correctness despite uncertainty in checkpoint times in reactive checkpointing. We apply distinct criteria to record differentials depending on different memory segments, including the ability of allowing cross-frame references along an arbitrary nesting of function calls.

2.2 Intermittent Computing

We can effectively divide the literature in three classes, depending on device architectures.

Non-volatile main memories. Device employing non-volatile main memories trade increased energy consumption and slower memory access for persistence [16]. When using FRAM as main memory with MSP430, for example, energy consumption increases by 2-3× and the device may only operate up to half of the maximum clock frequency [25].

The persistence brought by non-volatile main memory also creates data consistency issues due to repeated execution of non-idempotent operations that could lead to incorrect executions. Solutions exist that tackle this problem through specialized compilers [45] or dedicated programming abstractions [10, 26, 28]. The former may add up to 60% run-time overhead, whereas the latter require programmers to learn new language constructs, possibly slowing down adoption. An open research question is what are the conditions—for example, in terms of energy provisioning patterns—where the trade-off exposed by these platforms play favorably.

NVM for checkpoints. We target devices with volatile main memories and external NVM facilities for checkpoints [18, 24, 29, 35]. Existing literature in this area focuses on striking a trade-off between postponing the checkpoint as long as possible; for example, in the hope the environment provisions new energy, and anticipating the checkpoint to ensure sufficient energy is available to complete it.

Hibernus [5] and Hibernus++ [4] employ specialized hardware support to monitor the energy left. Whenever it falls below a threshold, both systems *react* by firing an interrupt that preempts the application and forces the system to take a checkpoint. Checkpoints may thus take place at *any* arbitrary point in time. Both systems copy the entire memory area—including unused or empty portions—onto NVM. We call this strategy *copy-all*.

MementOS [40] and HarvOS [8] employ compile-time strategies to insert specialized system calls to check the energy buffer. Checkpoints happen *proactively* and *only* whenever the execution reaches one of these calls. During a checkpoint, every *used* segment in main memory is copied to NVM regardless of changes since the last checkpoint. We call such a strategy *copy-used*.

Improving checkpoints. Unlike our approach of proactively tracking changes in application state, solutions exist that evaluate the differential at checkpoint time; either via hash comparisons [1] or by comparing main memory against a word-by-word sweep of the checkpoint data on NVM [6]. We call these approaches *copy-if-change*.

Note, however, that these systems are *fundamentally incompatible* with both the *reactive* and the *proactive* checkpointing systems that DICE aims to complement. The fundamental limitation is the inability to determine the energy cost of a checkpoint a priori, which is mandatory to decide *when* to trigger a checkpoint and is necessary input to all of the aforementioned systems. DICE provides an estimate of the *actual* cost of a checkpoint at any moment in execution, allowing to dynamically update system parameters triggering a checkpoint.

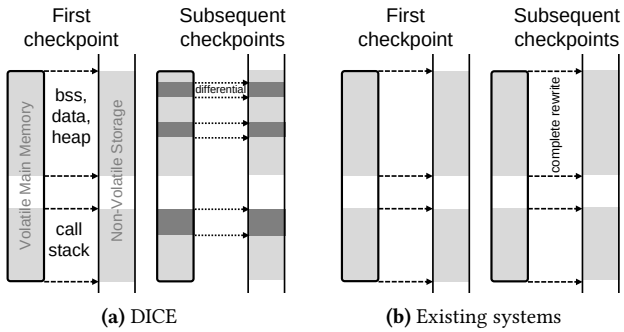


Figure 1. DICE fundamental operation. *DICE updates checkpoint data based on differentials at variable level in the global context, or with modified stack frames.*

Compared to *copy-if-change*, DICE also minimizes accesses to NVM. We achieve this through a specialized code instrumentation step. This inserts functionality to track changes in application state that *exclusively* operates in main memory. Operations on NVM are thus limited to updating the relevant blocks when checkpointing, with no additional preprocessing or bookkeeping required. The checkpoint data is then ready to be reloaded when computation resumes, with no further elaboration.

3 DICE In A Nutshell

Fig. 1 describes the fundamental operation of DICE. Once an initial checkpoint is available, DICE tracks changes in main memory to only update the affected slices of the existing checkpoint data, as shown in Fig. 1a. We detail such a process, which we call *recording differentials*, in Sec. 4.

Differentials. We apply different criteria to determine the granularity for recording differentials. The patterns of reads and writes, in fact, are typically distinct depending on the memory segment [22]. We individually record modifications in global context, including the BSS, DATA, and HEAP segments. Such a choice minimizes the size of the update for these segments on checkpoint data. Differently, we record modifications in the call stack at frame granularity. Local variables of a function are likely frequently updated during a function's execution. Their lifetime is also the same: they are allocated when creating the frame, and collectively lost once the function returns. Because of this, recording differentials at frame-level amortizes the overhead for variables whose differentials are likely recorded together.

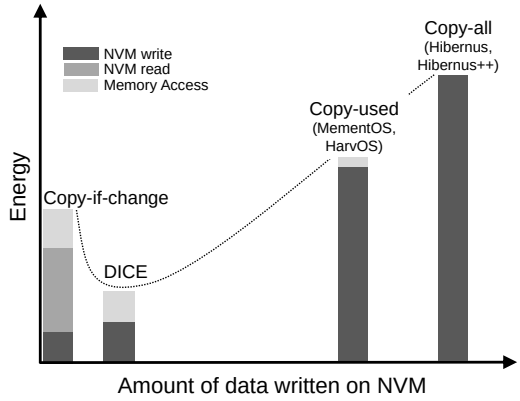


Figure 2. Qualitative comparison of the checkpoint techniques. Copy-all *has highest energy costs due to maximum writes on NVM.* Copy-used *avoids copying unused memory areas, reducing the energy cost.* Copy-if-change *further reduces energy costs, using NVM reads to compute differentials. DICE records changes in the stack at frame granularity, but only operates in main memory.*

A dedicated *precompiler* instruments the code to record both kinds of differentials. For global context, we insert DICE code to populate an *in-memory* data structure with information about modified memory areas. Writes to global variables can be statically identified, while indirect writes via pointers have to be dynamically determined. Therefore, we instrument direct writes to global variables and *all* indirect writes in the memory via pointer dereferencing. The precompiler also instruments the code to record differentials in the call stack by tracking the changes to the base pointer.

At the time of checkpointing, the in-memory data structures contain sufficient information to identify what slices of NVM data require an update. Unlike existing solutions [1, 6], this means that a checkpoint operation only accesses NVM to perform the actual updates to checkpoint data, whereas any other processing happens in main memory.

Our approach is sound but pessimistic, as we overestimate differentials. Our instrumentation is non-obtrusive as it only reads program state and records updates in a secluded memory region that will not be accessed by a well-behaved program. Similarly, an interrupted execution will be identical to an uninterrupted one because a superset of the differential is captured at the checkpoint and the entire program state is restored to resume execution. The differential is correctly captured because the grammar of the target language allows us to identify all direct or indirect (via pointers) memory writes. Any writes introduced by the compiler, such as register spilling, are placed on the current stack frame that is always captured, as described in Sec. 4.

DICE and the rest. Reducing NVM operations is the key to DICE performance. Fig. 2 qualitatively compares the energy performance of checkpointing solutions discussed thus far.

```
int var,*ptr; //global variables
...
record(&var,sizeof(var));
var++;
...
record(&ptr,sizeof(*ptr));
ptr = &var;
```

Figure 3. Example instrumented code.

Hibernus [5] and Hibernus++ [4] lie at the top right with their *copy-all* strategy. The amount of data written to NVM is maximum, as it corresponds to the entire memory space regardless of occupation. Both perform no read operations from NVM during checkpoint, and essentially no operation in main memory. MementOS [40] and HarvOS [8] write fewer data on NVM during checkpoint, as their *copy-used* strategy only copies the occupied portions. To that end, they need to keep track of a handful of information, such as stack pointers, adding minimal processing in main memory.

The *copy-if-change* [6] strategy lies at the other extreme. Because of the comparison between the current memory state and the last checkpoint data, the amount of data written to NVM is reduced. Performing such comparison, however, requires to sweep the entire checkpoint data on NVM, resulting in a high number of NVM reads. Because write operations on NVM tend to be more energy-hungry than reads [31], the overall energy overhead is still reduced.

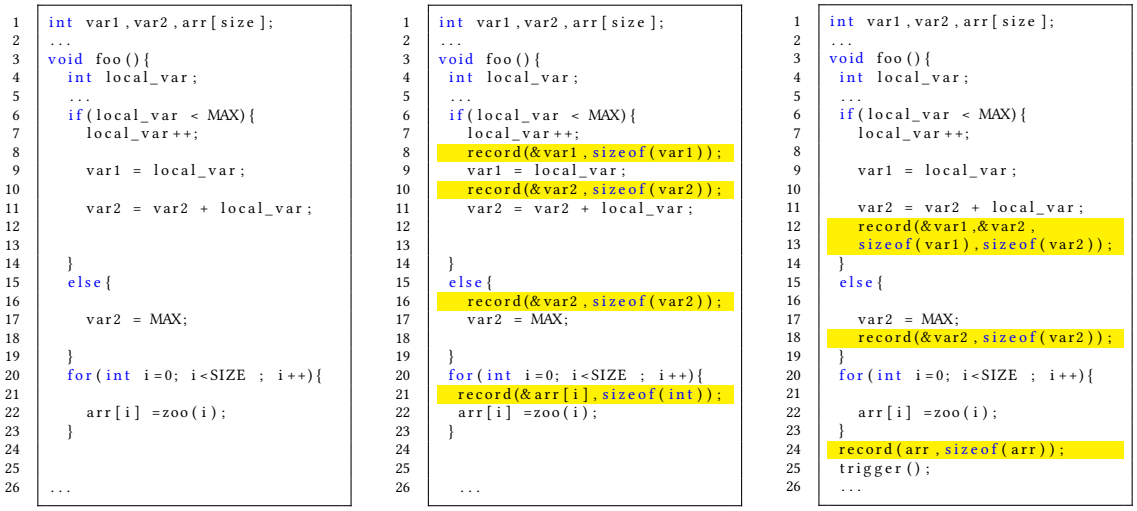
In contrast, DICE writes slightly more data to NVM compared to existing differential techniques, because modifications in the call stack are recorded at frame granularity. However, recording differentials only require operations in main memory and no NVM reads. As main memory is significantly more energy efficient than NVM, energy performance improves. Sec. 6 and Sec. 7 offer quantitative evidence.

4 Recording Differentials

We describe how we record differentials in global context, how we identify modified stack frames, and how we handle pointer dereferencing efficiently, while maintaining correctness. The description is based on a C-like language, as it is common for resource-constrained embedded platforms. Note our techniques work based on a well-specified grammar of the target language. We cannot instrument platform-specific inline assembly code, yet its use is extremely limited as it breaks cross-platform compatibility [14].

4.1 Global Context

DICE maintains a data structure in main memory, called *modification record*, to record differentials in global context. It is updated as a result of the execution of a **`record()`** primitive the DICE precompiler inserts when detecting a potential change to global context. The modification records are not part of checkpoint data.



(a) Before instrumentation (b) After instrumentation (reactive) (c) After instrumentation (proactive)

Figure 4. Example instrumentation for reactive or proactive checkpoints. *With reactive checkpoints, each statement possibly changing global context data is preceded by a call to* **`record()`**. *With proactive checkpoints, code locations where a checkpoint may take place are known, so calls to* **`record()`** *may be aggregated to reduce overhead.*

Fig. 3 shows an example. The **`record()`** primitive simply takes as input a memory address and the number of bytes allocated to the corresponding data type. This information is sufficient to understand that the corresponding slice of the checkpoint data is to be updated. How to inline the call to **`record()`** depends on the underlying system support.

Reactive systems. In Hibernus [4, 5], an interrupt may preempt the execution at any time to take a checkpoint. This creates a potential issue with the placement of **`record()`**.

If the call to **`record()`** is placed right after the statement modifying global context and the system triggers a checkpoint right after such a statement, but before executing **`record()`**, the modification record includes no information on the latest change. The remedy would be atomic execution of the statement changing data in global context and **`record()`**; for example, by disabling interrupts. With systems such as Hibernus [4, 5], however, this may delay or miss the execution of critical functionality.

Because of this, we choose to place calls to **`record()`** right before the relevant program statements, as shown in Fig. 4(b). This ensures that the modification record is pessimistically updated before the actual change in global context. If a checkpoint happens right after **`record()`**, the modification record might tag a variable as updated when it was not. This causes an unnecessary update of checkpoint data, but ensures correctness.

If a checkpoint happens right after **`record()`**, however, the following statement is executed first when resuming from checkpointed state. The corresponding changes are not tracked in the next checkpoint, as **`record()`** already executed before. We handle this by re-including in the next checkpoint the memory region reported in the most recent **`record()`** call. We prefer this minor additional overhead for these corner cases, rather than atomic executions.

Proactive systems. MementOS [40] and HarvOS [8] insert systems calls called **`trigger`**s in the code. Based on the state of the energy buffer, the triggers decide whether to checkpoint before continuing. This approach exposes the code to further optimizations.

As an example, Fig. 4(c) shows the same code as Fig. 4(a) instrumented for a proactive system. For segments without loops, we may aggregate updates to the modification record at the *basic block* level or just before the call to **`trigger()`**, whichever comes first[1]. The former is shown in line 8 to 14, where however we cannot postpone the call to **`record()`** any further, as branching statements determine only at runtime what basic block is executed.

In the case of loops over contiguous memory areas, further optimizations are possible. Consider lines 20 to 23 in Fig. 4: a call to **`record()`** inside the loop body, necessary in Fig. 4(b) for every iteration of the loop, may now be replaced with a single call before the call to **`trigger()`**. This allows DICE to record modifications in the whole data structure at once, as shown in Fig. 4(c) line 24.

Certain peculiarities of this technique warrant careful consideration. For instance, loops may, in turn, contain branching statements. This may lead to false positives in the modification record, which would result in an overestimation of differentials. Fine-grained optimizations may be possible in these cases, which however would require to increase the

[1]The aggregation of updates does not apply to pointers: if a single pointer modifies multiple memory locations, only the last one will be recorded.

complexity of instrumentation and/or to ask for programmer intervention. We opt for a conservative approach: we record modifications on the entire memory area that is *possibly*, but not definitely modified inside the loop.

4.2 Call Stack

Unlike data in global context, we record differentials of variables local to a function at frame level, as these variables are often modified together and their lifetime is the same. To this end, DICE monitors the growth and shrinking of the stack without relying on architecture support as in Clank [17].

Normally, *base pointer* (BP) points to the base of the frame of the currently executing function, whereas the *stack pointer* (SP) points to the top of the stack. DICE only requires an additional pointer, called the *stack tracker* (ST), used to track changes in BP between checkpoints. We proceed according to the following four rules:

R1: ST is initialized to BP every time the system resumes from the last checkpoint, or at startup;
R2: ST is unchanged as long as the current or additional functions are executed, that is, ST does *not* follow BP as the stack grows;
R3: whenever a function returns that possibly causes BP to point deeper in the stack than ST, we set ST equal to BP, that is, ST follows BP as the stack shrinks;
R4: at the time of checkpoint, we save the memory region between ST and SP, as this corresponds to the frames possibly changed since the last checkpoint.

Fig. 5 depicts an example. Say the system is starting with an empty stack. Therefore, ST, SP, and BP point to the base of the stack as per **R1**. Three nested function calls are executed. While executing **`F3`**, BP points to the base of the corresponding frame. Say a checkpoint happens at this time, as shown under checkpoint #1 in Fig. 5: the memory region between ST and SP is considered as a differential since the initial situation, due to **R4**.

When resuming from checkpoint #1, ST is equal to BP because of **R1**. Function **`F3`** continues its execution; no new functions are called and no functions return. According to **R2**, ST and BP remain unaltered. The next checkpoint happens at this time. As shown for checkpoint #2 in Fig. 5, **R4** indicates that the memory region to consider as a differential for updating the checkpoint corresponds to the frame of function **`F3`**. In fact, the execution of **`F3`** might still alter local variables, requiring an update of checkpoint data.

When the system resumes from checkpoint #2, **`F3`** returns. Because of **R3**, ST is updated to point to the base of the stack frame of **`F2`**. If a checkpoint happens at this time, as shown under checkpoint #3 in Fig. 5, **R4** indicates the stack frame of function **`F2`** to be the differential to update. This is necessary, as local variables in **`F2`** might have changed once **`F3`** returns control to **`F2`** and the execution proceeds within **`F2`**.

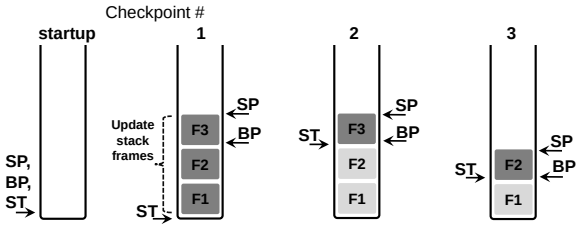


Figure 5. Identifying possibly modified stack frames. *The stack tracker (ST) is reset to the base pointer (BP) when the system resumes or at startup. ST does not follow BP as the stack grows, but it does so as the stack shrinks. The dark grey region between ST and the stack pointer (SP) is possibly modified.*

```
void foo(int a){
     int var = a,*ptr; //local variables
     ...
     ptr = &var;
     record_p(ptr,sizeof(*ptr));
     *ptr= a+5; //current stack frame
     ...        //modification
     bar(&var);
     ...
}
void bar(int *lptr){
     int a = 5; //local variables
     ...
     record_p(lptr,sizeof(*lptr));
     *lptr = a + 5; //previous stack frame
     ...            //modification
}
```

Figure 6. Example instrumented code to record local variables that are passed by reference.

Note that the efficiency of recording differentials at frame level also depends on programming style. If function calls are often nested, the benefits brought by this technique likely amplify compared to tracking individual local variables.

4.3 Pointer Dereferencing

Special care is required when tracking changes in main memory through dereferencing pointers. We use a separate **`record_p()`** primitive to handle this case.

With **`record_p()`**, we check if the pointer is currently accessing the global context or a local variable inside a stack frame. In the former case, the modification record is updated as described in Sec. 4.1. Otherwise, there are two possibilities depending on whether the memory address pointed to lies between ST and SP. If so, the corresponding change is already considered as part of the checkpoint updates, as per **R4** above. Otherwise, we find ourselves in a case like Fig. 6 and update ST to include the frame being accessed. As a result, we include the memory changes in the update to existing checkpoint data at the next checkpoint, as per **R4** above. This ensures correctness of our approach even if local variables are passed by reference along an arbitrary nesting of function calls or when using recursion.

5 Implementation

We describe implementation highlights for DICE, which are instrumental to understand our performance results.

Precompiler. We use ANTLR [37] to implement the precompiler for the C language. The precompiler instruments the source code for recording modifications in the global context as described in Sec. 4.1, depending on the underlying system support, for identifying modified regions of the stack, as explained in Sec. 4.2, and to handle pointer dereferencing as illustrated in Sec. 4.3.

As a result of code instrumentation, DICE captures changes in main memory except for those caused by peripherals through direct memory accesses (DMA). In embedded platforms, DMA buffers are typically allocated by the application or by the OS, so we know where they are located in main memory. We may simply flag them as modified as soon as the corresponding peripheral interrupts fire, independent of their processing.

DICE runtime. We implement `record()` as a variable argument function. In the case of proactive systems, this allows us to aggregate multiple changes in main memory, as shown in Fig. 4(c). We employ a simple bit-array to store the modification record, where each bit represents one byte in main memory as modified or not. This representation is compact, causing little overhead in main memory—12.5% in the worst case, corresponding to one bit in the modification record per byte of main memory.

Our choice allows `record()` to run in constant time, as it supports direct access to arbitrary elements. This prevents `record()` from changing the application timings, which may cause issues on resource-constrained embedded platforms [47]. Such a data structure, however, causes no overhead on NVM, as it does not need to be part of the checkpoint. Every time the system resumes from the previous checkpoint, we start afresh with an empty set of modification records.

We customize the existing checkpoint procedures with DICE-specific ones. Hibernus [5] and Hibernus++ [4] set the voltage threshold for triggering a checkpoint to match the energy cost for writing the entire main memory on NVM, as they use a *copy-all* strategy. HarvOS [8] bases the same decision on a worst-case estimate of the energy cost for checkpointing at specific code locations, as a function of stack size. When using DICE, due to its ability to limit checkpoints to a slice of the application state, both approaches are overly pessimistic. We set these parameters based on the *actual* cost of checkpointing. We obtain this by looking at how many modification records we accumulate and the positions of *ST* and *SP* at a given point in the execution.

MementOS [40] sets the threshold for triggering a checkpoint based on repeated emulation experiments using progressively decreasing voltage values and example energy traces, until the system cannot complete the workload. This processing requires no changes when using DICE; simply, when using DICE, the same emulation experiments generally return a threshold smaller than in the original MementOS, as the energy cost of checkpoints is smaller.

6 Evaluation: Synthetic Power Profiles

We dissect the performance of DICE using a combination of three benchmarks across three system support and two hardware platforms. Based on 107,000+ data points and compared with existing solutions on the same workload, our results indicate that using DICE reflects into:

- up to one-eighth smaller energy buffer to complete the same workload, cutting the time to reach the operating voltage and enabling smaller device footprints;
- up to 97% fewer checkpoints, which is a direct effect of DICE's ability to use a given energy budget for computing rather than checkpointing;
- up to one order of magnitude shorter completion time for a given workload, increasing system's responsiveness and despite the instrumentation overhead.

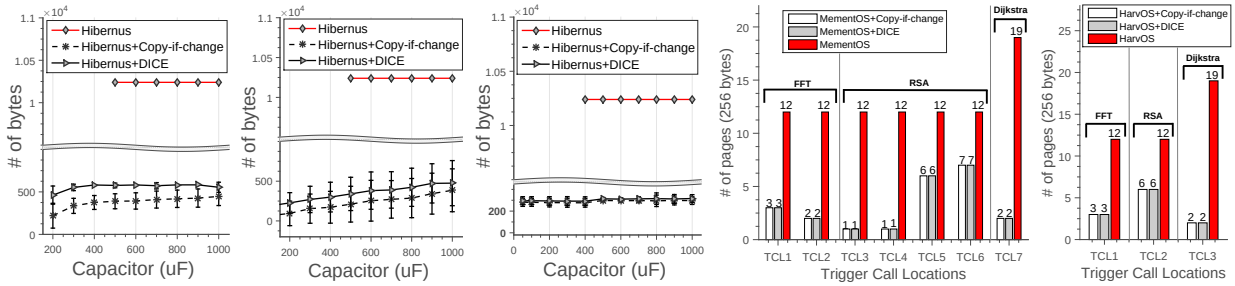
In the following, Sec. 6.1 describes the settings, whereas Sec. 6.2 to Sec. 6.4 discuss the results.

6.1 Settings

Benchmarks. We consider three benchmarks widely employed in intermittent computing [5, 20, 40, 45]: *i)* a Fast Fourier Transform (FFT) implementation, *ii)* RSA cryptography, and *iii)* Dijkstra spanning tree algorithm. FFT is representative of signal processing functionality in embedded sensing. RSA is an example of security support on modern embedded systems. Dijkstra's spanning tree algorithm is often found in embedded network stacks [19].

These benchmarks offer a variety of different data structures, memory access patterns, and processing load. The FFT implementation operates mainly over variables local to functions and has moderate processing requirements; RSA operates mostly on global data and demands great MCU resources; Dijkstra's algorithm only handles integer data types and exhibits much deeper levels of nesting. This diversity allows us to generalize our conclusions. All implementations are taken from public code repositories [30].

Systems and platforms. We measure the performance of DICE with both reactive (Hibernus) and proactive (MementOS, HarvOS) checkpoints. We consider as baselines the unmodified systems using either the *copy-all* or *copy-used* strategies. We also test the performance of *copy-if-change* [6] when integrated with either of these systems. Note that, in principle, such integration is not possible due to the inability of *copy-if-change* to predetermine checkpoint cost. For completeness, we still provide results assuming an *oracle* that provides this cost and enables integration. To make our analysis of MementOS independent of specific energy traces,



(a) Hibernus (FFT) **(b)** Hibernus (RSA) **(c)** Hibernus (Dijkstra) **(d)** MementOS **(e)** HarvOS

Figure 7. Update size. *The size of NVM updates is significantly smaller when using DICE compared to the original systems. Compared to* copy-if-change*, it remains the same or marginally larger.*

we manually sweep the possible parameter settings at steps of 0.2V and use the best performing one.

We run Hibernus on an MSP430-based TelosB interfaced with a byte-programmable 128 KByte FRAM chip, akin to the hardware originally used for Hibernus [5]. MementOS and HarvOS run on a Cortex M3-based ST Nucleo with a standard flash chip, already used to compare MementOS and HarvOS [8]. Both boards offer a range of hooks to trace the execution, enabling fine-grained measurements. Further, our choice of platforms ensures direct comparison with existing literature. In the same way as the original systems [4, 8, 40], our experiments focus on the MCU. Peripherals may operate through separate energy buffers [15] and dedicated solutions for checkpointing their states also exist [27, 43].

Metrics. We compute four metrics:

- The *update size* is the amount of data written to NVM during a checkpoint. This is the key metric that DICE seeks to reduce: measuring this figure is essential to understand the performance of DICE in all other metrics.
- The size of the *smallest energy buffer* is the minimum energy to complete a workload. If too small, a system may be unable to complete checkpoints, and ultimately make no progress. However, target devices typically employ capacitors: a smaller capacitor reaches the operating voltage sooner and enables smaller device footprints.
- The *number of checkpoints* is the total number of checkpoints to complete a workload. The more are necessary, the more the system subtracts energy from useful computations. Reducing NVM operations allows the system to use energy more for computations than checkpoints, allowing an application to progress further on the same charge.
- The *completion time* is the time to complete a workload, excluding deployment-dependent recharge times. DICE introduces a run-time overhead due to recording differentials. On the other hand, fewer NVM operations reduce both the time required for a single checkpoint and, because of the above, their number.

Power profile and measurements. We use a foundational power profile found in existing literature [6, 20, 40, 45] that provides fine-grained control over executions and facilitates interpreting results. The device boots with the full capacitor and computes until the capacitor is empty again. In the mean time, the environment provides no additional energy. Once the capacitor is empty, the environment provides new energy until the capacitor is full again and computation resumes.

This profile is also representative of a staple class of intermittently-powered applications, namely, those based on wireless energy transfer [9, 40]. With this technology, devices are quickly charged with a burst of wirelessly-transmitted energy until they boot. Next, the application runs until the capacitor is empty again. The device rests dormant until another burst of wireless energy comes in.

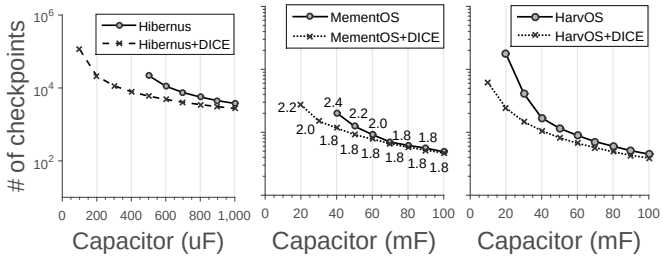
We trace the execution on real hardware using an attached oscilloscope along with the ST-Link debugger and μVision for the Nucleo board. This equipment allows us to ascertain the time taken and energy consumption of every operation during the execution, including checkpoints on FRAM or flash memory. The results are obtained from 1,000 (10,000) benchmark iterations on the MSP430 (Cortex M3) platform.

6.2 Results → Update Size

We compare the update size in DICE with the original designs of Hibernus, MementOS, and HarvOS as well as when these are combined with *copy-if-change*. We do this despite its inability to work together with these systems because of integration issues, as previously discussed.

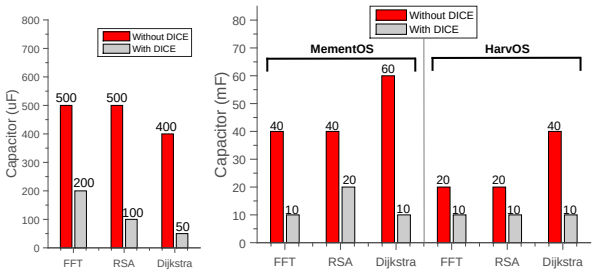
Fig. 7 shows the results. With Hibernus, the code location where a checkpoint takes place is unpredictable: depending on the capacitor size, an interrupt eventually fires prompting the system to checkpoint. Fig. 7a, 7b, and 7c[2] thus report the average update size during an experiment until completion

[2]Some data points are missing in the charts for the original design of Hibernus, as it is unable to complete the workload in those conditions. We investigate this aspect further in Sec. 6.3.



(a) Hibernus (RSA) **(b)** MementOS (RSA) **(c)** HarvOS (RSA)

Figure 8. Number of checkpoints necessary against varying capacitor sizes. *With smaller capacitors, highly-intermittent executions greatly benefit from DICE.*



(a) Hibernus **(b)** MementOS and HarvOS

Figure 9. Smallest capacitor. *A DICE-equipped system completes the workload with smaller capacitors. This is due to a reduction in the energy cost of checkpoints, enabled by the reduction in update size.*

of the workload, as a function of the capacitor size. Compared to the original *copy-all* strategy, DICE provides orders of magnitude improvements. These are a direct result of limiting updates to those determined by the instrumented code. Using *copy-if-change* with Hibernus provides marginal advantages over DICE, because modifications in the call stack are recorded at word-, rather than frame-granularity.

With MementOS and HarvOS, the update size is a function of the location of the trigger call as determined by the original design of either system, which DICE has no impact on. This is because the stack may have different sizes at different places in the code. Fig. 7d and Fig. 7e[3] show that DICE reduces the update size to a fraction of that in the original *copy-used* strategy, no matter the location of the trigger call. The same charts show that the performance of *copy-if-change* when combined with MementOS or HarvOS is the same as DICE. This is an effect of the page-level programmability of flash storage, requiring an entire page to be rewritten on NVM even if a small fraction of it requires an update.

The cost for *copy-if-change* to match or slightly improve the performance of DICE in update size is, however, prohibitive in terms of energy consumption. *Copy-if-change* indeed requires a complete sweep of the checkpoint data on NVM before updating, and even for FRAM, the cost of reads is comparable to writes [31]. As an example, we compute the energy cost of a checkpoint with the data in Fig. 7b to be 93% higher with *copy-if-change* than with DICE, on average. For Hibernus, *copy-if-change* would result in an energy efficiency worse than the original *copy-all* strategy. As energy efficiency is the figure users are ultimately interested in, we justifiably narrow down our focus to comparing a DICE-equipped system with the original ones.

6.3 Results → Checkpoints and Energy Buffers

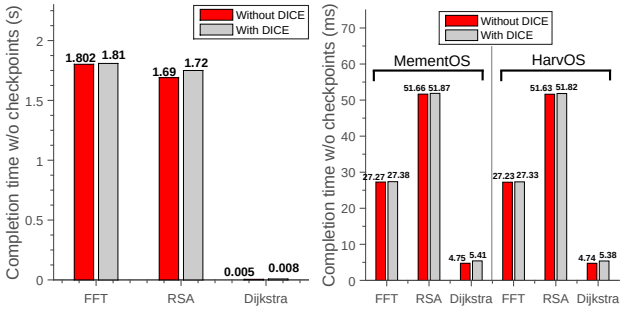
The results in the number of checkpoints and in the size of the smallest energy buffer are intimately intertwined. Fig. 8[4] shows an excerpt of the results we gather in the number of checkpoints against variable capacitor sizes, focusing on the most complex benchmark we test. The results are similar or better for other benchmarks. A significant area of these charts only shows the performance of the DICE-equipped systems, as the original ones are unable to complete the workload with too small capacitors. As soon as a comparison is possible, the improvements for DICE with small capacitors are significant and apply consistently across benchmarks.

Fig. 9 reports the minimum size of the capacitor required to complete the given workloads. A DICE-equipped system constantly succeeds with smaller capacitors. With Hibernus, DICE allows one to use a capacitor up to one-eighth of the one required with the original *copy-all* strategy. For MementOS and HarvOS, the smallest capacitor one may employ is about half the size of the one required in the original designs.

These results are directly enabled by the reduction in update size, discussed in Sec. 6.2. With fewer data to write on NVM, the energy cost of checkpoints reduces. This has two direct consequences. First, the smallest amount of energy the system needs to have available at once to complete the checkpoint reduces. Second, the system can invest the available energy to compute rather than checkpointing.

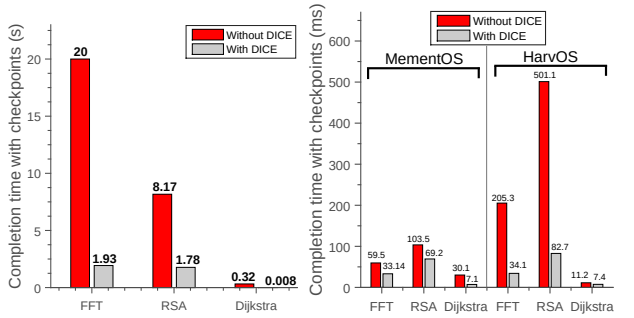
With larger capacitors, the improvements in Fig. 8 are smaller, but still appreciable[5]. This is expected: the larger is the capacitor, the more the application progresses farther on a single charge, thus executions are less intermittent and checkpoints are sparser in time. Modifications since

[3]For MementOS, the trigger call locations refer to the "function call" placement strategy in MementOS [40]. We find the performance with other MementOS strategies to be essentially the same. We omit that for brevity.
[4]For MementOS, we tag every data point with the minimum voltage threshold that allows the system to complete the workload, if at all possible, as it would be returned by the repeated emulation runs [40].
[5]Note the log scale on the Y axis of Fig. 8.



(a) Hibernus **(b)** MementOS and HarvOS

Figure 10. Completion time without concrete checkpoints. *Overhead due to code instrumentation is limited.*



(a) Hibernus **(b)** MementOS and HarvOS

Figure 11. Completion time including checkpoints. *The run-time overhead due to DICE is overturn by reducing size and number of checkpoints, yielding in shorter completion times.*

the previous checkpoint accumulate as a result of increased processing times. The state of main memory then becomes increasingly different than the checkpoint data, and eventually DICE updates a significant part of it.

This performance allows systems to reduce the time invested in checkpoint operations, because of a reduction in their number and in the time taken for single checkpoints due to fewer NVM operations. This reduces the time to complete the workload, as we investigate next.

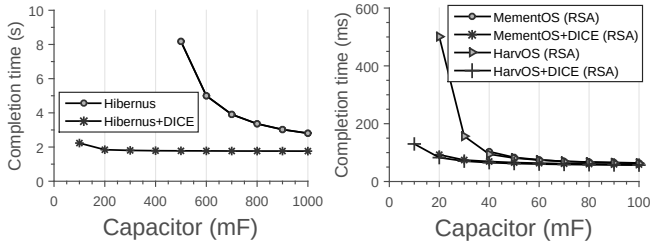
6.4 Results → Completion Time

DICE imposes a cost, mainly due to run-time overhead for recording differentials. On the other hand, based on the above results, DICE enables more rapid checkpoints as it reduces NVM operations. In turn, this allows the system to reduce their number as energy is spent more on completing the workload than checkpoints. These factors should conversely reduce the completion time.

Fig. 10 investigates this aspect in a single iteration of the benchmarks where the code executes normally, but we skip the actual checkpoint operations. This way, we observe the net run-time overhead due to `record[_p]()`. The overhead is limited. This is valid also for reactive checkpoints in Hibernus, despite the conservative approach at placing `record[_p]()` calls due to the lack of knowledge on where the execution is preempted.

Fig. 11 includes the time required for checkpoint operations with the smallest capacitor allowing both the DICE-equipped system and the original one to complete the workload, as per Fig. 9. The overhead due to `record[_p]()` is not only compensated, but actually overturn by fewer more rapid checkpoints. Using these configurations, DICE allows the system to complete the workload much earlier, increasing the system's responsiveness.

Fig. 12 provides an example of the trends in completion time against variable capacitor sizes. The improvements are



(a) Hibernus (RSA) **(b)** Mementos and HarvOS (RSA)

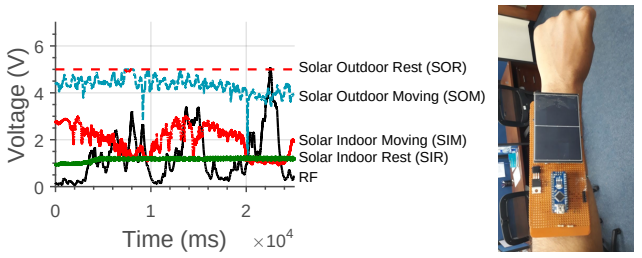
Figure 12. Completion time against capacitor size. *DICE-enabled gains are higher with more intermittent executions.*

significant in a highly-intermittent setting with smaller capacitors. Similar to Fig. 8, two factors contribute to the curves in Fig. 12 approaching each other. Larger capacitors allow the code to make more progress on a single charge, so the number of checkpoints reduces. As more processing happens between checkpoints, more modifications occur in application state, forcing DICE to update a larger portion of checkpoint data.

7 Evaluation: Variable Power Profiles

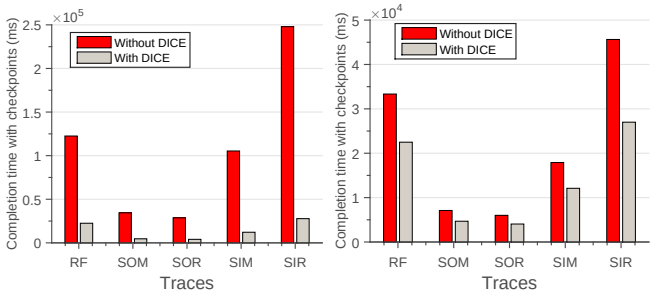
We investigate the impact of DICE using a variety of different power profiles. We build the same activity recognition (AR) application often seen in existing literature [10, 26, 28, 40], using the same source code [14]. We use an MSP430F2132 MCU, that is, the MCU used in the WISP platform [42] for running the same AR application. The rest of the setup is as for Hibernus in Sec. 6. We focus on completion time, as defined in Sec. 6.1, in a single application iteration. All of the performance metric we study eventually converge on completion time.

Power traces. We consider five power traces, obtained from diverse energy sources and in different settings.



(a) Voltage traces excerpt. **(b)** Device setup.

Figure 13. Example voltage traces and device setup.



(a) Completion time with 50uF. **(b)** Completion time with 100uF.

Figure 14. Performance of the AR app running on Hibernus with 50uF and 100uF capacitor. *Performance gains are observed across diverse power traces obtained from different energy sources.*

One of the traces is the RF trace from MementOS [23, 40], recorded using the WISP 4.1. The black curve in Fig. 13 shows an excerpt, plotting the instantaneous voltage reading at the energy harvester over time. We collect four additional traces using a mono-crystalline solar panel [44] and an Arduino Nano [2] to measure the voltage output across a 30kOhm load, roughly equivalent to the resistance of an MSP430F2132 in active mode. Using this setup, we experiment with different scenarios. We attach the device to the wrist of a student to simulate a fitness tracker. The student roams in the university campus for outdoor measurements (SOM), and in research lab for indoor measurements (SIM). Alternatively, we keep the device on the ground right outside the lab for outdoor measurements (SOR), and at desk level in our research lab for the indoor measurements (SIR). Fig. 13 visually demonstrates the extreme variability and considerable differences among the power traces we consider.

Results. Across all traces, we find that a 10uF capacitor is sufficient for a DICE-equipped Hibernus to complete a single iteration of the AR application. Without DICE, Hibernus needs a five times larger capacitor for the same workload.

Fig. 14a shows the performance with a 50uF capacitor, where both the original Hibernus and the DICE-equipped one can complete. Completion times diminish by at least one order of magnitude using DICE. This means better energy efficiency and increased reactivity to external events. Best performance is obtained with the outdoor solar power trace in a static setup, as expected in that it supplies the largest energy. However, DICE constantly improves over the original Hibernus, regardless of the power trace.

The trends we discuss in Sec. 6 with larger capacitors are confirmed here. Fig. 14b plots the performance using a 100uF capacitor. Improvements are reduced compared with Fig. 14b: applications progress farther on a single charge, checkpoints are sparser, and DICE records more changes to the application state between checkpoints.

8 Conclusion

DICE is a differential checkpointing solution for intermittent computing. To reduce the amount of data written on NVM, we conceive different ways to track changes in main memory, depending on the memory segments. These techniques are embedded within existing code in an automatic fashion using a specialized pre-processing step, and designed to only operate in main memory until checkpointing. This helps existing system support complete a given workload with i) smaller energy-buffers, ii) fewer checkpoints, and thus better energy efficiency, and iii) reduced completion time. Our benchmark evaluation, based on a combination of three benchmarks across three different systems and two different hardware platforms, provides quantitative evidence. The improvements are confirmed against variable power profiles: experiments with an AR application show orders of magnitude improvements against power traces as diverse as RF-based wireless energy transfer and solar radiation.

References

[1] Faycal Ait Aouda, Kevin Marquet, and Guillaume Salagnac. 2014. Incremental checkpointing of program state to NVRAM for transiently-powered systems. In *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip*.

[2] ARDUINO. 2018. *NANO*. https://store.arduino.cc/usa/arduino-nano (accessed 2018-02-28).

[3] Satu Arra, Jarkko Leskinen, Janne Heikkila, and Jukka Vanhala. 2007. Ultrasonic power and data link for wireless implantable applications. In *2nd International Symposium on Wireless Pervasive Computing*.

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

[5] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* (2015).

[6] Naveed Bhatti and Luca Mottola. 2016. Efficient state retention for transiently-powered embedded sensing. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks*.

[7] Naveed Anwar Bhatti, Muhammad Hamad Alizai, Affan A Syed, and Luca Mottola. 2016. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions*

on Sensor Networks (2016).

[8] 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*.

[9] Michael Buettner, Benjamin Greenstein, and David Wetherall. 2011. Dewdrop: An Energy-Aware Runtime for Computational RFID. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*.

[10] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[11] Canan Dagdeviren, Pauline Joe, Ozlem L Tuzman, Kwi-Il Park, Keon Jae Lee, Yan Shi, Yonggang Huang, and John A Rogers. 2016. Recent progress in flexible and stretchable piezoelectric devices for mechanical energy harvesting, sensing and actuation. *Extreme Mechanics Letters* (2016).

[12] Bernhard Egger, Younghyun Cho, Changyeon Jo, Eunbyung Park, and Jaejin Lee. 2016. Efficient Checkpointing of Live Virtual Machines. *IEEE Transactions on Computers* (2016).

[13] M. Muztaba Fuad and Michael J. Oudshoorn. 2007. Transformation of Existing Programs into Autonomic and Self-healing Entities. In *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems*.

[14] Abstract Research Group. 2018. *Benchmark Applications*. www.github.com/CMUAbstract/releases#benchmark-applications (accessed 2018-02-28).

[15] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*.

[16] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems*.

[17] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[18] Texas Instruments. 2013. *MSP430 Solar Energy Harvesting Development Tool*. http://www.ti.com/tool/EZ430-RF2500-SEH (accessed 2018-08-03).

[19] Oana Iova, Pietro Picco, Timofei Istomin, and Csaba Kiraly. 2016. RPL: The Routing Standard for the Internet of Things... Or Is It? *IEEE Communications Magazine* (2016).

[20] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. Quick Recall: A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems* (2015).

[21] Richard Koo and Sam Toueg. 1986. Checkpointing and Rollback-recovery for Distributed Systems. In *Proceedings of ACM Fall Joint Computer Conference*.

[22] P. Koopman. 2010. *Better Embedded System Software*. CMU Press.

[23] PERSIST Lab. 2018. *RF Trace*. https://github.com/PERSISTLab/BatterylessSim/tree/master/traces (accessed 2018-02-28).

[24] Libelium. 2017. *Waspmote*. http://www.libelium.com/products/waspmote/hardware/ (accessed 2018-08-06).

[25] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *Leibniz International Proceedings in Informatics*.

[26] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[27] Giedrius Lukosevicius, Alberto Rodriguez Arreola, and Alexander Weddell. 2017. Using Sleep States to Maximize the Active Time of Transient Computing Systems. In *Proceedings of the 5th ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*.

[28] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM on Programming Languages* (2017).

[29] Robert Margolies, Maria Gorlatova, John Sarik, Gerald Stanje, Jianxun Zhu, Paul Miller, Marcin Szczodrak, Baradwaj Vigraham, Luca Carloni, and Peter Kinget. 2015. Energy-harvesting active networked tags (EnHANTs): Prototyping and experimentation. *ACM Transactions on Sensor Networks* (2015).

[30] mbed. 2017. *IoT OS*. goo.gl/u918jX.

[31] Kresimir Mihic, Ajay Mani, Manjunath Rajashekhar, and Philip Levis. 2007. Mstore: Enabling storage-centric sensornet research. In *ACM/IEEE International Conference on Information Processing in Sensor Networks*.

[32] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture*.

[33] Robert HB Netzer and Mark H Weaver. 1994. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI*, Vol. 94. 313–325.

[34] Hoang Anh Nguyen, Anna Forster, Daniele Puccinelli, and Silvia Giordano. 2011. Sensor node lifetime: An experimental study. In *Pervasive Computing and Communications Workshops*.

[35] Expansion of STM32 Nucleo boards. 2017. *Data Sheet: X-NUCLEO-NFC02A1*. https://www.st.com/resource/en/data_brief/x-nucleo-nfc02a1.pdf (accessed 2018-08-06).

[36] Eunbyung Park, Bernhard Egger, and Jaejin Lee. 2011. Fast and Space-efficient Virtual Machine Checkpointing. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*.

[37] Terence Parr. 2013. *The Definitive ANTLR 4 Reference*. goo.gl/RR1s.

[38] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX Technical Conference*.

[39] Brian Randell, Pete Lee, and Philip C. Treleaven. 1978. Reliability issues in computing system design. *Comput. Surveys* (1978).

[40] Benjamin Ransford. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[41] B. Ransford. 2013. *Transiently Powered Computers*. Ph.D. Dissertation. School of Computer Science, UMass Amherst.

[42] Joshua R. Smith, Alanson P. Sample, Pauline S. Powledge, Sumit Roy, and Alexander Mamishev. 2006. A Wirelessly-powered Platform for Sensing and Computation. In *Proceedings of the 8th International Conference on Ubiquitous Computing*.

[43] Rebecca Smith and Scott Rixner. 2015. Surviving Peripheral Failures in Embedded Systems.. In *USENIX Annual Technical Conference*.

[44] IXYS SolarMD. 2018. *SLMD481H08L*. http://ixapps.ixys.com/ (accessed 2018-02-28).

[45] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*.

[46] Guang Yang, Bernard H Stark, Simon J Hollis, and Steve G Burrow. 2014. Challenges for Energy Harvesting Systems Under Intermittent Excitation. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2014).

[47] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. 2007. Clairvoyant: A Comprehensive Source-level Debugger for Wireless Sensor Networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*.