Fast and Energy-efficient State Checkpointing for Intermittent Computing

SAAD AHMED, Lahore University of Management Sciences (LUMS), Pakistan
NAVEED ANWAR BHATTI, Air University, Pakistan
MUHAMMAD HAMAD ALIZAI, Lahore University of Management Sciences (LUMS), Pakistan
JUNAID HAROON SIDDIQUI, Lahore University of Management Sciences (LUMS), Pakistan
LUCA MOTTOLA, Politecnico di Milano, Italy and RISE, Sweden

Intermittently-powered embedded devices ensure forward progress of programs through state checkpointing in non-volatile memory. Checkpointing is, however, expensive in energy and adds to the execution times. To minimize this overhead, we present DICE, a system that renders differential checkpointing profitable on these devices. DICE is unique because it is a software-only technique and efficient because it only operates in volatile main memory to evaluate the differential. DICE may be integrated with reactive (Hibernus) or proactive (MementOS, HarvOS) checkpointing systems, and arbitrary code can be enabled with DICE using automatic code-instrumentation requiring no additional programmer effort. By reducing the cost of checkpoints, DICE cuts the peak energy demand of these devices, allowing operation with energy buffers that are one-eighth of the size originally required, thus leading to benefits such as smaller device footprints and faster recharging to operational voltage level. The impact on final performance is striking: with DICE, Hibernus requires one order of magnitude fewer checkpoints and one order of magnitude shorter time to complete a workload in real-world settings.

CCS Concepts: • **Computer systems organization → Embedded systems**;

Additional Key Words and Phrases: transiently powered computers, intermittent computing, differential checkpointing

ACM Reference Format:
Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2020. Fast and Energy-efficient State Checkpointing for Intermittent Computing. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2020), 25 pages. https://doi.org/10.1145/3391903

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 [9]. 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

Authors' addresses: Saad Ahmed, saad.ahmed@lums.edu.pk, Lahore University of Management Sciences (LUMS), Lahore, Punjab, Pakistan, 54792; Naveed Anwar Bhatti, naveed.bhatti@mail.au.edu.pk, Air University, Islamabad, Pakistan; Muhammad Hamad Alizai, Lahore University of Management Sciences (LUMS), Lahore, Punjab, Pakistan, 54792, hamad.alizai@lums.edu.pk; Junaid Haroon Siddiqui, Lahore University of Management Sciences (LUMS), Lahore, Punjab, Pakistan, 54792, junaid.siddiqui@lums.edu.pk; Luca Mottola, Politecnico di Milano, Italy and RISE, Sweden, luca.mottola@polimi.it.

1539-9087/2020/1-ART1 $15.00
https://doi.org/10.1145/3391903

in size. System shutdowns due to energy depletion are thus difficult to avoid. Computing then becomes *intermittent* [52, 63]: periods of normal computation and periods of energy harvesting come to be unpredictably interleaved [37].

Problem. System support exists to enable intermittent computing, employing a form of *checkpointing* to let the program cross periods of energy unavailability [7, 51]. 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 [8, 45]. FRAM improves these figures; still, checkpoints often represent the dominating factor in an application's energy and time profile [7, 10]. As the cost of checkpoint is subtracted from the energy for useful computations, taming this overhead is crucial.

To that end, differential techniques are commonly employed when providing fault tolerance in operating systems [32] and when maintaining consistency in distributed databases and transactions [33, 49].In Sec. 3, we analytically scrutinize existing approaches. While some of these are simply not applicable in embedded systems due to lack of requisite hardware support, we observe that the energy cost of existing software-based approaches is often not worth the benefit on energy constrained platforms. Therefore, newer techniques must be established to reap the benefits of differential checkpointing in intermittently-powered systems.

Contribution. To cater for the specific challenges of intermittently-powered devices, we design DICE (DIfferential ChEckpointing), a system that efficiently evaluates the differential between the previous checkpoint data and the volatile application state. DICE uses this information to limit the checkpoint operation to a slice of NVM data, namely, the one corresponding to changed application state. The fundamental contribution of DICE rests in identifying an efficient design point that allows the notion of differential checkpointing to be profitable in intermittently-powered systems, as elaborated in Sec. 4.

To that end, as described in Sec. 4, the design of DICE integrates three contributions:

1) unlike previous attempts [3, 8] 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; 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 [6, 7] operate in a *reactive* manner: an interrupt is fired that may preempt the application at *any* point in time. Differently, systems such as MementOS [51] and HarvOS [10] 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. 5 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 [16] and implants [5]. 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. 6, our quantitative assessment in this respect is two-pronged. Sec. 7 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. 8 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 for multi-processing OSes and virtual machines exist. Here, checkpoints are mainly used for fault tolerance and load balancing. Systems use specialized hardware support to compute differentials [48], 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 [29, 43, 50] 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. [18] 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 [44], 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. As shown in Fig. 1, solutions exist that target device architectures that employ non-volatile processors [34, 57, 62] or non-volatile main memory [24], normally FRAM. The former relieve the system from checkpoints altogether, yet require dedicated processor designs still far from massive production. Device employing non-volatile main memories trade increased energy consumption and slower memory access for persistence [24]. 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 [36].

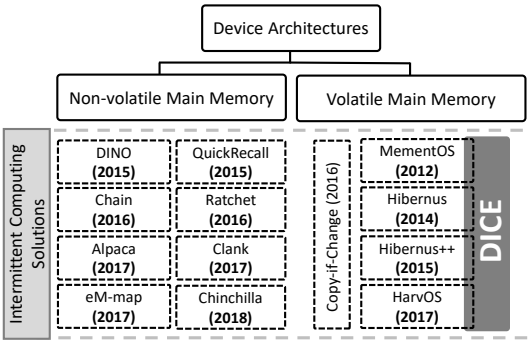


Fig. 1. Intermittent computing solutions. *DICE targets traditional device architectures with volatile main memory and seamlessly integrates with all the corresponding solutions.*

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 [60] or dedicated programming abstractions [15, 37, 39]. 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 [26, 35, 40, 46]. 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 [7] and Hibernus++ [6] 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 [51] and HarvOS [10] 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 [3] or by comparing main memory against a word-by-word sweep of the checkpoint data on NVM [8]. 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.

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 pre-processing or bookkeeping required. The checkpoint data is then ready to be reloaded when computation resumes, with no further elaboration.

3 SCRUTINIZING DIFFERENTIAL CHECKPOINTING TECHNIQUES

Checkpointing system state is widely used to achieve fault tolerance and data consistency in mainstream computing. For example, OS checkpoints are used to recover the system to a stable state after a faulty update [21], distributed system checkpoints are used to combat communication and node failures [59], and database checkpoints are used to maintain a backup at a known good point before applying changes from the log [33]. Although these systems typically do not suffer from limitation of resources such as computation, memory and power, evaluating state differentials is still preferred to spend minimum of these useful resources during checkpoint operations, which restrain the system from doing useful work.

Before embarking on our journey to develop a differential checkpointing solution for resource-constrained embedded devices, we want to scrutinize the existing work for making informed design choices. We discard solutions that rely on specialized hardware support because of their high power requirements, which is prohibitive in energy-constrained platforms. For instance, processors with MMUs expose the state differentials via *dirty-bits* in page tables, clearly specifying the pages altered in a process address space. This information can be exploited to incrementally checkpoint the process state. However, such information is not accessible in embedded MCUs, which usually lack MMU support due to energy limitations [14]. *Software-only* solutions for differential checkpointing also exist that may provide a plausible lead into resolving this limitation of embedded MCUs.

We categorize existing software-only approaches in three broad categories and analyze their suitability for resource constrained platforms, specifically in the context of intermittently-powered ones.

3.1 Hashing

A hashing based approach computes hashes over memory chunks of equal sizes [1, 3, 17]. Since the hash is a function of memory content, a change in memory content of a specific region modifies its hash value. At the time of checkpoint, the current and previous hash values of the target memory region are compared. If these values are different, the corresponding checkpoint in secondary storage is updated along with the newly computed hash. An unmodified hash value indicates that the memory region has not been updated since the last checkpoint, and hence ignored in the interest of reducing expensive write operations in the secondary storage. Multidimensional hashes [19] can also be used to group multiple memory chunks and representing them with a single hash value, which is a function of the hash values of each individual chunk.

Although hashing is a compute-intensive operation, this approach has still proven to be profitable in main stream computing as it avoids checkpointing significant portions of a typically very large main memory. Whether or not this tradeoff plays favorably in embedded systems, remains to be seen.

3.2 Tracking

A dynamic, tracking based approach proactively tracks write accesses in main memory. This is often achieved through specialized function calls instrumented in the code at compile time [44]. These

function calls capture the memory addresses being modified and records them in a separate data structure. At the time of checkpoint, only the content of recorded memory addresses are updated in secondary storage. If a fault occurs, the last stored checkpoint is used to roll back the process state. This approach is particularly useful for long running programs as it avoids re-execution of the entire program by rolling the state back to the nearest known good point.

Deciding what and when to track plays a key role in defining the granularity of differential analysis and, thus, the number of instructions required to be re-executed to reach the original state. A fine-grained analysis requires fewer instructions to be replayed to reach the correct program state at the cost of high runtime overhead. On the contrary, a coarse-grained analysis incurs smaller runtime overhead but increases the re-execution overhead. Thus, the frequency of faults can be a defining factor in determining the tracking granularity.

Tracking changes in memory is apparently less compute-intensive than computing hashes. However, memory access patterns of an application may be crucial in establishing the feasibility of this approach in embedded systems. For example, in applications with frequent memory access operations, the tracking overhead may eventually surpass the benefits accrued from smaller checkpoint data.

3.3 Static analysis

To mitigate the runtime cost of tracking approaches, static analysis can be used to identify variables that are modified by different program execution paths [11, 12, 58]. A key-value data structure is used to create a mapping between different paths and their variables, where the key is the unique path identity and the value is all the variables modified on that path. The runtime only needs to track the execution of paths, and checkpoint all the variables on the executed paths obtained from the key-value data structure.

This approach reduces energy and computational costs as tracking is limited to coarse-grained program paths instead of per variable. However, the memory overhead may increase exponentially for applications with large number of paths, resulting in large key-value stores that record the complete addresses of variables. This approach may be improved by modifying compilers to intelligently allocate main memory [54, 61], and group variables that are modified on a single path. As a result, the key-value data structure will only need to store the start and end address of adjacently allocated group of variables. However, since variables may be modified on multiple paths, such allocation may not be possible in some applications.

Although this approach is attractive due to its negligible computational requirements at runtime, the memory overhead is still a matter of concern on platforms with limited memory capacities.

3.4 Comparative Analysis

As an impetus to our quest for developing a differential state-checkpointing solution for resource-constrained devices, we conduct a "feeler" analysis of these software-based techniques. The ultimate goal of this analysis is to ascertain at a macro level whether a full-fledged implementation of a particular approach is worth the effort for embedded devices. We first elaborate on the experimental settings used for this analysis followed by a one-on-one comparison between different techniques.

Settings. We consider two applications, fast fourier transform (FFT) and activity recognition (AR), widely used as benchmarks in intermittent computing [20, 22]. Both these applications possess useful characteristics needed for our analysis. For example, FFT is a signal processing algorithm that frequently accesses variables in main memory. AR provides large number of paths needed to evaluate the performance of static analysis technique described in Sec. 3.3.

We execute these applications on an MSP430 platform, which is widely considered as a defacto standard in energy-harvesting devices [53]. We are interested in different metrics for different approaches: Since hashing is compute-intensive, we are interested in its *computational* overhead, that is, the time needed to compute and compare hashes of different memory regions and in performing the differential checkpoint operation. Differently, static analysis has a negligible computational overhead but we need to analyze its *memory* requirements for saving the key-value data structure, and whether such requirements can be borne by memory-constrained systems.

Finally, the tracking based approach effects both: it performs additional computations to track memory accesses as well as needs a data structure to record modified memory addresses.

For hashing, we use SHA-1 over memory regions of size 256 bytes. Thus, the entire 10 KB memory is divided in 40 hash regions. The static analysis is performed using the LLVM compiler toolchain to establish a mapping between all program paths and the variables modified by each one of them. We then augment the application with a key-value data structure representing this information at runtime for differential checkpointing. For tracking, we use the code instrumentation technique similar to the one in [44] along with a bit-array data structure to track modifications in memory, where each bit represents one byte (or word) in main memory [2].

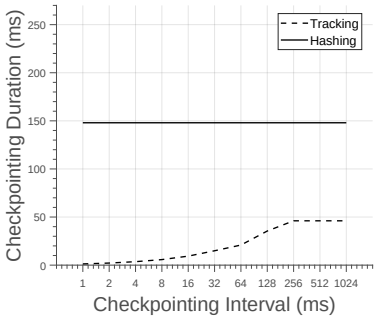


Fig. 2. Computational overhead of differential checkpointing techniques in FFT application.

Computational overhead. Fig. 2 shows the additional time taken by checkpointing operations in FFT application. We omit the static analysis approach from this comparison because of its negligible runtime overhead.

For hashing, this overhead is defined by the time taken for computing and comparing fresh hash values with the previous hash values retrieved from the secondary storage, and updating the checkpoint with modified memory regions. Regardless of the checkpoint interval, hashing based approach has an almost constant but high computational overhead, which is dominated by hash computations.

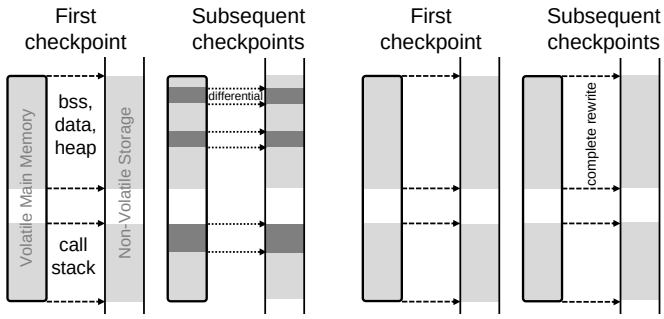
Approach	Memory (bytes)
Static analysis	2240
Tracking	150

Table 1. Memory overhead of different techniques in AR application

On the other hand, the tracking based approach has a variable but low computational overhead. Tracking is comparatively a lighter operation as it only needs a few cycles to update the in-memory data structure for recording modified memory locations. Here the computational overhead is dominated by the checkpoint updates in secondary storage. This overhead understandably increases with checkpointing interval, as more variables are likely modified when the program executes for longer durations due to fewer checkpoint interruptions in between.

Memory overhead. Table 1 shows the additional memory required by different techniques in AR application. This includes the in-memory data structure for recording addresses of modified memory locations and differentially updating the checkpoint with them. The hashing based approach is omitted because it only temporarily occupies memory to compute and compare hash values during checkpoint operations. These hash values are flushed out in secondary storage at the time of checkpoint, and hence, no additional memory is consumed during normal program execution.

We can see that the static analysis based technique results in unbearable memory overhead: 16× greater than the actual size of application. This overhead is, in principle, a function of the



(a) DICE (b) Existing systems

Fig. 3. DICE fundamental operation. *DICE updates checkpoint data based on differentials at variable level in the global context, or with modified stack frames.*

number of paths in a program. The more the number of paths in a program, the more the number of key-value data structures needed to keep path-to-variables mapping.

The tracking based approach performs well in terms of memory overhead as well. The use of bit-array results in restricting the memory overhead of the tracking based approach to at most one-eighth of the memory size.

Summary. Our feeler analysis provides useful observations that can guide our pursuit of a differential checkpointing solution for energy-harvesting devices.

Although the hashing based approach has negligible memory overhead, its heavy computational requirements far exceed the limitations of energy-harvesting devices typically equipped with low-end MCUs. Similarly, the static analysis based approach addresses these limitations of hashing, yet the memory requirements are inconceivable for memory-constrained MCUs. The tracking based approach evidently gets much closer to the sweet spot in this cost benefit spectrum: it efficiently controls both the computational and memory overhead needed to evaluate memory differentials for efficient checkpointing.

While taking note of this feeler analysis, we next present our techniques for differential checkpointing in energy-harvesting embedded systems, whose foundations are firmly held in a tracking based approach. However, the specific constraints of these systems demand specialized tracking techniques based on various memory access patterns and the regions where they occur.

4 DICE IN A NUTSHELL

Fig. 3 describes the fundamental operation of our approach, 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. 3(a). We detail such a process, which we call *recording differentials*, in Sec. 5.

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 [30]. 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.

Technique	Handles heap	Hardware independant performance	Differential checkpoints	Minimizes NVM accesses
MementOS [51]	✗	✓	✗	✗
Hibernus [6, 7]	✓	✗	✗	✗
HarvOS [10]	✗	✓	✗	✗
Copy-if-change [8]	✓	✓	✓	✗
DICE	✓	✓	✓	✓

Table 2. Feature Comparison: DICE and the rest.

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.

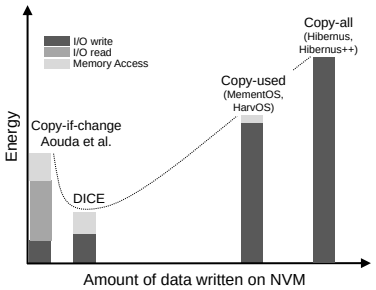


Fig. 4. Qualitative comparison of the checkpoint techniques. Copy-all *has highest energy costs due to maximum writes on NVM.* Copy-used *avoids copying unused memory areas, reducing the energy cost.* Copy-if-change *further reduces energy costs, using NVM reads to compute differentials. DICE records changes in the stack at frame granularity, but only operates in main memory.*

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 [3, 8], 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. 5.

DICE and the rest. Reducing NVM operations is the key to DICE performance. Fig. 4 qualitatively compares the energy performance of checkpointing solutions discussed thus far.

Hibernus [7] and Hibernus++ [6] 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 [51] and HarvOS [10] 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* [8] 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 [42], 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. 7 and Sec. 8 offer quantitative evidence.

Table 2 provide a basic feature comparison between DICE and other techniques. The performance of Hibernus [7] is dependant on the underlying hardware, as its copy-all strategy relies on the availability of FRAM as NVM. Except for the copy-if-change approach [8], none of the other techniques recognize state differentials between checkpoints. DICE recognizes these state differentials but, unlike copy-if-change, does not require costly access to NVM to compute these differentials. Minimizing access to NVM offers great performance benefits in DICE-equipped systems, as we elaborate in Sec. 7.

5 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 [20].

5.1 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.

Fig. 5 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.

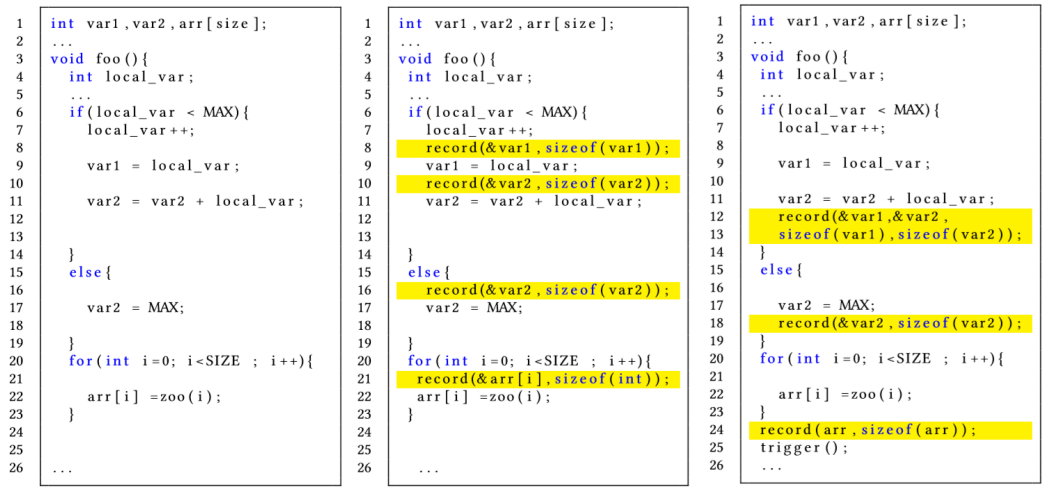
```
int var,*ptr; //global variables
...
record(&var,sizeof(var));
var++;
...
record(&ptr,sizeof(*ptr));
ptr = &var;
```

Fig. 5. Example instrumented code.

Reactive systems. In Hibernus [6, 7], 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 [6, 7], 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. 6(b). This ensures that the modification record is pessimistically updated before the actual change in global context. If a checkpoint happens right after **`record()`**,



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

Fig. 6. 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.*

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 [51] and HarvOS [10] 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. 6(c) shows the same code as Fig. 6(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 run-time 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. 6: a call to **`record()`** inside the loop body, necessary in Fig. 6(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. 6(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 complexity of instrumentation and/or

[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.

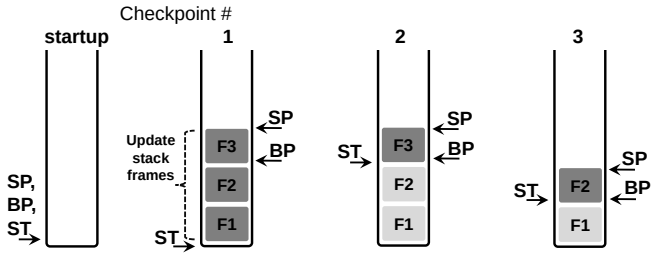


Fig. 7. Identifying possibly modified stack frames. *The stack tracker (ST) is reset to the base pointer (BP) when the system resumes or at startup. ST does not follow BP as the stack grows, but it does so as the stack shrinks. The dark grey region between ST and the stack pointer (SP) is possibly modified.*

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.

5.2 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 [25].

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. 7 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. 7: 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. 7, **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. 7, **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`**.

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. Similarly, multiple frames may enter and exit the stack without checkpointing in between. In such cases, tracking individual local variables may introduce redundant overhead.

5.3 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 (i.e., a global scalar or heap) or a local variable inside a stack frame. In the former case, the modification record is updated as described in Sec. 5.1. 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. 8 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.

```
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
}
```

Fig. 8. Example instrumented code to record local variables that are passed by reference.

6 IMPLEMENTATION

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

Precompiler. We implement the DICE precompiler targeting the C language using ANTLR [47]. The precompiler instruments the entire code, including the run-time libraries, for recording modifications in the global context as described in Sec. 5.1, depending on the underlying system support, and for identifying modified regions of the stack, as explained in Sec. 5.2.

As a result of this instrumentation, DICE captures modifications in main memory except for those caused by peripherals through direct memory access (DMA), which bypass the execution of the main code. 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 either always consider these memory areas as modified, or flag them as modified as soon as the corresponding peripheral interrupts fire, independent of their processing.

The `record()` function. We implement **`record()`** as a variable argument function. In the case of proactive systems, this allows us to aggregate multiple changes in main memory with a single call, as shown in Fig. 6(c).

Among the many data structures available to store the modification records, we choose to employ a simple bit-array, where each bit represents one byte in main memory as modified or not. This representation is particularly compact, causing little overhead in main memory. Crucially, it allows **`record()`** to run in constant time, as it supports direct access to arbitrary elements.

This is key to prevent `record()` from changing the application timings, which may be critical on resource-constrained embedded platforms [64].

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 to track the differentials since the time system restarts.

Checkpoint procedures. We also need to replace the existing checkpoint procedures with a DICE-specific one.

Hibernus [7] and Hibernus++ [6] 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 [10] 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 the slice of the application state that changed, both approaches are overly pessimistic. We set these parameters based on an estimate of the *actual* cost for checkpointing. We obtain this by looking at how many modification records we accumulate and the positions of *ST* and *BP* at a given point in the execution.

Differently, MementOS [51] 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 will generally return a threshold smaller than in the original MementOS, as the energy cost of checkpoints is smaller.

Similar to existing work [37, 51], we also ensure the validity of a checkpoint by adding a canary value at the beginning and at the end of the checkpoint.

7 BENCHMARK EVALUATION

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 the reduction in NVM operations enabled by DICE allows the system to shift part of the energy budget towards useful computations. This reflects into:

- 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, increasing system's responsiveness and despite the overhead of code instrumentation.

In the following, Sec. 7.1 describes the settings, whereas Sec. 7.2 to Sec. 7.5 discuss the results.

7.1 Settings

Benchmarks. We consider three benchmarks widely employed to evaluate system support for intermittently-powered computing [7, 28, 51, 60]: *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 a paradigmatic example of security support on modern embedded systems. Dijkstra's spanning tree algorithm is a staple case of graph processing, often found in embedded network stacks [27].

These benchmarks offer a variety of different programming structures, data types, memory access patterns, and processing load. For example, the FFT implementation operates mainly over variables local to functions and has moderate processing requirements; RSA operates mainly on global data and demands great MCU resources; whereas Dijkstra's algorithm mainly handles integer data types

as opposed to variable-precision ones, but exhibits much deeper levels of nesting due to loops and function calls. This diversity allows us to generalize our conclusions. All implementations are taken from public code repositories [41].

Systems and platforms. We measure the performance of DICE with both reactive (Hibernus) and proactive (MementOS, HarvOS) checkpoints, investigating the different instrumentation strategies in Sec. 5.1. 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* [8] with either of the existing systems. To make our analysis of MementOS independent of the energy traces used to identify a suitable voltage threshold, we manually sweep the possible parameter settings with steps of 0.2V, and always use the best performing one.

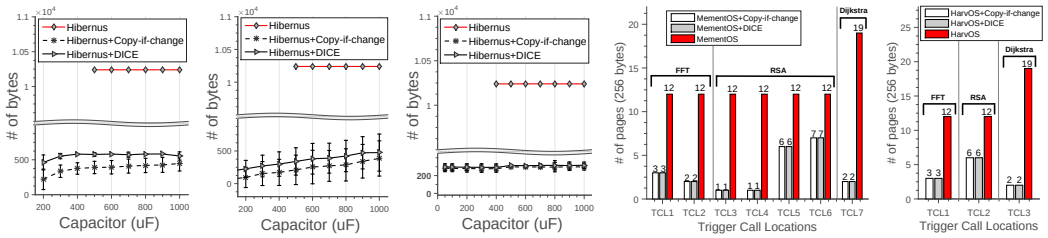
We run Hibernus on an MSP430-based TelosB interfaced with a byte-programmable 128 KByte FRAM chip, akin to the hardware originally used for Hibernus [7]. MementOS and HarvOS run on a Cortex M3-based ST Nucleo with a standard flash chip, already used to compare MementOS and HarvOS [10]. Both boards offer 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 [6, 10, 51], our experiments focus on the MCU. Peripherals may operate through separate energy buffers [23] and dedicated solutions for checkpointing their states also exist [38, 55].

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 smallest amount of energy that allows the system to complete a workload. If too small, a system may be unable to complete checkpoints, ending up in a situation where the execution makes 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 number of times the system must take a checkpoint to complete a workload. The more the checkpoints, the more the system subtracts energy from useful computations. In contrast, 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 the recharge times that are deployment-dependent. 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.

For these experiments, we use a foundational power profile often found in existing literature [8, 28, 51, 60]. The device boots with the capacitor fully charged, 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 generates paradigmatically intermittent executions [7, 28]. The more the environment provided energy while the device is computing, therefore postponing the time when a checkpoint is needed, the more the execution would resemble a traditional one, where no checkpoints are needed. Besides, this profile is also representative of a staple class of intermittently-powered applications, namely, those based on wireless energy transfer [13, 51]. With this technology, devices are quickly charged with a burst of wirelessly-transmitted energy until they boot. Next, the application runs



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

Fig. 9. 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.*

until the capacitor is empty again. The device rests dormant until another burst of wireless energy comes in.

To accurately compute the metrics above, we trace the execution on real hardware using an attached oscilloscope along with the ST-Link in-circuit debugger and Kiel 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.

7.2 Results → Update Size

Fig. 9 shows the results for reduction in update size enabled by DICE. 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. 9(a), 9(b), and 9(c)[2] thus report the average update size we measure during an experiment until completion of the workload, as a function of the capacitor size. Compared to the original *copy-all* strategy, DICE provides orders of magnitude improvements. These gains are a direct result of limiting updates to those determined by the modification records. On the other hand, 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 the original MementOS and HarvOS, the update size is a function of the location of the trigger call, because the stack may have different sizes at different places in the code. Fig. 9(d) and Fig. 9(e)[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.

Overall, 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 [42]. As an example, we compute the energy cost of a checkpoint with the data in Fig. 9(b) 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. Similar considerations apply when using the technique of Aouda et al. [3], due to the

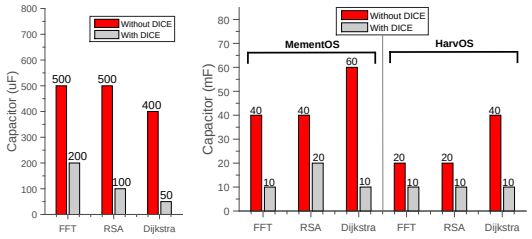
[2]Some data points are missing in the charts for the original design of Hibernus, as it is unable to complete the workload in those conditions. We investigate this aspect further in Sec. 7.4.

[3]For MementOS, the trigger call locations refer to the "function call" placement strategy in MementOS [51]. We find the performance with other MementOS strategies to be essentially the same. We omit that for brevity.

operation of the garbage collector. 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.

7.3 Results → Smallest Energy Buffer

Fig. 10 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 that is up to 88% smaller than the one required with the original *copy-all* strategy. Similarly, for MementOS and HarvOS, the smallest capacitor one may employ is about half the size of the one required in the original designs. Smaller capacitors mean reaching operating voltage faster and smaller device footprints.



(a) Hibernus (b) MementOS and HarvOS

Fig. 10. Smallest capacitor. *A DICE-equipped system completes the workload with smaller energy buffers. This is due to a reduction in the energy cost of checkpoints, enabled by the reduction in update size.*

Such a result is directly enabled by the reduction in the update size, discussed in Sec. 7.2. With fewer data to write on NVM at every checkpoint, their energy cost reduces proportionally. As a result, the smallest amount of energy the system needs to have available at once to successfully complete the checkpoint reduces as well. Provided the underlying system support correctly identifies when to start the checkpoint, the workload can be completed with a smaller capacitor.

7.4 Results → Checkpoints

Fig. 11[4] depicts the reduction in the number of checkpoints against variable capacitor sizes.

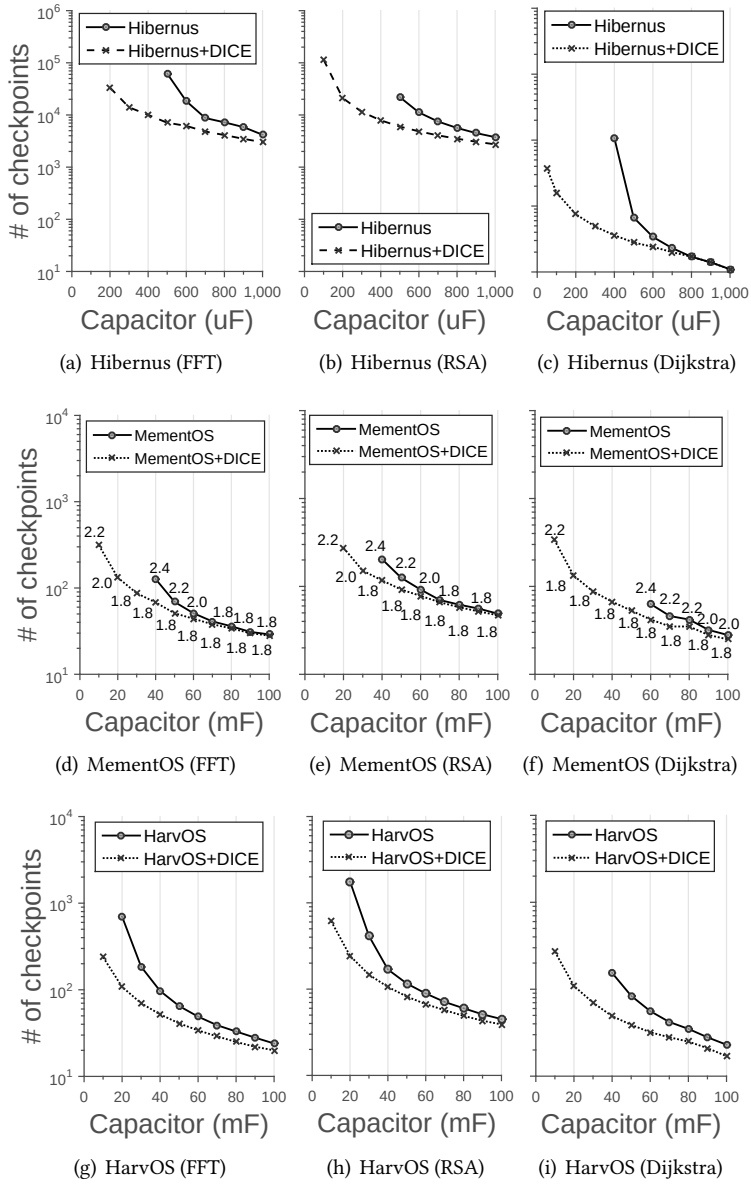
These results are directly enabled by the reduction in update size, discussed in Sec. 7.2. Hibernus adopts the copy-all strategy to checkpoint entire main memory onto NVM regardless of its occupation, as discussed in Sec. 4. As a result, it ends up spending more energy on checkpoints rather than program execution. This ultimately results in Hibernus consuming more checkpoints than MementOS and HarvOS for the same workload, as shown in Fig. 11(a), Fig. 11(d) and Fig. 11(g).

Although MementOS and HarvOS only checkpoint allocated portions of memory, this is still much larger than the actual change in application state. A DICE-equipped system recognizes these state differentials, thus minimizing the amount of energy spent on and the number of checkpoints.

Interestingly, 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 small capacitors. As soon as a comparison is possible, the improvements for DICE with small capacitors are significant and apply consistently across benchmarks as visible in Fig. 11(a), Fig. 11(b) and Fig. 11(c).

With fewer data to write on NVM at every checkpoint, the energy cost of this operation reduces proportionally. This has two direct consequences. First, the smallest amount of energy the system needs to have available at once to complete the checkpoint reduces as well. Smaller capacitors mean reaching operating voltage faster and smaller device footprints. Second, the system can invest the available energy to compute rather than checkpointing, improving the overall energy efficiency. In this setting, DICE provides the greatest advantages.

[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 [51].



(a) Hibernus (FFT) (b) Hibernus (RSA) (c) Hibernus (Dijkstra)

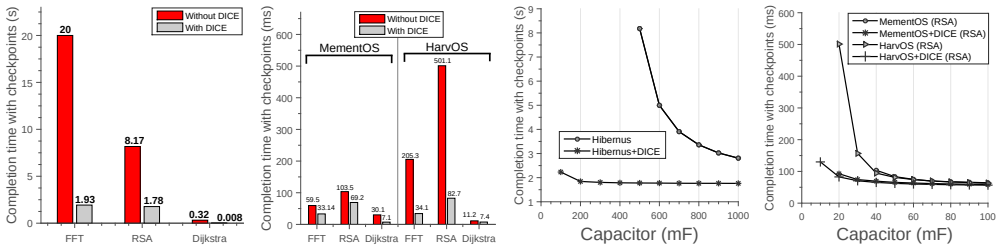
(d) MementOS (FFT) (e) MementOS (RSA) (f) MementOS (Dijkstra)

(g) HarvOS (FFT) (h) HarvOS (RSA) (i) HarvOS (Dijkstra)

Fig. 11. Number of checkpoints necessary against varying capacitor sizes. *With smaller capacitors, highly-intermittent executions greatly benefit from DICE.*

With larger capacitors, the improvements in Fig. 11 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. As a consequence, modifications since 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.

[5]Note the log scale on the Y axis of Fig. 11.

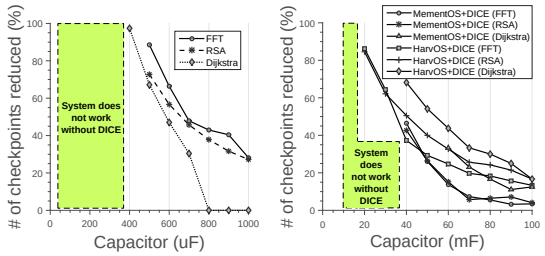


(a) Hibernus (b) MementOS and HarvOS (c) Hibernus (RSA) (d) MementOS and HarvOS (RSA)

Fig. 12. Completion time including checkpoints. *The run-time overhead due to DICE is overturn by reducing size and number of checkpoints, ultimately resulting in shorter completion times.*

Fig. 13 provides a summary view on these results, plotting the percentage reduction unlocked by DICE against variable capacitor sizes and across systems.

A significant part of the chart is empty as no comparison is possible, because the workload cannot be completed in the original designs. When a comparison is possible, improvements are largest with smaller capacitors, topping above 80% in most cases. The curves ultimately tend to flatten with larger capacitors for the aforementioned reasons.



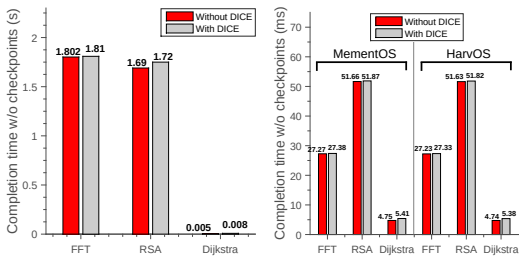
(a) Hibernus (b) MementOS and HarvOS

Fig. 13. Improvement at different capacitor sizes enabled by DICE. *The improvements are larger in highly-intermittent executions.*

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

7.5 Results → Completion Time

DICE naturally imposes a cost for the benefits reported thus far. Such a cost materializes as run-time overhead due to recording differentials, which may increase the time to complete a given workload. 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. Both factors should reduce the completion time.



(a) Hibernus (b) MementOS and HarvOS

Fig. 14. Completion time without concrete checkpoints. *The additional run-time overhead due to code instrumentation is limited.*

Fig. 14 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 executing **`record()`**. The chart shows that this overhead is generally limited. This is valid also for reactive

checkpoints in Hibernus, despite the conservative approach at placing **`record()`** calls due to the lack of knowledge of where the execution is preempted.

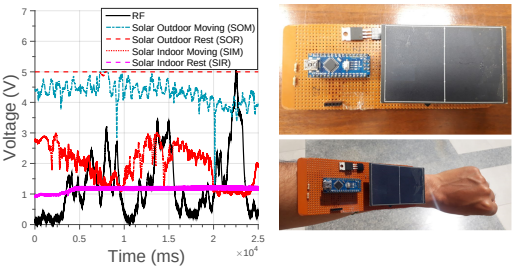
Fig. 12 includes the time required for checkpoint operations with the smallest capacitor allowing both the DICE-equipped system and the original one to complete the workload, as per Fig. 10. The overhead due to executing **`record()`** calls is not only compensated, but actually overturn by fewer, more rapid checkpoints. Using these configurations, DICE allows the system to complete the workload much earlier, increasing the system's responsiveness.

Differently, Fig. 12(c) and Fig. 12(d) provide an example of the trends in completion time against variable capacitor sizes. The improvements are significant in a highly-intermittent computing setting with smaller capacitors. Similar to Fig. 11, two factors contribute to the curves in Fig. 12(c)and Fig. 12(d) approaching each other. Larger capacitors allow the code to make more progress on a single charge, so the number of required checkpoints reduces. As more processing happens between checkpoints, more modifications occur in application state, forcing DICE to update a larger portion of the checkpoint data. Eventually, these two factors compensate each other.

8 APPLICATION EVALUATION

We investigate the impact of DICE in a full-fledged application, using a variety of different power profiles. We build the same activity recognition (AR) application used for evaluating several support systems for intermittent computing [15, 37, 39, 51], including relying on the same source code [20]. We use an MSP430F2132 MCU, that is, the MCU used in the WISP platform [53] for running the same AR application. The rest of the setup is as for Hibernus in Sec. 7.

We focus on number of checkpoints and completion time, as defined in Sec. 7.1, in a single application iteration.



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

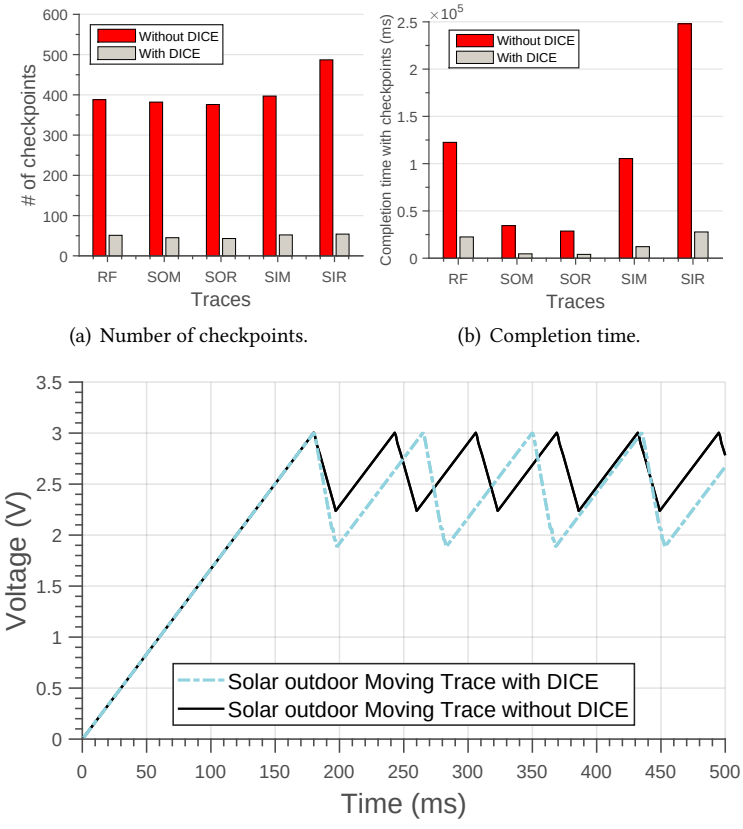
Fig. 15. Example voltage traces and device setup.

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

One of the traces is the RF trace from MementOS [31, 51], recorded using the RF front end of the WISP 4.1. The black curve in Fig. 15(a) shows an excerpt of this trace, plotting the instantaneous voltage reading at the energy harvester over time. The curve is oscillating as it was recorded by a person carrying the WISP while walking around an RFID reader within about eight feet of range.

We collect four additional traces using a mono-crystalline high-efficiency solar panel [56], shown in Fig. 15(b), in different settings. The solar panel has high efficiency ($\approx$ 20%) with good response to both indoor and outdoor lightning. We use an Arduino Nano [4] to measure the voltage output across a 30kOhm load, roughly equivalent to the resistance of an MSP430F2132 in active mode, attached to the solar panel.

Using this device setup, we experiment with different settings. 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 our 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). The curves other than the black one in Fig. 15(a) exemplify the trends.



(a) Number of checkpoints.

(b) Completion time.

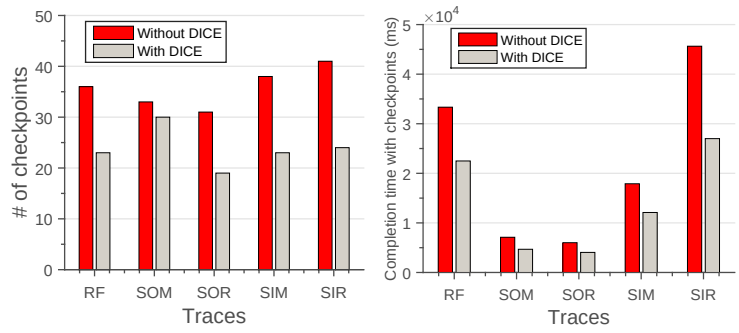
(c) Voltage Variation

Fig. 16. Performance of the AR application running on Hibernus with a 50uF capacitor. *Performance gains are observed across diverse power traces obtained from different energy sources.*

Overall, Fig. 15(a) visually demonstrates the extreme variability and considerable differences among the power traces we consider. For example, the RF power trace often reaches a 0V output, due to multipath loss worsened by RF harvester mobility. Conversely, the solar traces always maintain a non-zero output, yet with crucial differences. When operating outdoor, the output voltage rarely falls below the operating voltage of the hardware we use, whereas this happens frequently in an indoor setting. Orthogonal to this, mobility induces substantial variations in the voltage output, due to shadows and occlusions, whereas the output is nearly constant in a static setup.

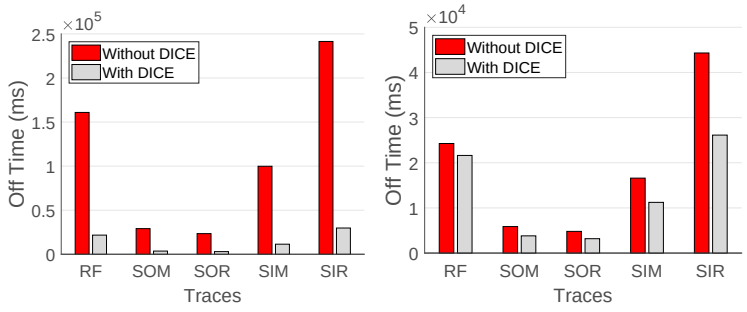
Results. Across all power traces, we find out that a 10uF capacitor is sufficient for a DICE-equipped Hibernus to complete an iteration of the AR application. Without DICE reducing the size of checkpoints, and thus their energy cost and time, Hibernus needs a five times larger capacitor to complete the same workload.

Fig. 16 shows the performance with a 50uF capacitor, where both the original Hibernus and the DICE-equipped one can complete the workload. The number of required checkpoints diminishes by one order of magnitude using DICE, as shown in Fig. 16(a), and we appreciate similar improvements for completion time as well, as shown in Fig. 16(b). Fewer checkpoints and shorter completion times mean better energy efficiency and increased reactivity to external events, ultimately providing increased quality of service to end users.



(a) Number of checkpoints. (b) Completion time.

Fig. 17. Performance of the AR application running on Hibernus with a 100uF capacitor. *Compared with Fig. 16, and as seen in Sec. 7, performance improvements are larger with more intermittent executions.*



(a) Off time for 50uF. (b) Off time for 100uF.

Fig. 18. Performance of the AR application running on Hibernus with a 50uF and 100uF capacitors. *Reduced off-time is observed across diverse power traces obtained from different energy sources.*

Best absolute performance in Fig. 16 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 which of the diverse power traces is considered. This performance evidently originates from the ability of DICE to abate the energy costs of and time taken for checkpoints. We achieve this by limiting NVM operations to updating the relevant slices depending on changes in the application state, as described in Sec. 4. This observation is confirmed also by looking at the voltage threshold Hibernus uses with DICE: on average, checkpoints start at 2.07V, rather than 2.8V as required in the original Hibernus as shown in 16(c).

The trends we discuss in Sec. 7 with larger capacitors are confirmed here. Fig. 17 plots the performance using a 100uF capacitor. Improvements are reduced compared with Fig. 16: applications progress farther on a single charge, checkpoints are sparser, and DICE records more changes to the application state between checkpoints. This reflects also in the voltage threshold used for triggering a checkpoint: the DICE-equipped Hibernus triggers a checkpoint at 1.93V, whereas the original Hibernus starts checkpoints at 2.4V.

We also analyze the impact of reduced checkpoint size on *off-time*, the total amount of time the device spends in recharging the buffer. It reflects on device's ability to react to external events, network on time, and quality of service in general. Fig. 18(a) shows the off-time on a 50uF capacitor. A DICE equipped system achieves significantly smaller off-times as compared to original Hibernus, as shown in Fig. 18. This ability stems from DICE's ability to reduce the number of checkpoints

and, therefore, the number of recharge cycles needed to complete the workload. Fig. 18(b) plots the performance using a 100uF capacitor. Improvements are reduced compared with Fig. 18(a) as the larger capacitor also benefits the original systems.

9 CONCLUSION

DICE is a set of compile-time techniques for intermittently-powered computers to reduce the amount of NVM operations during checkpoints. To do so, our compile-time instrumentation tracks changes in application state and records them in main memory. Based on this, DICE updates the existing checkpoint data by limiting NVM operations to those strictly necessary, helping existing systems complete a given workload with *i)* fewer checkpoints, thus better energy efficiency, and *ii)* shorter times, thus increased reactiveness and better quality of service.

Our benchmark evaluation, based on a combination of three benchmarks across three different existing system support and two different hardware platforms, provides quantitative evidence. For example, using DICE, HarvOS can complete the execution of the RSA algorithm with 86% fewer checkpoints, resulting in better overall energy utilization and a 34% reduction in completion time. These improvements are confirmed in concrete applications, yielding better energy efficiency and increased reactivity to external events. Experiments based on an activity recognition application show order of magnitudes improvements when using Hibernus and against power traces as diverse as RF-based wireless energy transfer and solar radiation.

Acknowledgments. This research has been partially supported by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. 2004. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 277–286.

[2] 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 International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, 70–81.

[3] 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*. IEEE, 1–4.

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

[5] 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*.

[6] 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), 1968–1980.

[7] 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), 15–18.

[8] Naveed Bhatti and Luca Mottola. 2016. Efficient state retention for transiently-powered embedded sensing. In *International Conference on Embedded Wireless Systems and Networks*. 137–148.

[9] 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), 24.

[10] 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*. ACM, 209–219.

[11] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 351–366.

[12] Florian Brandner, Benoit Boissinot, Alain Darte, Benoît Dupont De Dinechin, and Fabrice Rastello. 2011. *Computing Liveness Sets for SSA-Form Programs*. Research Report RR-7503. INRIA. 25 pages. https://hal.inria.fr/inria-00558509

[13] Michael Buettner, Benjamin Greenstein, and David Wetherall. 2011. Dewdrop: An Energy-Aware Runtime for Computational RFID. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 197–210.
[14] Hsung-Pin Chang, Yen-Ting Liu, and Shang-Sheng Yang. 2014. Surviving sensor node failures by MMU-less incremental checkpointing. *Journal of Systems and Software* 87 (2014), 74–86.
[15] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 514–530.
[16] 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), 269–281.
[17] Kurt B Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. 2011. libhashckpt: hash-based incremental checkpointing using gpu's. In *European MPI Users' Group Meeting*. Springer, 272–281.
[18] M. Muztaba Fuad and Michael J. Oudshoorn. 2007. Transformation of Existing Programs into Autonomic and Self-healing Entities. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. IEEE Computer Society, 133–144.
[19] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. 2003. Caches and Hash Trees for Efficient Memory Integrity Verification. In *HPCA*. IEEE Computer Society, 295–306.
[20] Abstract Research Group. 2018. *Benchmark Applications*. www.github.com/CMUAbstract/releases#benchmark-applications ((accessed 2018-02-28)).
[21] Berkin Güler and Öznur Özkasap. 2018. Efficient checkpointing mechanisms for primary-backup replication on the cloud. *Concurrency and Computation: Practice and Experience* 30, 21 (2018), e4707.
[22] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*. IEEE, 3–14.
[23] 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*. ACM, 5–16.
[24] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, Article 19, 13 pages.
[25] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 228–240.
[26] Texas Instruments. 2013. *MSP430 Solar Energy Harvesting Development Tool*. http://www.ti.com/tool/EZ430-RF2500-SEH (accessed 2018-08-03).
[27] 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), 16–22.
[28] 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), 8.
[29] Richard Koo and Sam Toueg. 1987. Checkpointing and Rollback-recovery for Distributed Systems. *IEEE Transactions on Software Engineering* (1987), 23–31.
[30] Phil Koopman. 2010. *Better Embedded System Software*. CMU Press.
[31] PERSIST Lab. 2018. *RF Trace*. https://github.com/PERSISTLab/BatterylessSim/tree/master/traces ((accessed 2018-02-28)).
[32] Eunji Lee, Seunghoon Yoo, Jee-Eun Jang, and Hyokyung Bahn. 2012. Shortcut-JFS: A write efficient journaling file system for phase change memory. In *28th Symposium on Mass Storage Systems and Technologies*. IEEE, 1–6.
[33] Juchang Lee, Kihong Kim, and Sang Kyun Cha. 2001. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 173–182.
[34] Fuyang Li, Keni Qiu, Mengying Zhao, Jingtong Hu, Yongpan Liu, Yong Guan, and Chun Jason Xue. 2018. Checkpointing-Aware Loop Tiling for Energy Harvesting Powered Nonvolatile Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 1 (2018), 15–28.
[35] Libelium. 2017. *Waspmote*. http://www.libelium.com/products/waspmote/ (accessed 2017-10-02).
[36] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages*. Leibniz International Proceedings in Informatics.
[37] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*,

575–585.
[38] Giedrius Lukosevicius, Alberto Rodriguez Arreola, and Alexander Weddell. 2017. Using Sleep States to Maximize the Active Time of Transient Computing Systems. In *ENSsys (with SenSys)*. ACM.
[39] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM on Programming Languages* 1, Article 96 (2017), 30 pages.
[40] R. Margolies et al. 2013. Demo: An Adaptive Testbed of Energy Harvesting Active Networked Tags (EnHANTs) Prototypes. In *IEEE INFOCOM'13*.
[41] mbed. 2017. *IoT OS*. goo.gl/u918jX.
[42] Kresimir Mihic, Ajay Mani, Manjunath Rajashekhar, and Philip Levis. 2007. Mstore: Enabling storage-centric sensornet research. In *IPSN*. Citeseer, 1–8.
[43] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 284–295.
[44] Robert HB Netzer and Mark H Weaver. 1994. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI*, Vol. 94. ACM, 313–325.
[45] Hoang Anh Nguyen, Anna Forster, Daniele Puccinelli, and Silvia Giordano. 2011. Sensor node lifetime: An experimental study. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. IEEE, 202–207.
[46] Expansion of STM32 Nucleo boards. 2017. *Data Sheet: X-NUCLEO-NFC02A1*. www.st.com (accessed 2017-10-02).
[47] Terence Parr. 2013. *The Definitive ANTLR 4 Reference*. goo.gl/RR1s.
[48] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*. USENIX Association, 18–18.
[49] Xiongpai Qin, Yanqin Xiao, Wei Cao, and Shan Wang. 2008. A parallel recovery scheme for update intensive main memory database systems. In *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 509–516.
[50] Brian Randell, Pete Lee, and Philip C. Treleaven. 1978. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)* 10, 2 (1978), 123–165.
[51] Benjamin Ransford. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 159–170.
[52] B. Ransford. 2013. *Transiently Powered Computers*. Ph.D. Dissertation. School of Computer Science, UMass Amherst.
[53] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement* 57, 11 (2008), 2608–2615.
[54] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: An LLVM-based Static Binary Translator. In *CASES*. ACM, 51–60.
[55] Rebecca Smith and Scott Rixner. 2015. Surviving Peripheral Failures in Embedded Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX Association, 125–137.
[56] IXYS SolarMD. 2018. *SLMD481H08L*. http://ixapps.ixys.com/ ((accessed 2018-02-28)).
[57] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnan Narayanan. 2017. Nonvolatile processors: Why is it trending?. In *2017 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 966–971.
[58] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 265–266.
[59] Florin Sultan, Thu Nguyen, and Liviu Iftode. 2000. Scalable fault-tolerant distributed shared memory. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 20–20.
[60] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention.. In *OSDI*. USENIX Association, 17–32.
[61] Ramakrishnan Venkitaraman and Gopal Gupta. 2004. Static Program Analysis of Embedded Executable Assembly Code. In *CASES*. ACM, 157–166.
[62] Mimi Xie, Mengying Zhao, Chen Pan, Jingtong Hu, Yongpan Liu, and Chun Jason Xue. 2015. Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor. In *DAC*. 1–6.
[63] 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).
[64] 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*. ACM, 189–203.