Liveness-Aware Checkpointing of Arrays for Efficient Intermittent Computing

Youngbin Kim, Yoojin Lim and Chaedeok Lim*
yb.kim@etri.re.kr, yoojin.lim@etri.re.kr, cdlim@etri.re.kr
Electronics and Telecommunications Research Institute (ETRI), Daejeon, Republic of Korea

***Abstract*—Intermittent computing enables computing under environments that may experience frequent and unpredictable power failures, such as energy harvesting systems. It relies on checkpointing to preserve computing progress between power cycles, which often incurs significant overhead due to energy-expensive writes to Non-Volatile Memory (NVM). In this paper, we present LACT (Liveness-Aware CheckpoinTing), as an approach to reducing the size of checkpointed data by exploiting the liveness of memory objects: excluding dead memory objects from checkpointing does not affect the correctness of the program. Especially, LACT can analyze the liveness of arrays, which take up most of the memory space but are not analyzable by existing methods for detecting the liveness of scalar objects. Using the liveness information of arrays, LACT determines the minimized checkpoint range for the arrays at compile time without any runtime addition. Our evaluation shows that LACT achieves an additional reduction of checkpointed data size of 37.8% on average over the existing state-of-the-art technique. Also, our experiments on a real energy harvesting environment show that LACT can reduce the execution time of applications by 27.7% on average.**

***Index Terms*—intermittent computing, liveness analysis**

I. Introduction

Energy harvesting systems perform computing relying on the energy collected from the environment, enabling battery-less computing systems. They are known to have several advantages over battery-powered platforms, including long lifetime, ease of maintenance, and less environmental impacts [1], [2]; thus, they have been adopted widely in application domains such as the Internet of Things devices, sensors, and wearables [3]–[5].

Meanwhile, power gathered from an environment is often unstable [6] and insufficient to provide energy for continuous computing. As a result, the computing system may experience power failure at any time of its execution. Computing in such an environment, known as *intermittent computing*, requires a mechanism to make a program progress even under unpredictable and frequent power failures.

To preserve computing progress between power cycles, intermittent systems rely on checkpoint and recovery. Along with traditional volatile components (i.e., memory and registers), they are equipped with Non-Volatile Memory (NVM), which can retain data during power-offs. Intermittent system regularly saves its volatile states into NVM during execution (*checkpoint*). Once power fails and the energy storage is recharged again, the system recovers its previous state from the checkpoint and resumes the computation (*recovery*).

This work was supported by IITP grant funded by the Korea government (MSIT) (No.2021-0-00360, Development of Core Technology for Autonomous Energy-driven Computing System SW in Power-instable Environment).

*Corresponding author: Chaedeok Lim. Email: cdlim@etri.re.kr

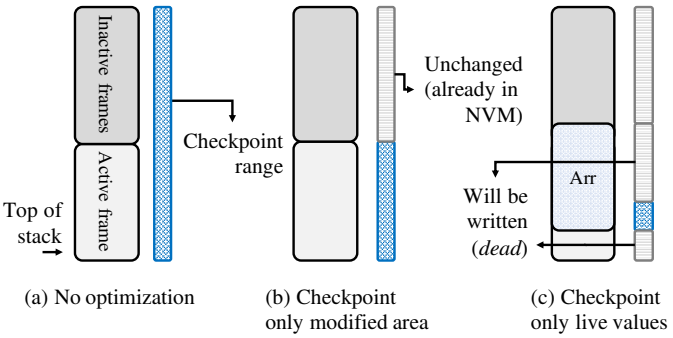


Fig. 1: Stack checkpoint size reduction techniques.

Since checkpointing involves a number of energy-expensive NVM writes, minimizing its cost is critical for efficient intermittent computing. One approach to reducing the checkpoint size (i.e., the amount of data to checkpoint) is to track and checkpoint only modified memory areas, instead of checkpointing the entire memory every time [3], [4], [7], [8]. These works record memory modifications between subsequent checkpoints and only checkpoint the changes, as the unmodified areas are already on NVM. Since tracking every single modification at run-time incurs considerable overhead, they propose using different tracking techniques (e.g., hash [7] or in-memory data structure [4]) and granularities (e.g., from per-value to stack frames [3], [4], [7]) to maximize the tradeoff between tracking accuracy and its overhead.

However, although most redundant memory copies can be eliminated by previous techniques, not *all* of the modified memory areas should necessarily be checkpointed. In general, there exist *dead* memory areas (i.e., which will not be read in the future) at the time of checkpointing, and excluding them from the checkpoint does not affect the correctness of a program. Such *liveness-aware* checkpointing can contribute to further reducing the size of the checkpoint.

Fig 1 shows a comparison between the previous works and liveness-aware checkpointing, as an example in a stack. The required checkpoint range is depicted on the right of each figure representing different techniques. Fig 1b shows a previous technique [4] that tracks modifications of a stack at a granularity of stack frame. While it can detect unmodified stack areas (i.e., inactive frames) and exclude them from the

978-3-9819263-7-8/DATE23/© 2023 EDAA

checkpoint, the entire active frame (i.e., the stack frame of the currently executed function) is checkpointed in this technique. On the other hand, liveness-aware checkpointing (Fig 1c) can even optimize the checkpoint of *active frame* by ignoring dead values, thus further reduces the required checkpoint size.

In practice, the benefit of liveness-aware checkpointing becomes significant especially when it makes (part of) arrays be excluded, as arrays tend to take up a large portion of the memory space. The challenge is that, unlike scalar variables or registers [9], arrays are not atomically modified by a single instruction. Instead, most arrays are accessed or modified progressively along with loop iterations. This complicates realizing liveness-aware checkpointing of arrays since it requires element-wise liveness information [10], [11], which cannot simply be gathered from the existing scalar-based analysis.

In this paper, we propose LACT (Liveness-Aware CheckpoinTing), a compile time analysis and optimization method to enable liveness-aware checkpointing of arrays and stack objects. LACT can detect element-wise dependencies of arrays at any iteration in the loop, and statically inserts optimized checkpoint calls without requiring run-time additions. In summary, this paper makes the following contributions:

- LACT achieves an additional stack checkpoint size reduction of 37.8% over previous state-of-the-art technique [4].
- LACT is orthogonal to the existing approaches [3], [4], [7], [8]. When both techniques are applied simultaneously, stack checkpoint size can be reduced by 52.5%.
- Our evaluation on the energy harvesting environment demonstrates that LACT can reduce the execution times of intermittent computing applications by 27.7%.

II. Background and Related Work

A. SRAM-based vs. NVM-based Main Memory

Intermittent systems can be broadly classified as a traditional system using SRAM for main memory [6], [12]–[15], and one utilizing NVM as a main memory [1], [9], [16]–[18]. NVM-based architecture can significantly reduce checkpoint size since it eliminates the need for checkpointing memory; the registers are the only volatile state. On the other hand, such benefit comes with a tradeoff: it has its own challenges and limitations, which include memory inconsistency [9], slow and power-consuming memory accesses [3], [19], and/or SRAM power leakage (e.g., for off-the-shelf platforms, it is often impossible to turn off or remove the SRAM completely).

In this paper, we evaluate the efficiency of LACT on SRAM-based architecture. However, NVM-based platforms, especially ones determining checkpoint sites in compile-time [1], can benefit from our liveness analysis since they also need a mechanism to record memory modifications to avoid inconsistent memory states [16], [17].

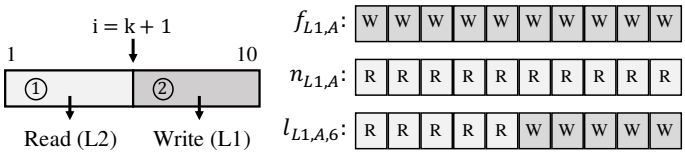
B. Techniques for Reducing Checkpoint Overhead

In SRAM-based intermittent systems, efforts to reduce checkpoint overheads can be classified into two approaches. One approach tries to reduce the checkpoint size by copying only modified memory areas, as introduced in Section I [3], [4], [7], [8]. The second approach focuses on making the system execute checkpointing as little as possible. Early approaches check the energy buffer and execute checkpoints only when the energy level is low [6]. This scheme is further improved by the studies that minimize the checkpoint triggers by exploiting the system's energy buffer assumptions [1], [13]. On the other hand, [14], [15] attempt to execute checkpointing based on interrupts, which are triggered only when the available energy goes below a threshold.

LACT exploits a different optimization opportunity (i.e., the liveness of memory objects), so it can be applied along with most previous techniques. Our evaluation in Section IV shows that LACT can be applied on top of the existing technique [4] and achieves considerable additional improvements.

```
L1: for i = 1 to n { Arr[i] = foo(i) }
L2: for i = 1 to n { print(Arr[i]) }
```

(a) Example code consisting of two loops, $L1$ and $L2$.



(b) Checkpointing `Arr` in $L1$, where $i = k + 1$.

(c) The access vectors computed during the analysis.

Fig. 2: Example code and analysis steps of LACT.

C. Element-wise Liveness Analysis

Element-wise liveness analysis has been studied in the context of optimizing memory allocation [10], [11]. These works utilize polyhedral analysis to detect the elements becoming dead during the loop execution. Then they reuse such memory areas for future iterations to minimize the memory footprint of the program, especially for large multi-dimensional arrays. However, these works only consider the liveness of array elements in a single loop, rather than considering the entire CFGs, which is mandatory in the context of checkpointing. LACT analyzes the entire program with a more lightweight implementation and also includes an analysis to determine the minimized checkpoint range.

III. Liveness-Aware Checkpointing

A. Motivational Example

Consider a simple program in Fig 2a consisting of two loops, $L1$ and $L2$. In this example, $L1$ initializes an array of length 10, and $L2$ prints its elements. If we want to checkpoint `Arr` during the execution of $L1$, before starting iteration $i = k + 1$, what could be the smallest address range that we *must* checkpoint while keeping the semantics of the program correct? To answer this question, we need to know the *liveness* of each array element: whether it will be read (i.e., *live*) or written (i.e., *dead*) after the checkpoint execution. We may checkpoint only live elements and ignore the dead elements, as dead elements will be overwritten anyway.

For `Arr[k+1:10]`, we can infer that they are dead and thus can be excluded from checkpointing, as the remaining iterations will overwrite them. On the other hand, for `Arr[1:k]`, we need to consider the access information after $L1$ to decide their liveness. In this example, since $L2$ will load `Arr[1]` to `Arr[k]` to print them, they are live and should be checkpointed. As a result, the checkpoint range for `Arr` in loop $L1$ can be reduced to $[1, k]$, rather than the entire array.

In subsequent sections, we describe our design and approach to statically analyze the liveness of arrays in a program to determine minimized checkpoint ranges.

B. Methodology

Our goal is to design a compile-time analysis that enables minimizing checkpoint size of arrays by computing their liveness and saving only live elements. More specifically, we find a mapping $\mathbb{R} : (L, A) \rightarrow [i, j]$ where L is a loop, A is an array in the program, and $[i, j]$ is the minimized checkpoint range for A that preserves the correctness of the program, when the checkpoint is executed during L. As in Section III-A, the range i and j can be either a constant or an expression of k, which is the induction variable of L (e.g., $[1 : k]$).

Fig 2b summarizes the discussion about where the liveness information can be obtained from (Section III-A). As in the figure, some parts of data dependency can be resolved in the currently executing loop (②), but others require the access information after the loop (①) — fundamentally because we execute checkpointing in the middle of the loop iterations. Consequently, LACT computes two kinds of access information for each loop and array in the program: *first access vector* (FAV) and *next access vector* (NAV). The former holds access information inside the loop (e.g., for ②), and the latter keeps the access patterns after executing the loop (e.g., for ①). Section III-C and III-D describe the precise definitions of these vectors and present our approach to compute them.

Once FAVs and NAVs are computed, exact checkpoint ranges $\mathbb{R}$ can be derived from these vectors. Section III-E describes the procedure to compute such an optimized checkpoint range.

Finally, we present a liveness-aware checkpointing technique on scalar stack objects. Since they are typically small in size compared to arrays, the primary focus in this step is minimizing the overhead of the optimization, while keeping its benefit at a reasonable level. Section III-F presents our approach to implement such optimization in a lightweight manner.

Throughout this section, we assume that checkpoints are inserted at the header of loops (as in [6]), without loss of generality. Also, we use the following definitions and notations to keep the text concise:

- Σ Set of operations, equals to $\{R, W, N\}$. Each element represents *Read*, *Write* and *None*, respectively.
- $\mathbf{X}^n$ A vector of X, of length n. i.e., $(X, X, ..., X) \in \Sigma^n$.

C. Finding First Access Vector (FAV)

As an initial step, we compute *First Access Vector* (FAV), denoted as $f_{L,A}$ where L is a loop and A is an array in the program. It is a vector in Σ^n, where n is the length of A. Here, ith element of the vector $f_{L,A}(i)$ represents the type of the first access to $A[i]$ in L. For example, in our example in Fig 2a, $f(L1, \texttt{Arr}) = \mathbf{W}^n$ since the first access to every element of `Arr` in $L1$ is write (Fig 2c also illustrates the resulting vector). Note that if there exist multiple accesses to an array in the loop, FAV analysis considers only the first access to each element as that determines the data dependency.

We utilize an existing analysis called Scalar Evolution (SCEV) in LLVM [20] to compute FAVs at compile time. SCEV analyzes the change of variables over iterations of the loop and represents the results in the form of *recurrences* [21]. Recurrence consists of the initial value and the update function (e.g., stride) in its basic form, and also it is possible to operate on them (e.g., adding two recurrences) to compute the recurrences of expressions (i.e., *chains of recurrences algebra* [21]).

We first find the instructions accessing A in L, and build access vectors of them using SCEV results. This work only considers array accesses that have a stride of 1 and leaves more complex forms for future work. Owing to this assumption, we can always decide which access instruction precedes another (e.g., $A[k + 1]$ precedes $A[k]$) thus FAV can be computed by merging these access vectors. For array accesses whose range is too complex or cannot be determined statically (e.g., `A[foo(i)]`), we fill the entire vector with pessimistic assumptions — with R when the access is a load (to make *all* elements live), or N when the access is a store (not to mark *any* of the elements to be dead).

D. Computing Next Access Vector (NAV)

Next Access Vector (NAV) $n_{L,A}$ is also a vector in Σ^n, but it represents the first accesses after finishing L. For example, in our example in Fig 2b, $n_{L1,\texttt{Arr}} = \mathbf{R}^n$, since after finishing $L1$, all elements are read in the subsequent execution in $L2$ (Depicted as the second vector in Fig 2c).

Computing NAVs involves CFG traversal since 1) there can exist multiple exiting paths from a loop, and 2) loops may partially access an array. Algorithm 1 shows our traversal procedure to find NAVs. Function FINDNAV (line 1) is responsible for finding $n_{L,A}$. It sets a vector $nav = \mathbf{N}^n$ and repeats traversing CFG until the liveness of every element is found, starting from the exit blocks of L (line 2-4). In each iteration, FINDNEXT returns possible first accesses from the current basic blocks (line 5), and nav is updated accordingly (line 6).

The actual traversal happens in FINDNEXT (line 5). It finds *every possible* first access to A when execution starts from one of the basic blocks in $startBBs$. For this purpose, it traverses down to CFG (line 20) until finding access to A (line 15). Once such access is found, the access information can be obtained from pre-computed FAV (line 16-17).

Since there may exist multiple starting blocks and/or control can diverge during the traversal, FAVs obtained from different paths should be merged together (line 17). Function MERGE shows the process of merging an FAV f into a vector v (line 22). As discussed in Section III-C, merges should be done in a conservative manner: if there exists an element that may be both read and written, the merged vector should be R (line 23-24) to keep it alive.

Algorithm 1 Computing Next Access Vector (NAV)

```
1: function FINDNAV(loop L, array A)
2:     nav ← N^n                                  ▷ n: length of A
3:     startBBs ← exit blocks of L
4:     while startBBs ≠ ∅ and ∃N ∈ nav do
5:         v, endBBs ←FINDNEXT(A, startBBs)
6:         nav_k ← v_k for ∀k where nav_k = N and v_k ≠ N
7:         startBBs ← endBBs
8:     return nav
9: function FINDNEXT(array A, startBBs)
10:    visited, endBBs ← ∅, ∅
11:    Push ∀bb ∈ startBBs to stack toVisit
12:    v ← N^n                                    ▷ n: length of A
13:    while toVisit ≠ ∅ do
14:        bb ← pop an element from toVisit
15:        if bb has access to A then
16:            l ← loop containing bb
17:            v ← MERGE(v, f_{l,A})   ▷ f_{l,A}: FAV for l and A
18:            Add bb into endBBs
19:        else
20:            Add successors of bb into toVisit
21:    return v, endBBs
22: function MERGE(v, f)
23:    for all k s.t. v_k = N do v_k ← f_k
24:    for all k s.t. v_k = W and f_k = R do v_k ← R
25:    return v
```

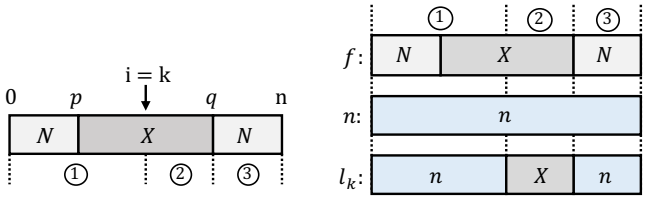
E. Determining Checkpoint Range

By combining $f_{L,A}$ and $n_{L,A}$, we can compute a *liveness vector* $l_{L,A,k} \in \Sigma^n$, which represents element-wise liveness of A in loop L at iteration k. Fig 2c shows such a procedure in our example when $k = 6$. For elements already has been accessed in L (e.g., `Arr[1:5]`), $n_{L,A}$ determines their liveness. In contrast, elements which are not accessed yet in L (e.g., `Arr[6:10]`), their liveness can be obtained from $f_{L,A}$. Liveness vector $l_{L,A,k}$ determines the minimal checkpoint range: by checkpointing only live elements (i.e., $A[r]$ for $l_{L,A,k}(r) = R$), checkpoint size can be minimized while preserving the correctness of the program.

On the other hand, the desired output of our optimization is slightly different. To minimize the overhead from procedure calls (e.g., DMA or memory copy), we find the smallest continuous range $[i, j]$, where $A[i : j]$ contains all live elements at the time of checkpointing (at iteration k). Let us say v^- and v^+ denote the smallest and the largest index of live elements in vector v. Then, we can derive that $i = l^-_{L,A,k}$ and $j = l^+_{L,A,k}$ as it is the minimal range that includes all live elements.

In the subsequent sections, we describe our method to find $l^-_{L,A,k}$ and $l^+_{L,A,k}$ from the given $f_{L,A}$ an $n_{L,A}$. Also, we use simplified notations for the remaining sections by omitting L and A from vectors (e.g., f instead of $f_{L,A}$) since the pair L and A are invariant in the calculation.

*1) **Simplifying f and n**:* Since we consider the accesses having a stride of 1 (Section III-C), most FAVs are represented in a *simple form* — consecutive accesses with optional Ns around: i.e., $\mathbf{N}^p\mathbf{X}^q\mathbf{N}^r$, where $p + q + r = n$ and $X = R$ or W.



(a) At iteration k, FAV can be divided into three ranges.

(b) computing liveness vector l_k by merging f and n.

Fig. 3: Building liveness vector from FAV and NAV.

We call such vectors *read FAV* and *write FAV* when $X = R$ and $X = W$, respectively. This form is the basis of our checkpoint range decision algorithm, and the compile-time only optimization can be enabled from its simplicity.

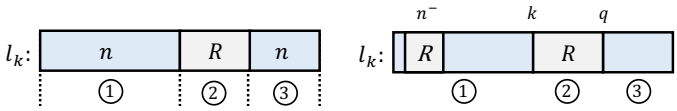
However, an exception may happen when the loop L has multiple instructions accessing to the array A. In this case, FAVs may not be in the simple form since the first access to each element can be from different instructions. If this is the case, we transform f into a simple form and update n accordingly, as follows.

First, we find *header access*, an instruction that precedes all other instructions (Section III-C), and build a new FAV as if it is the only access instruction in L. Let's say, for example, the new FAV is computed as $f' = \mathbf{N}^p\mathbf{W}^q$. While some liveness information can be lost (i.e., for $\mathbf{N}^p$) from this simplification, the loss is minimal since header access precedes all other instructions. For the lost ranges, we conservatively make them always checkpointed; to this end, we set the corresponding ranges in n to R (e.g., $n[0 : p] = R$). With this transformation, we can make all FAVs in a simple form while minimizing the impact on optimization performance.

*2) **Building Liveness Vector**:* With the simplified f and n, liveness vector l_k can be derived. Fig 3a shows an example of f at loop iteration k. In the figure, the index of the first and last accesses of f are denoted by p and q. Since f is in a simple form, it is trivial that the range $[k, q]$ is not accessed yet in L. Therefore, we can divide the ranges into three parts, the one not accessed yet (②: $[k, q]$) and the others (① and ③). As our example (Fig 2c) implies, l_k can be obtained by taking f (for to-be-accessed range ②) and substituting already accessed ranges (① and ③) of it to n. Fig 3b illustrates such process and the resulting l_k.

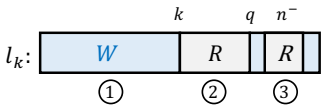
From the computed l_k, we can observe that l_k is obtained by systematically combining f and n. It implies that l^-_k and l^+_k, the goal of the analysis, may be computed by comparing f and n only. Indeed, l^-_k can be derived from a simple comparison depending on where n^- is in (among ①, ② and ③), without explicitly computing l_k (the same applies to l^+_k and n^+). As a final step, we discuss each case when f is read FAV and write FAV, and determine the final optimized range $[i, j]$.

*3) **Case of Read FAV**:* Fig 4a shows l_k when f consists of R. Let us first consider the case of l^-_k. We predict the value of l^-_k depending on where n^- belongs to. If n^- is in ① (Fig 4b), it is clear that n^- is the first live element of the vector, thus $l^-_k = n^-$. On the other hand, Fig 4c shows the case when n^- is in ③. As n^- is the *first* live element of n, any elements before



(a) Illustration of l_k when $X = R$.

(b) Case when n^- is in ①.

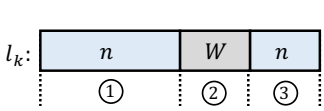


(c) Case when n^- is in ③.

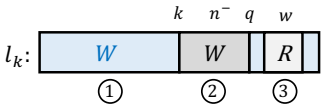
	①	②	③	
If n^- is in:	n^-	k	k	$\rightarrow l_k^-$
If n^+ is in:	f^+	f^+	n^+	$\rightarrow l_k^+$

(d) All possible cases of l_k^- and l_k^+, depending on k, f and n.

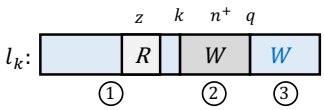
Fig. 4: Computing the checkpoint interval for Read FAV.



(a) Illustration of l_k when X=W.



(b) Case when n^- is in ②.



(c) Case when n^+ is in ②.

	①	②	③	
If n^- is in:	n^-	w	n^-	$\rightarrow l_k^-$
If n^+ is in:	n^+	z	n^+	$\rightarrow l_k^+$

(d) All possible cases of l_k^- and l_k^+, depending on k, f and n.

Fig. 5: Computing the checkpoint interval for Write FAV.

n^- cannot be R (denoted in the figure by filling ① with W). Therefore, k will be the first R in the vector, thus $l_k^- = k$. The same applies when n^- is in ②.

The same approach can be applied to evaluate l_k^+ (not illustrated). In this case, l_k^+ will be either q (i.e., n^+ is in ① or ②) or n^+ (in ③). As a result, l_k^- and l_k^+ can be statically determined in all possible cases. Fig 4d summarizes the result of the computations. In conclusion, the final interval $\mathbb{R}(L, A) = [i, j]$ is:

$$i = \begin{cases} n^- & (\text{if } n^- \leq k) \\ k & (\text{if } k < n^-) \end{cases}, \quad j = \begin{cases} q & (\text{if } n^+ \leq q) \\ n^+ & (\text{if } q < n^+) \end{cases}$$

and also can be further simplified as $i = \min(n^-, k)$ and $j = \max(q, n^+)$. Note that all values of n^-, q and n^+ are constant at the time of the analysis. Therefore, j is also constant and finding i requires at most one comparison at runtime.

*4) **Case of Write FAV**:* Fig 5a shows f in case of write FAV. We only depict the cases when n^- and n^+ are in ②, since calculating l_k^- and l_k^+ is straightforward in the other cases. Fig 5d presents the results when n^- and n^+ are in ① or ③.

Fig 5b illustrates the case where n^- is in ②, when calculating l_k^-. The range ① is filled with W since no elements are live before n^-. The resulting vector l_k implies that, the first live element should exist in ③, if any. We denote such index as w, which is the index of the first live element where $q < w$. It can be determined immediately since q and the vector n are already known.

Finally, Fig 5c shows the case where n^+ is in ②, when finding l_k^+. Similarly, the range ③ is marked as W. In this case, the last live element l_k^+ should be in the range ①. Let's denote such index z. Unlike the other cases, finding a fixed z at compile time is impossible, as k is not a constant, and n can be in an arbitrary form. In our implementation, we approximate z to k, which is always possible since $z \leq k$, to avoid run-time calculation at the cost of a slight loss of accuracy. As a result, the final interval $[i, j]$ is computed as follows.

$$i = \begin{cases} w & (\text{if } k \leq n^- < q) \\ n^- & (\text{otherwise}) \end{cases}, \quad j = \begin{cases} z & (\text{if } k \leq n^+ < q) \\ n^+ & (\text{otherwise}) \end{cases}$$

F. Optimizing Checkpoint of Scalar Stack Objects

Liveness-aware checkpointing can also be applied on scalar stack objects. LACT performs liveness analysis of stack objects based on backward traversal of CFGs. The analysis is done after register allocation pass since stack frame objects are not fixed before the register allocation as registers may spill.

To implement liveness aware checkpointing with minimal overhead, LACT estimates the access frequency of each object (considering the object size and block frequency) and reorders them so that frequently accessed objects are moved to top of stack. Before executing checkpoint, LACT adjusts the stack pointer to ignore dead objects near top of stack, using single `add` instruction. Once the checkpoint is finished, stack pointer is restored. While it limits the optimizable ranges on stack (i.e., consecutive objects from top of stack), it achieves the goal at fair trade-off in a highly lightweight manner.

IV. Evaluation

A. Experimental Setup

We extended FreeRTOS, a widely used real-time operating system, with support for checkpoint and recovery. Our experiments are conducted on a custom-built board featuring ARM Cortex M4 Core (STM32L496ZGTE) and 1 MB Ferroelectric RAM (FRAM). For evaluation in a real energy-harvesting environment (Section IV-C), we set up our environment based on TI BQ25570EVM energy harvesting board with supercapacitors and a programmable power supply.

We compare LACT with the stack management approach of DICE [3], [4], which is recognized as a state-of-the-art technique. Checkpoints are inserted statically at the headers of the loops as in [6]. We ported four benchmarks from miBench benchmark suite [22]. Three benchmarks having arrays (`sha`, `fft` and `stringsearch`) are the main targets of the evaluation. We include `basicmath`, which does not have an array, for evaluating optimization for scalar stack objects.

B. Checkpoint Size Reduction

Fig 6 shows the stack checkpoint sizes of DICE and LACT, compared to no-optimization case. The bars on the left show the checkpoint size of DICE, and the blue bars on the right present the cases when LACT is applied along with DICE. On average, LACT achieves an additional checkpoint size reduction of 37.8% over DICE, and the cumulative reduction is 52.5%. We observe that LACT is also effectively optimizes scalar stack objects (36.0% reduction on `basicmath`).

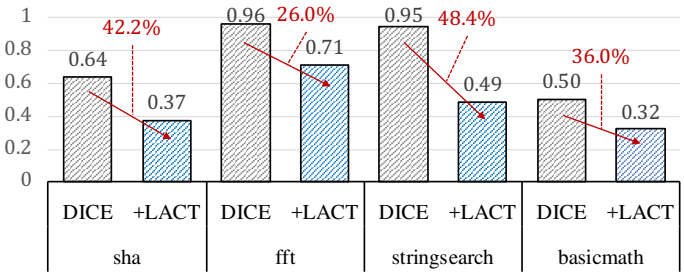


Fig. 6: Normalized stack checkpoint sizes of DICE and LACT.

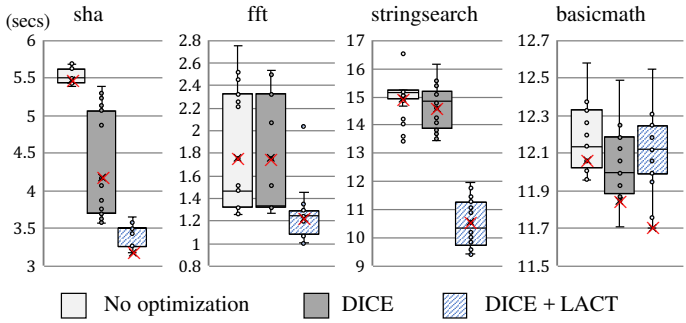


Fig. 7: Execution times measured from 50 samples under a real energy harvesting environment.

While DICE is effective especially for applications with deep call stack frames (e.g., `sha`), LACT shows more even improvements among the benchmarks. This is because most benchmarks have general patterns that LACT can optimize, such as array initialization and iterating array members. Also, some arrays are often used only until the limited points of the program and not used later. LACT can also optimize such cases by excluding the arrays from the checkpoint after their last use.

C. Performance on Energy Harvesting Environments

We evaluate LACT on a real energy harvesting environment, as described in Section IV-A. Supercapacitors equivalent to 118mF are used, and 20mW power is supplied to the energy harvesting board. We execute each benchmark 50 times under three cases (no optimization, DICE only, and LACT on top of DICE) and measure the execution times. Compared to the experimental designs which periodically inject power failures [9], [12], our setup is more realistic since the energy consumption of NVM accesses is also reflected in the result.

Fig 7 shows the distribution of the measured execution times along with the quartiles and the average (red X mark). Among the benchmarks having arrays, LACT shows an execution time reduction of 27.2% compared to DICE, on average. When DICE and LACT are applied simultaneously, they achieve a 33.8% reduction on the execution time (20.7% and 26.1%, respectively, when including `basicmath`).

Although the execution time is affected by various aspects other than stack checkpoint size (e.g., checkpointing TCB/registers, recovery mechanism, and hardware initialization), the checkpoint size still considerably impacts the execution time. This is because the impact of checkpoint size reduction is twofold: it not only directly contributes to finishing the program faster (by reducing the total number of instructions) but also makes the benchmark executes longer by avoiding energy-expensive NVM writes, thus consuming less energy.

V. Conclusion

This paper presents LACT, a compile-time optimization that reduces the checkpoint size of arrays in intermittent computing systems. LACT analyzes the liveness of array elements and minimizes the required checkpoint range by excluding dead elements from the checkpoint. Our evaluation shows that LACT can reduce the required stack checkpoint size by 37.8% on average, compared to the previous technique. In the experiment on a real energy harvesting environment, LACT can reduce the execution times of applications having arrays by 27.7%.

References

[1] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *RTAS*, pp. 40–54, IEEE, 2022.

[2] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *SenSys*, pp. 1–6, 2017.

[3] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Efficient intermittent computing with differential checkpointing," in *LCTES*, pp. 70–81, 2019.

[4] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Fast and energy-efficient state checkpointing for intermittent computing," *TECS*, vol. 19, no. 6, pp. 1–27, 2020.

[5] J. Kwak, H. Kim, and J. Cho, "Icer: An intermittent computing environment based on a run-time module for energy-harvesting iot devices with nvram," *Electronics*, vol. 10, no. 8, p. 879, 2021.

[6] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," in *ASPLOS*, 2011.

[7] F. A. Aouda, K. Marquet, and G. Salagnac, "Incremental checkpointing of program state to nvram for transiently-powered systems," in *ReCoSoC*, pp. 1–4, IEEE, 2014.

[8] N. Bhatti and L. Mottola, "Efficient state retention for transiently-powered embedded sensing," in *EWSN*, pp. 137–148, 2016.

[9] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *OSDI*, pp. 17–32, 2016.

[10] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, "Storage size reduction by in-place mapping of arrays," in *VMCAI*, Springer, 2002.

[11] A. Darte, A. Isoard, and T. Yuki, *Liveness Analysis in Explicitly-Parallel Programs*. PhD thesis, CNRS; Inria; ENS Lyon, 2016.

[12] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *ASPLOS*, pp. 85–99, 2020.

[13] N. A. Bhatti and L. Mottola, "Harvos: Efficient code instrumentation for transiently-powered embedded sensing," in *IPSN*, IEEE, 2017.

[14] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *ESL*, vol. 7, no. 1, 2014.

[15] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 35, no. 12, 2016.

[16] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *PLDI*, pp. 1101–1116, 2019.

[17] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *PLDI*, pp. 1005–1021, 2020.

[18] V. Kortbeek, S. Ghosh, J. Hester, S. Campanoni, and P. Pawełczak, "Wario: efficient code generation for intermittent computing," in *PLDI*, 2022.

[19] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," *SNAPL*, 2017.

[20] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, pp. 75–86, IEEE, 2004.

[21] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *ISSAC*, pp. 242–249, 1994.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC*, pp. 3–14, IEEE, 2001.