Efficient State Retention through Paged Memory Management for Reactive Transient Computing

Sivert T. Sliper[1], Domenico Balsamo[1], Nikos Nikoleris[2],
William Wang[2], Alex S. Weddell[1] and Geoff V. Merrett[1]
[1]{sts1u16, db2a12, asw, gvm}@ecs.soton.ac.uk; [2]{nikos.nikoleris, william.wang}@arm.com
[1]School of Electronics and Computer Science, University of Southampton, UK; [2]Arm Research

ABSTRACT

Reactive transient computing systems preserve computational progress despite frequent power failures by suspending (saving state to nonvolatile memory) when detecting a power failure, and restoring once power returns. Existing methods inefficiently save and restore all allocated memory. We propose lightweight memory management that applies the concept of paging to load pages only when needed, and save only modified pages. We then develop a model that maximises available execution time by dynamically adjusting the suspend and restore voltage thresholds. Experiments on an MSP430FR5994 microcontroller show that our method reduces state retention overheads by up to 86.9% and executes algorithms up to 5.3× faster than the state-of-the-art.

KEYWORDS

Transient Computing, Internet of Things, Batteryless Computing

1 INTRODUCTION

Ambient energy sources are theoretically infinite, offering devices powered by energy harvesting (EH) the potential for unbounded lifetime. However, EH is an unstable power source, so today's devices require rechargeable batteries or large supercapacitors to provide stability. Batteries and supercapacitors limit device lifetime and increase cost, size and environmental impact; avoiding them may therefore provide a more scalable IoT.

Transient computing (TC) is an emerging paradigm where computational progress is made despite a highly unstable power source; thus enabling battery-less devices that are powered directly from EH. Published works in software-based TC can be separated into two fundamentally different approaches: static and reactive TC [13]. Static TC inserts state-saving checkpoints at design/compile time [4, 12], or divides programs into small atomic tasks that are executed by a runtime [5, 8–10]. However, static TC suffers from wasted energy spent on code re-execution and superfluous checkpoints/task-transitions because optimal checkpoint placement or task decomposition is difficult [4, 6]. When power fails, any execution since the previous checkpoint/task-boundary has to be repeated in the next power cycle. Doing so not only wastes energy, but also risks inconsistencies between volatile and nonvolatile memory (VM and NVM), so static methods must take expensive steps to re-execute code safely [5, 9, 10]. *Alpaca*, a recent static TC approach, imposes runtime overheads of 1.3 − 3.6× when continuously powered [10].

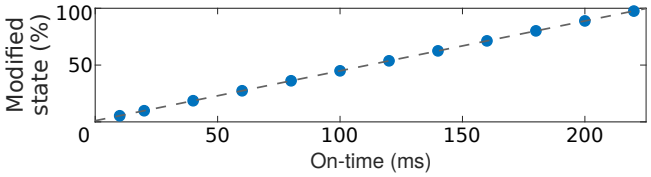

Figure 1: Percentage of allocated memory that is modified at the end of a power cycle in relation to on-time.

In contrast, reactive TC [1, 2, 7] uses interrupt-based supply voltage monitoring to trigger *Suspend* and *Restore* operations which maintain state persistency. The operations depend on hardware architecture, but are independent of the application. Only when a power failure is detected, *Suspend* saves a snapshot of CPU registers and VM to *NVM*, then halts execution (sleep). When power returns, the device sleeps until the supply voltage recovers beyond the restore threshold to guarantee sufficient energy for the next *Suspend*, then *Restores* state from a snapshot and resumes execution from where it was interrupted. Thus, code re-execution is eliminated, and superfluous save operations are minimised. Because of these advantages, reactive TC is the focus of this work.

Existing reactive TC methods [1, 2] suffer from inefficient state retention because the entirety of VM is saved and restored in every power cycle. Recent works proposed tracking dynamic memory to avoid backups of unallocated state [3, 15], henceforth termed *AllocatedState*. However, we observe that not all allocated memory is modified or even referenced during power cycles with short on-periods. Fig. 1 shows the result of an experiment measuring the percentage of modified allocated memory in relation to the on-time of a power cycle. The experiment was performed by executing 128-bit AES encryption on a 2 *kB* string on an MSP430FR5994 microcontroller [14] running at 8 MHz. Bhatti et al. [3] proposed comparing the current state with a saved snapshot to identify unmodified data, but this is only applicable for asymmetric memories such as flash, where reads are much cheaper than writes [15]. For FRAM, where reads and writes have the same energy cost, the method is counterproductive. An alternative to explicit state retention is to use NVM as main memory [7], but this degrades overall performance due to increased access energy [1, 14].

In this work, we propose ManagedState, a lightweight page-based memory manager that tracks active and modified regions of memory. We then develop a mathematical model that leverages

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

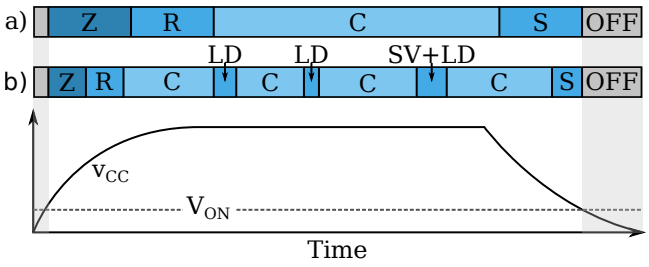

Figure 2: Conceptual comparison between (a) AllocatedState, and (b) ManagedState, in response to a power supply trace.

the memory manager to increase available execution time by calculating suspend and restore voltage thresholds at runtime. Based on device characteristics, the model also yields a constraint on the number of active and modified pages. Because violating this constraint may lead to a corrupt snapshot, the memory manager maintains a limit on the number of modified pages by writing back inactive modified pages when necessary.

A conceptual comparison between ManagedState and AllocatedState is shown in Fig. 2. Knowing precisely the active regions of memory speeds up restore (R), while knowing which regions are modified speeds up suspend (S). Improved suspend and restore performance also enables runtime threshold adjustments to defer suspend to the last possible moment and to wake up from sleep (Z) and restore as early as possible; thus maximising the time spent on useful computation (C). To enable this, ManagedState loads (LD) pages as needed and saves (SV) pages to NVM at runtime.

The contributions of this work are:

- ManagedState, a page-based memory manager for tracking active and modified regions of memory, resulting in 26.8 – 86.9% reduction in accumulated suspend and restore time, with minimal memory footprint (84 – 1102*B*) and minimal to moderate runtime overhead (1.05 – 1.48×) (Section 2).
- Runtime calculation of safe suspend/restore thresholds, based on the amount of active and modified memory, that eliminate code re-execution and corrupt snapshots, as well as increase available execution time. The combination of both contributions result in up to 5.3× faster application execution (Section 3).

2 MANAGEDSTATE: TRACKING AND LIMITING VOLATILE STATE

In order to load memory only when it is needed, and to avoid writing unmodified data when suspending, tracking of active and modified memory is necessary. While high-performance processors use hardware memory management units (MMUs) to translate and control access to memory [11], low-power microcontrollers suited for EH-powered applications do not have such features. Additionally, MMUs are designed for applications with megabytes to gigabytes of memory footprint as opposed to the kilobytes of a typical embedded application.

ManagedState is a light-weight memory manager that applies the well-known concept of paging to track accesses to volatile memory. In this work, it is implemented and evaluated with bare-metal applications, although it could also be used in conjunction with a typical embedded/IoT operating system. Like traditional paged memory, ManagedState loads pages only when they are referenced, and writes them back to main memory only if they are modified. However, ManagedState is much simpler: it does not perform address translations or relocations, as the entire application is assumed to fit in main memory.

To use ManagedState, the application calls *Acquire* before using a block of data with a pointer to the start of the data in main memory, the number of elements to be acquired and the reference mode (*RO*/*RW*, explained later in this section). ManagedState is then responsible for loading the relevant pages from NVM into main memory and for maintaining their persistence through power cycles. When the application no longer needs the block of data, it calls *Release* with a pointer to the data, and the number of elements.

When possible, it is most efficient to process blocks of data one page at a time; thus keeping only a single page active (or two, if an element crosses a page boundary) without excessive *Acquire*/*Release* calls. For non-linear *RW* access, where data residing in several different pages are accessed sporadically, the application must either acquire the entire block (minimal overhead, but keeps the entire block active) or acquire a few elements at a time (minimal active pages, but large overhead). A third option, to alleviate the overhead of non-linear *RW* memory accesses, is to restructure the application algorithm to improve locality, as demonstrated in Section 4.2.

Like AllocatedState, ManagedState saves and restores all CPU registers, the stack, and the `.data` section. Additionally, ManagedState provides a new section, `.mmdata` (memory managed data), where application data can be allocated for efficient state retention. `.mmdata` is divided into a set, P, of pages, p, where each page is of size $|p|$. Accesses to variables located in `.mmdata` are tracked to determine the following three sets at runtime.

- R: the **r**esident set of pages held in main memory
- M: the set of **m**odified pages
- A: the set of **a**ctive, i.e. currently referenced, pages

A reference consists of the operations *Acquire* and *Release*, shown in Algorithms 1 and 2; these two operations maintain R, M and A. There are two types of references:

- *RO* - **R**ead **O**nly reference
- *RW* - **R**ead **W**rite reference

An *RO* reference to a variable causes the corresponding page to be added to A. Similarly, an *RW* reference to a variable causes the corresponding page to be added to A and to M. If the page is not already in R, it is loaded into main memory and added to R. The number of references to each page is tracked. When the reference count to a page p is 0, it is removed from A.

When suspending, all pages that exist in M are saved to NVM. Pages which are in A, but not in M are *not saved*; their values are already guaranteed to be persistent. Pages that are in M, but not in A may be modified, but are not currently referenced; hence they are safely removed from M when they are saved to NVM. During startup, the volatile state is restored. With the proposed method, only active pages are loaded; that is, only pages in A. All other pages will be loaded only when they are referenced.

Algorithm 1 Acquire a variable residing in `.mmdata`.

```
1:  function ACQUIRE(pointer, ReferenceMode)
2:      p ← getPageNumber(pointer)
3:      if ReferenceMode = RW then
4:          while |M| ≥ M̂ do
5:              w ← nextPage ∈ LRU
6:              if w ∉ A then
7:                  saveNVM(w)
8:                  M ← M − {w}
9:          M ← p ∪ M
10:     if p ∉ R then
11:         load(p)
12:         R ← p ∪ R
13:     refCount[p] = refCount[p] + 1
14:     A ← p ∪ A
    return
```

Algorithm 2 Release a variable residing in `.mmdata`.

```
1:  function RELEASE(pointer)
2:      p ← getPageNumber(pointer)
3:      refCount[p] = refCount[p] − 1
4:      if refCount[p] = 0 then
5:          A ← A − {p}
    return
```

The pages in M comprise the volatile state of `.mmdata`. ManagedState maintains an upper limit $\hat{M}$ (determined in Section 3) on the number of modified pages $|M|$. When an *RW* access that would cause $\hat{M}$ to be exceeded occurs, a page needs to be removed from M. ManagedState finds a page in M that is not in A (a page that may have been modified, but is not currently active), saves it to NVM and removes it from M. If no such page can be found, $\hat{M}$ must be increased or the application's usage of *Acquire* and *Release* modified to reduce the number of concurrently active pages. A least-recently-used (LRU) table is maintained to avoid excessive page thrashing (repeated write-back of the same page). A page is never loaded more than once during a power cycle.

3 SUSPEND AND RESTORE THRESHOLDS

When the supply voltage drops below the suspend threshold, *suspend* is triggered and execution halted. Later, when supply recovers beyond the restore threshold, *restore* is triggered to resume execution. To maximise the time spent on useful computation, and to minimise superfluous state saving, *suspend* should be postponed as much as possible, as illustrated in Fig. 2. Hence the suspend threshold V_S should be minimised, while still ensuring sufficient energy for successful state saving. The restore threshold V_R protects against re-execution, so must guarantee that sufficient energy for suspend remains after restoring. For safety, V_R must be calculated under the assumption that no energy is supplied after restore begins. This minimum restore threshold is termed $V_{R,min}$. Practical systems should increase V_R beyond $V_{R,min}$ by a voltage Δv_C that ensures a minimum of computational progress in every power cycle. The optimal Δv_C depends on power supply characteristics

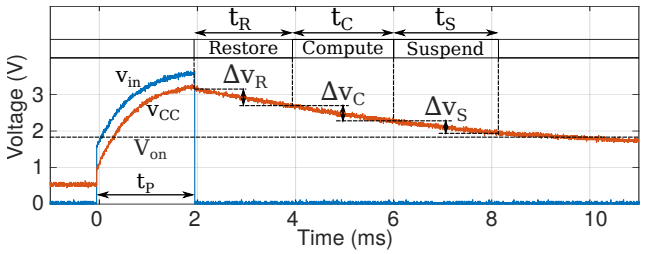

Figure 3: Safe restore and suspend thresholds, ensuring sufficient energy to guarantee a minimum of useful computation (t_C) and a successful snapshot.

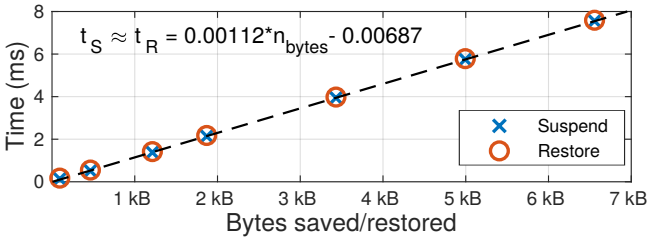

Figure 4: Execution time of suspend and restore operations.

and application constraints. For a low-current source, maximising Δv_C effectively amortises the cost of suspend and restore. In contrast, minimising Δv_C makes the device more responsive to events because of frequent wake-ups, but also less energy efficient because less computation is done per power cycle. For a sparse supply, where energy arrives in short pulses, Δv_C should be small to ensure that every pulse is utilised instead of being wasted through leakage and sleep currents.

Fig. 3 shows measured voltage traces and operation on a worst-case power pulse, which charges v_{cc} to V_R, then immediately drops to zero. As is typical for microcontrollers, the one used in this work draws nearly constant current (6% variation) over its operating voltage range because of its on-chip linear voltage regulator [14]. Current draw does, however, depend on peripherals, so the application's most power-hungry combination of simultaneously active peripherals should be assumed when calculating suspend/restore thresholds. Constant current draw makes the suspend, restore and compute voltage drops Δv_S, Δv_R and Δv_C, respectively, proportional to their respective execution times t_S, t_R and t_C. Thus, a linear model is suitable to calculate the threshold voltages

$$V_S = V_{on} + \Delta v_S = V_{on} + \frac{\Delta v_{CC}}{\Delta t} t_S, \tag{1}$$

$$V_R = V_S + \Delta v_C + \Delta v_R = V_S + \frac{\Delta v_{CC}}{\Delta t}(t_C + t_R). \tag{2}$$

The voltage drop $\frac{\Delta v_{CC}}{\Delta t}$ can be calculated from the platform's current draw I and capacitance C as

$$\frac{\Delta v_{CC}}{\Delta t} = \frac{I}{C}, \tag{3}$$

although measuring $\frac{\Delta v_{CC}}{\Delta t}$ directly is often simpler.

Furthermore, t_S and t_R are proportional to the amount of data to be saved/restored, as shown in Fig. 4, where the average execution time of suspend and restore were measured in relation to the amount of memory saved/restored. The dashed line shows the linear fit, confirming that

$$t_S(n_{bytes}) \approx t_R(n_{bytes}) = \alpha n_{bytes}, \tag{4}$$

where α is the time it takes to read/save one byte, and n_{bytes} is the number of bytes saved/restored, yields accurate estimation of t_S and t_R. The constant term shown in the figure is negligible for reasonable snapshot sizes ($> 10\ B$). Combining (1), (2) and (4) yields

$$V_S = V_{on} + \frac{\Delta v_{CC}}{\Delta t}(\alpha n_{ut} + \alpha|p||M|), \tag{5}$$

$$V_R = V_S + \frac{\Delta v_{CC}}{\Delta t}(t_C + \alpha n_{ut} + \alpha|p||A|), \tag{6}$$

where n_{ut} is the number of untracked bytes, which must be saved and restored in every power cycle. Hence $n_{ut} + |p||M|$ is the number of bytes written during suspend, and $n_{ut} + |p||A|$ is the number of bytes read during restore. The CPU registers, allocated stack, tables of ManagedState, and untracked application variables comprise the untracked memory. The restore threshold has an upper bound V_{max}, given by the maximum output voltage of the supply or the maximum operating voltage, constraining V_R to

$$V_R \leq V_{max}. \tag{7}$$

Substituting (5) and (7) into (6) and rearranging yields the constraint

$$|A| + |M| \leq \frac{V_{max} - V_{on}}{\frac{\Delta v_{CC}}{\Delta t}\alpha|p|} - \frac{t_C}{\alpha|p|} - \frac{2n_{ut}}{|p|}, \tag{8}$$

determining the maximum number of modified pages at runtime as

$$\hat{M} = floor\Big(\frac{V_{max} - V_{on}}{\frac{\Delta v_{CC}}{\Delta t}\alpha|p|} - \frac{t_C}{\alpha|p|} - \frac{2n_{ut}}{|p|} - |A|\Big). \tag{9}$$

During runtime, V_R and $\hat{M}$ are updated when a page is acquired or released, while V_S is updated when the application issues an *RW* reference and when saving modified pages during suspend. Calculating $\hat{M}$ at runtime maximises the capacity of M, minimising the number of pages saved during execution, while still eliminating corrupt snapshots and re-execution by guaranteeing the success of future suspends and restores.

4 EXPERIMENTAL VALIDATION

Memory tracking and runtime threshold calculation was experimentally validated on an MSP430FR5994-based development board. No energy storage beyond the platform's 10 μF decoupling capacitance was added. This small capacitance limits the amount of state that can be safely restored and suspended in an on-period. For large applications, this necessitates techniques such as this work to limit said state during runtime, as demonstrated in Section 4.3.

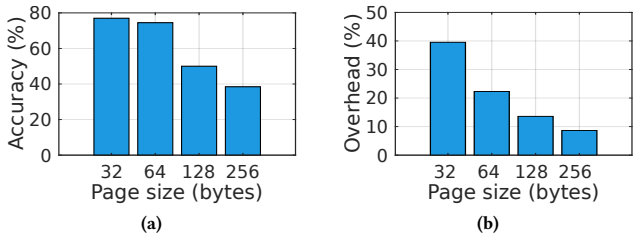

Figure 5: (a) Tracking accuracy and (b) suspend time overhead, in relation to page size $|p|$.

4.1 Benchmarks

Like hierarchical memory systems, ManagedState's performance depends mainly on the application's memory access patterns. Three benchmarks that reflect typical tasks for IoT sensing devices and exhibit different memory access behaviour were used for evaluation:

- **AES**: 128-bit AES encryption of a 2kB string. Typically used to secure communication.
- **CRC32**: 32-bit checksum generation. Typically used for error checking of communication payloads.
- **MATMUL**: Multiplication of two 25×25 matrices of 16-bit values. Representative workload for signal processing and classification tasks.

AES and CRC32 access memory linearly, hence ManagedState tracks their memory accesses efficiently. In contrast, MATMUL, accesses memory in a sparse and repetitive manner, leading to excessive *Acquire* and *Release* calls, hence large overhead. AES modifies the whole data buffer, while MATMUL only modifies the output buffer and CRC32 only modifies a single output variable.

4.2 Memory Tracking

Efficient and safe state retention through power cycles is the main aim of this work. Memory consistency was verified by taking a core dump (capturing memory and processor state using a debugger) immediately before suspending state, then power cycling the platform, then another core dump immediately after restoring the state. The allocated portion of the two core dumps were confirmed as identical, thus state is consistent through power cycles.

The page size $|p|$ affects tracking accuracy, suspend performance and memory overhead. To evaluate accuracy, the number of bytes saved during suspend by ManagedState (i.e. $n_{ut} + |p||M|$) while executing the AES benchmark were compared with the actual number of modified bytes since the previous snapshot. Tracking accuracy decreases with increased page size (Fig. 5a). Execution time overhead of suspend was measured and compared to the time taken to save the same amount of state without paging. Decreased page size leads to increased suspend overhead (Fig. 5b) because of the larger number of page attributes to maintain.

The static memory overhead of ManagedState consists of the LRU and attribute tables. The LRU table consists of $\hat{M}$ 1-byte page-number entries. The attribute table consists of $|P|$ 1-byte entries; each entry's *MSB* indicate whether the page is loaded (in R), the next bit indicates whether the page is modified (in M), and the

Table 1: Overhead of ManagedState on a continuous supply when compared to AllocatedState.

Benchmark	Runtime overhead	Memory overhead
AES	1.05×	86 *B* (3.5%)
CRC32	1.14×	86 *B* (3.5%)
MATMUL	6.81×	86 *B* (5.0%)
MATMUL_TILED	1.48×	102 *B* (2.5%)

remaining 6 bits constitute the reference counter ($A = \{p \in P : ReferenceCounter[p] > 0\}$). For the remainder of the experiments, $|p| = 128\ B$, balancing suspend overhead, memory overhead and tracking accuracy.

ManagedState's memory overhead and application execution time was compared to that of AllocatedState, results are shown in Table 1. The runtime overheads of AES and CRC32 are both small. MATMUL, however, has very large overhead due to poor spatial and temporal locality. There are two ways to alleviate the overhead of MATMUL. The first is to acquire all three matrices, perform the computation, then release; this alleviates the tracking overhead at the expense of restore and suspend performance. This solution yields nearly identical results to using AllocatedState, which also loads/saves the entire matrices at every restore/suspend. The second alternative is to improve locality by e.g. using a tiled implementation – a technique often used for performance improvement in systems with hierarchical memory. Computing MATMUL using 5×5 tiles, reduced the overhead to 1.48× (MATMUL_TILED in Table 1). The matrices were now acquired 3 tiles at a time (2 inputs, 1 output), maintaining superior suspend and restore performance when compared to AllocatedState, as evaluated in 4.3. For the remainder of the paper, MATMUL_TILED will be used for evaluation, due to the large overhead of the naive MATMUL implementation.

4.3 Suspend and Restore Time

ManagedState allows large applications to run intermittently. Fig. 6 shows the result of a comparison between the suspend and restore times (proportional to energy) of AllocatedState and ManagedState in relation to application size while running the AES benchmark. The size of the string was modified to vary the application size. For fair comparison, on-time was set higher than the application's run time. Both methods' suspend times grow linearly until ≈ 3 *kB*. AllocatedState can no longer find a safe restore threshold after 3.93 *kB* (shaded area on the figure), because the platform's capacitance cannot store sufficient energy for a safe restore-compute-suspend cycle. The dashed line shows projected suspend/restore time for AllocatedState, growing linearly with application size. ManagedState limits volatile state according to $\hat{M}$, thus keeping suspend time at a safe level regardless of application size.

ManagedState provides substantial improvement in suspend and restore performance. Fig. 7 shows measured accumulated suspend and restore time, $t_{S+R} = t_S + t_R$, in relation to on-time. The reduction in t_{S+R} when compared to AllocatedState (dashed line) is 26.8 − 86.9%, 86.7% and 49.9 − 65.3% for AES, CRC32 and MATMUL_TILED, respectively. For AES, t_{S+R} grows linearly until on-time approaches the run time of the application. For CRC32, t_{S+R} remains constant, because only a single page is active and only a single page is modified, as CRC32 accesses data linearly and only modifies a single output variable. MATMUL_TILED quickly reaches $\hat{M}$, after which ManagedState starts saving modified pages to NVM.

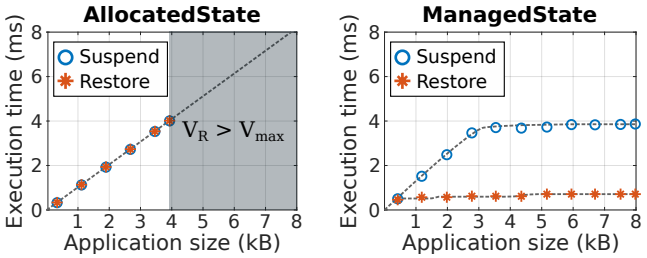

Figure 6: Suspend and restore time of ManagedState and AllocatedState in relation to application size.

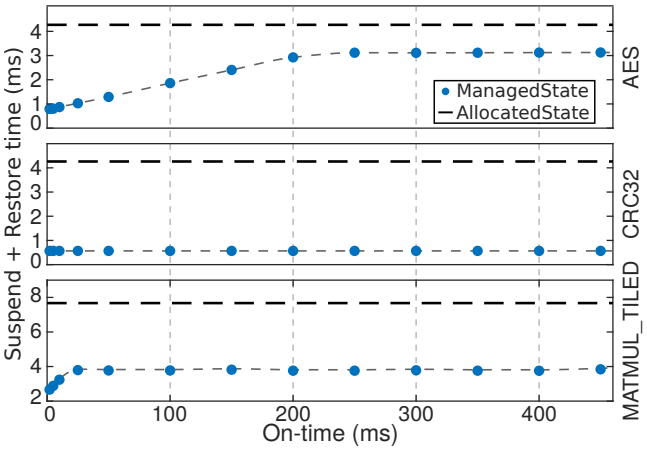

Figure 7: Accumulated suspend and restore time in relation to the on-time of a power cycle.

4.4 Suspend Threshold

Adjusting the suspend threshold at runtime assures that snapshots are saved successfully while maximising energy spent on application execution. The safety and accuracy of the suspend threshold was evaluated by using an oscilloscope to measure v_{CC} at the start and completion of the suspend operation, v_{start} and $v_{complete}$, respectively. A function generator provided 3.6 *V* square-wave pulses to power the platform which executed AES. The range of on-time spans from a short 10 *ms* pulse, modifying only a few pages, to a long 350 *ms* pulse, sufficient to finish the entire application. The results, shown in Fig. 8, show that the method adjusts the suspend threshold such that corrupt snapshots are avoided, i.e. $v_{complete} > V_{ON}$, while wasting minimal energy by keeping $v_{complete}$ close to V_{ON}. The variance of $v_{complete}$ increases with the amount of state saved (proportional to on-time), indicating imperfections in the constant-current model of Section 3. However, to minimise overhead, the computational complexity must be low.

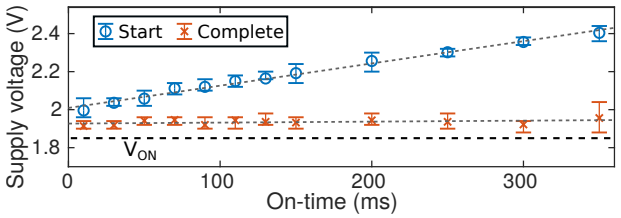

Figure 8: Supply voltage at the start and completion of suspend. Points show the average of 10 measurements, the bars show minimum and maximum measurements.

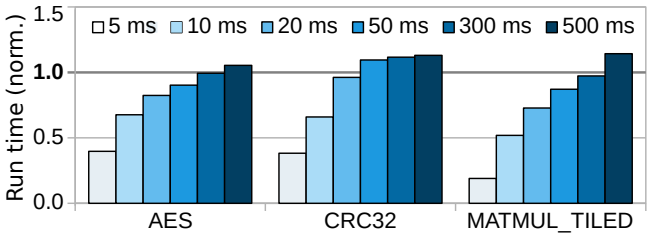

Figure 9: Benchmark run time relative to AllocatedState when powered by square-wave pulses.

4.5 Application Performance

Benchmark run times were compared against AllocatedState while powered by an intermittent supply, as shown in Fig. 9. For clear and repeatable results on an intermittent supply, the platform was powered by 3.6 V variable-width square-wave pulses from a function generator. A rectifying diode was used to prevent the function generator from discharging the platform during off-periods; thus v_{CC} discharges gradually through the platform's current draw. Operation is shown in Fig. 3, where t_P denotes pulse width (note that the figure shows a minimal pulse that barely suffices to charge v_{CC} to v_R). The duty cycle of the square-wave was adjusted such that the platform's capacitance was completely discharged (through leakage and sleep current) between the pulses. Using a real energy harvester for evaluation would make the results difficult to reason about, and hardly repeatable, because of the complex dynamics of specific harvesters in relation to their environment. Because our method's application performance depends largely on on-time, the square-wave results are readily transferable when considering our method for use with a specific energy harvester.

ManagedState is most effective when on-periods are short, completing the application up to 5.3× faster than AllocatedState. The extra execution time gained by restoring early and deferring suspend, as well as the improved suspend and restore performance, becomes less significant when on-time increases. Hence the run time of ManagedState approaches the overheads from Table 1 when on-time approaches an application's completion time.

5 CONCLUSIONS

Tracking memory references can alleviate the inefficiencies of existing state retention methods, reducing accumulated suspend and restore time by 26.8 − 86.9%, at a small cost in runtime overhead (1.05 − 1.14×) for applications with good locality. For applications with poor locality, tracking becomes expensive (6.81× overhead), but this may be alleviated by improving locality (1.48× overhead). Limiting the amount of state to be saved when power fails (M), allows larger application size (`.data` > $4kB$) without corrupt snapshots. Runtime calculation of suspend and restore thresholds capitalise on efficient state retention to improve energy efficiency, while still protecting against corrupt snapshots. Combining memory tracking using ManagedState, and runtime suspend and restore threshold calculations resulted in up to 5.3× faster algorithm execution time when on-time was short. In the future, this work may be integrated into a suitable operating system, allowing transient operation without the burden of inefficient state retention.

ACKNOWLEDGMENTS

Source code is available at https://git.soton.ac.uk/energy-driven. Experimental data associated with the paper is available at DOI: 10.5258/SOTON/D0835. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) under an iCASE award and Grant EP/P010164/1.

REFERENCES

[1] Domenico Balsamo, Alex S Weddell, Anup Das, Geoff V Merrett, Bashir M Al-hashimi, Davide Brunelli, and Luca Benini. 2016. Hibernus++ : A Self-Calibrating and Adaptive System for Intermittently-Powered Embedded Devices. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. – TCAD* 35, 12 (2016), 1968–1980.

[2] 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 Syst. Lett.* 7, 1 (2015), 15–18.

[3] Naveed Anwar Bhatti and Luca Mottola. 2016. Efficient State Retention for Transiently-powered Embedded Sensing. In *Proc. ACM Int. Conf. Embedded Wireless Systems Networks – EWSN*. 137–148.

[4] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proc. ACM/IEEE Int. Conf. Inf. Process. Sensor Netw. – IPSN*. Pittsburgh, PA, USA, 209–219.

[5] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Prog., Syst., Languages, Appl. – OOPSLA*, Vol. 51. Amsterdam, Netherlands, 514–530.

[6] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proc. IEEE/ACM Int. Conf. Compiler Construction – CC*. 116–127.

[7] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. 330–335.

[8] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2016. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in IoT Edge Devices. In *IEEE Int. Conf. VLSI Design*. 264–269.

[9] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proc. ACM SIGPLAN Conf. Programming Language Design Implementation – PLDI*. 575–585.

[10] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proc. ACM Programming Languages* 1 (2017).

[11] David A. Patterson and John L. Hennesey. 2016. *Computer Organization And Design* (1 ed.). Morgan Kaufmann.

[12] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. In *Proc. ACM Int. Conf. Architectural Support Programming Languages Operating Systems – ASPLOS*. Newport Beach, CA, USA, 159–170.

[13] Sivert T. Sliper, Domenico Balsamo, Alex S. Weddell, and Geoff V. Merrett. 2018. Enabling Intermittent Computing on High-performance Out-of-order Processors. In *Proceedings of the 6th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys '18)*. ACM, New York, NY, USA, 19–25.

[14] Texas Instruments. 2017. MSP430FR599x , MSP430FR596x Mixed-Signal Microcontrollers Datasheet. (2017).

[15] Theodoros D. Verykios, Domenico Balsamo, and Geoff V. Merrett. 2018. Selective policies for efficient state retention in transiently-powered embedded systems: Exploiting properties of NVM technologies. *Sustainable Computing: Informatics and Systems* (2018).