A Software Caching Runtime for Embedded NVRAM Systems

Harrison Williams
hrwill@vt.edu
Virginia Tech
USA

Matthew Hicks
mdhicks2@vt.edu
Virginia Tech
USA

Abstract

Increasingly sophisticated low-power microcontrollers are at the heart of millions of IoT and edge computing deployments, with developers pushing large-scale data collection, processing, and inference to end nodes. Advanced workloads on resource-constrained systems depend on emerging technologies to meet performance and lifetime demands. High-performance Non-Volatile RAMs (NVRAMs) are one such technology enabling a new class of systems previously made impossible by memory limitations, including ultra-low-power designs using program state non-volatility and sensing systems storing and processing large blocks of data.

Unfortunately, existing NVRAM significantly underperforms SRAM's access latency/energy cost and flash's read performance—condemning systems dependent on NVRAM to pay a steep energy and time penalty for software execution. We observe that this performance penalty stems predominately from instruction fetches into NVRAM, which represent >75% of memory accesses in typical embedded software. To eliminate this performance bottleneck, we propose *SwapRAM*, a new operating model for NVRAM-based platforms which repurposes underutilized SRAM as an instruction cache, maximizing the proportion of accesses directed towards higher-performance SRAM. *SwapRAM* consists of a set of compile-time code transformations and a runtime management system that transparently and dynamically copies code into SRAM throughout execution, with an extensible logic to delay eviction of hot code. Across nine embedded benchmarks running on a real FRAM platform, *SwapRAM*'s software-based design increases execution speed by up to 46% (average 26%) and reduces energy consumption by up to 36% (average 24%) compared to a baseline system using the existing hardware cache.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0391-1/24/04.
https://doi.org/10.1145/3622781.3674183

ACM Reference Format:
Harrison Williams and Matthew Hicks. 2024. A Software Caching Runtime for Embedded NVRAM Systems. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3622781.3674183

1 Introduction

Ultra-low-power embedded systems bring intelligence and connectivity to domains traditionally beyond the reach of computing devices due to power, size, or cost constraints. Shrinking device sizes combined with increasingly powerful, high-resolution sensors enable new applications and information streams—including deeply deployed image capture and recognition [7, 15], environmental/urban monitoring [8, 21, 26], and implantable/wearable devices [9, 10]—but also dramatically increase the data handling workload relative to available computational and energy resources. State-of-the-art mobile/Internet-of-Things deployments depend on novel techniques at the architecture, device, and protocol levels to meet performance and lifetime requirements.

Emerging Non-Volatile RAMs (NVRAMs), which blend the non-volatility and density of traditional flash memory with the read/write performance of Static RAM (SRAM), are one such innovation underpinning new approaches to low-power edge computing. High-density, byte-addressable NVRAMs enable long-lived sensing deployments recording bulk data on-chip, where such deployments were previously limited by flash's high write energy and low program/erase endurance [22, 48]. Several candidate technologies—including Spin-Torque Transfer Magnetoresistive RAM, Resistive RAM, and Ferroelectric RAM (FRAM)—demonstrate promising performance [11, 35]; many of today's systems depend on FRAM, which is commercially available as a flash replacement on a fully integrated system-on-chip [40]. Research using commercial FRAM-based systems demonstrates the value of unified NVRAM storing both code and data for 1) its non-volatility enabling ultra-low-power hibernation modes disabling volatile memory [5, 24] and 2) its density advantage over SRAM (approximately 1 transistor per bit versus 6 for SRAM [23, 39]) supporting more sophisticated and memory-intensive applications on low-power devices [19, 28, 49].

Unfortunately, current deployments of existing NVRAMs underperform more mature SRAM and flash memories in several important ways that limit the performance of NVRAM-based systems. Despite potential theoretical improvements, recently-commercialized embedded FRAM sacrifices read speed and energy versus similar flash memories [42, 45]. FRAM reads and writes are also both slower and more energy-intensive than similar SRAM accesses. As a result, FRAM currently falls short as a general-purpose, drop-in replacement for *either* memory technology due to its slower access rate when fetching instructions and limited interface width when accessing data. Thus, designers of ultra-low size, weight, and power devices must either accept the lower performance of current NVRAMs, or increase system size, cost, and complexity by switching to a larger device with sufficient SRAM—lest they forgo the application entirely.

Platform designers mitigate NVRAM's underperformance by allocating an on-chip buffer to use as a read cache for code execution [42]. While this buffer reduces pressure on the NVRAM, it forces a permanent tradeoff: SRAM allocated to the NVRAM cache cannot be used as main memory, and main memory cannot be used as an NVRAM cache. Several factors, including hardware overhead and the increasing memory demands of emerging applications, dictate that hardware designers designate the vast majority of SRAM as program memory to maximize the resources available to the programmer and use comparatively little for caching (e.g., a 32 byte cache as part of the memory controller for a 128KiB FRAM [42]). Systems using FRAM as unified memory realize a limited improvement from existing caches, as CPU accesses to disjoint code and data addresses cause cache contention (Section 2.2)—failing to address the performance penalty for a primary use case of NVRAM devices.

We observe that SRAM *underutilization* in current NVRAM systems, combined with insights about the memory access patterns typical of embedded software, enables techniques to mitigate poor access characteristics and bring the common-case performance of NVRAM systems into line with those using mature non-volatile memory technologies while maintaining the unique advantages of NVRAM. Unified-memory systems avoid the size limitations of SRAM by mapping program data into NVRAM, leaving small but performant SRAM free for reuse. On today's general-purpose low-power devices, however, users cannot change the hardware cache size and must instead implement such a cache in software. A software approach to caching has a new set of tradeoffs: increasing complexity increases runtime overhead, as the processor spends cycles managing the cache instead of executing application code. At the same time, shifting cache design from processor-design-time to software-compile-time allows software specific information sources to improve caching performance (such as typical access patterns and control flow analysis).

We build on prior work developing software-based instruction caches designed to overcome the limitations of code storage in Dynamic RAM (DRAM) for mid- to high-end embedded systems [33]. This approach embeds fine-grain (i.e., basic-block-level) instrumentation throughout the program to precisely predict instruction fetches and minimize the number of time-intensive DRAM accesses, while bookkeeping to track the block-level cache overlaps with DRAM access stalls—effectively masking the delay introduced by additional software effort maintaining the cache. However, today's highly integrated low-power systems present a different set of constraints that make them a poor fit for existing work. The NVRAM used to store code is integrated directly onto the processor die, dramatically reducing the time penalty for a cache miss (e.g., to 3 cycles on our evaluation platform) compared to discrete DRAM chips and exposing the latency introduced by fine-grain software caching. SRAM main memory—which provides lower power accesses and runs at full CPU speed with no cache—is scarce owing to its low density combined with system size constraints, limiting the amount of cache metadata a system can maintain in SRAM without starving itself of space for application code storage. These new tradeoffs and challenges give rise to the central question this paper addresses: **can an efficient and effective instruction cache for resource-constrained low-power systems be implemented purely in software?**

We answer this question affirmatively by designing, implementing, and evaluating *SwapRAM*, a lightweight code transformation toolchain and runtime that maximizes code execution out of SRAM by transferring instructions into SRAM "just-in-time" during execution. *SwapRAM*'s code transformation component modifies application code at the assembly level, rendering functions runtime-relocatable to facilitate execution out of arbitrary addresses in SRAM. *SwapRAM*'s runtime interposes on calls to uncached functions, copying code into SRAM and transferring control flow to the newly cached version. The runtime maintains metadata describing cache usage and each cached function to prioritize candidates for eviction, minimizing thrashing while maximizing the time applications spend executing out of SRAM. *SwapRAM*'s runtime makes flexible, intelligent caching decisions by predicting program behavior based on static and dynamic code analysis, extending the temporal/spatial locality concepts underlying hardware caches across the semantic gap.

We implement and evaluate *SwapRAM* on a Commercial Off-The-Shelf (COTS) low-power FRAM-based microcontroller [43], executing nine embedded benchmarks from the MiBench2 suite [18, 20]; we measure energy consumption and total runtime for each benchmark to quantify *SwapRAM*'s end-to-end impact on real systems' performance.

We compare *SwapRAM* both to baseline systems executing code out of FRAM and a prior software-based instruction cache originally targeting systems using external (e.g., DRAM) code storage [33]; our results indicate that *SwapRAM* eliminates on average 65% of FRAM accesses by efficiently shifting execution to SRAM, resulting in an average execution speed improvement of 26% and energy reduction of 24% across benchmarks compared to baseline execution out of FRAM using the existing hardware cache. *SwapRAM* realizes this significant performance improvement on resource-constrained platforms where prior software-based caching approaches either degrade performance or fail to run at all, owing to its lightweight code instrumentation system and runtime. This paper makes the following technical contributions:

- We quantify memory access patterns in low-power COTS microcontrollers and find that the majority of accesses are to code space (Section 2), translating to a significant performance penalty when code is in stored in NVRAM operating at a lower frequency than the digital core or volatile data memory.
- We design and implement *SwapRAM*, a software system implementing an intelligent instruction cache using on-chip SRAM (Section 3). *SwapRAM* eliminates the bottleneck of NVRAM accesses during common-case code execution, enabling systems to run at their highest performance and most efficient operating points.
- We evaluate *SwapRAM* on a commercial NVRAM-based system running nine real-world benchmarks; our results indicate that *SwapRAM* significantly reduces energy consumption while increasing execution speed, outperforming both FRAM-based execution and earlier software caching approaches with a programmer-transparent software update (Section 5).
- We open-source our design/evaluation data for replication and to enable developers to explore *SwapRAM* for deployed systems.

2 Background and Related Work

2.1 Embedded Memory and Software Organization

Typical low-power edge devices employ a flattened memory model dividing the address space into two distinct sections, with little to no cache hierarchy [31, 32, 34]. This division is largely due to historical technical limitations on the memory available in embedded systems. Such systems store code and other non-volatile data in either mask ROM or flash memory, which both offer low cost, high density, and high read performance at the cost of poor write performance. Intermediate program state and results are stored in volatile Static RAM (SRAM), which offers the highest access performance at the cost of non-zero static power draw and relatively low density (DRAM's high static power due to its need for refresh cycles precludes its use in energy-constrained systems). Software treats non-volatile memory sections as functionally read-only during normal operation; while many devices include the necessary circuitry for in-system flash writes for initial programming or software updates, writing to flash is a slow, energy-intensive, and endurance-limited process rendering it unsuitable for use as main memory [48].

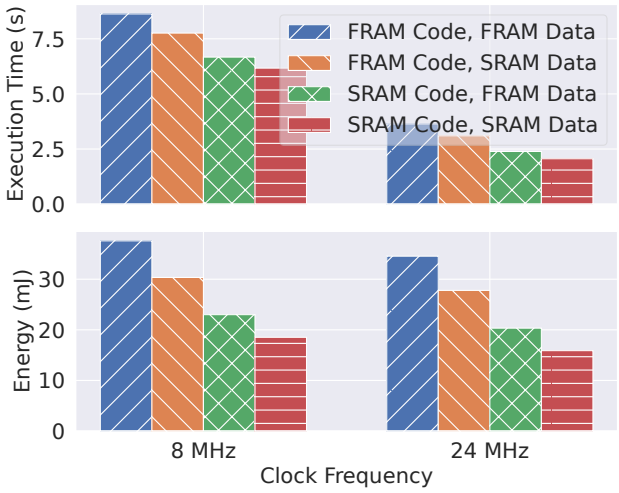

Figure 1. Unified memory approaches either limit performance (FRAM) or software complexity (SRAM) and—counter to convention—when forced to separate code and data, code is best placed in SRAM due to instruction access frequency and FRAM's slower access rate.

The success of emerging Non-Volatile RAM (NVRAM) technologies has given rise to systems which eliminate the division between non-volatile ROM and volatile RAM [11, 35, 40]. COTS microcontrollers using embedded Ferroelectric RAM (FRAM) as a drop-in replacement for flash memory are available at price and size points competitive with flash-based systems [42] and enable deployments using FRAM's blend of density, non-volatility, and access performance. Long-lived and deeply deployed systems depend on FRAM for low-power, high volume data capture or memory-intensive operations (e.g., on-chip inference [4, 16]) previously limited by flash's energy and endurance constraints [15, 22].

2.2 NVRAM as Unified Memory

Beyond data storage, new software approaches demonstrate the value of using FRAM as a universal memory storing both code and program data [30, 46]. Mapping program state into FRAM allows systems to disable the volatile SRAM array and minimize static leakage, crucial for low-duty-cycle battery-powered applications which spend most of their time hibernating [3] or batteryless systems which must sustain state across power failures [5, 19, 28, 29, 49]. Finally, FRAM's

density advantage over SRAM dramatically increases the amount of main memory available to software while maintaining a small footprint: one representative device contains only 2KiB of SRAM, but 128KiB of FRAM [42] which designers can split arbitrarily according to their needs in a unified-memory approach—allowing deployment of more complex or memory-intensive software on ultra-low-power devices.

Unfortunately, FRAM's current performance means systems pay a steep cost for its unique flexibility during common-case execution, as FRAM accesses are slower and more energy-intensive than similar SRAM accesses or flash reads. Commercial microcontroller-embedded FRAM operates at a maximum access frequency of 8MHz, despite the cores surrounding it running at up to 24 MHz [43]—forcing chip designers to artificially reduce effective clock frequency using wait states when accessing FRAM. Even at clock frequencies below FRAM's maximum, current FRAM-based systems consume over twice the power of comparable flash devices during program execution [43, 47]. For these reasons, commercial FRAM-based devices include a read cache to limit the number of wait states incurred while executing from FRAM [43].

Using FRAM as unified memory to store both code and data aggravates energy and runtime penalties by increasing the number of FRAM accesses and reduces the hit rate of hardware caches supporting the FRAM, owing to the poor locality of alternating accesses to code/data space. We quantify the performance of different memory allocation approaches by measuring the runtime and energy consumption of an arithmetic benchmark on an FRAM-based MSP430FR2355 (Section 4 describes our setup in more detail) while varying the physical location of code and data. We illustrate the results in Figure 1. Unified-memory operation significantly underperforms the "standard" configuration of storing code in FRAM and data in SRAM, even at 8 MHz—the frequency at which the CPU can access FRAM with zero wait states—because a single instruction execution can dispatch multiple simultaneous accesses to distant addresses in FRAM, bottlenecking memory accesses at the cache. Configurations which store at least one of code or data in SRAM reduce cache contention and eliminate wait states; placing and executing code in SRAM and data in FRAM further improves performance by reducing FRAM reads because most memory accesses are to code space (Section 2.4). Placing both code and data in SRAM maximizes performance, but is impractical as most deployments cannot fit both code and data in SRAM. The shortcomings associated with unified-memory operation prevent developers from taking advantage of FRAM as main memory due to the heavy time and energy penalty, and significantly degrade the performance of systems dependent on embedded NVRAM for memory-intensive processing—disqualifying NVRAM-based systems from energy- and performance-constrained applications they are otherwise uniquely suited for.

Research addressing NVRAM's applications in energy-constrained systems focuses largely on its use in batteryless devices, where FRAM is the only commercially available option to reliably sustain execution across power failures [5, 24, 25, 30]. Hardware approaches to mitigate NVRAM underperformance range from simple write-through caches [27] to multi-level, runtime-reconfigurable caches that write the minimal amount of data to NVRAM to maintain correctness across power cycles [13]. While hardware techniques provide a powerful tool for mitigating NVRAM underperformance, they require architectural modifications and compete with other hardware components for space and power on already resource-constrained systems.

2.3 Software Caching Techniques

Other work proposes compile-time approaches to reduce slow-memory pressure and improve performance on existing systems. Scratchpad-based approaches target systems with a fast, secondary "scratchpad" memory and insert code to copy hot data or small, frequently-executed code loops into the scratchpad [37, 38]. Static approaches demonstrate the potential of offloading typically non-volatile data to faster, volatile memory, but 1) depend on the programmer to adapt to context- or data-dependent execution paths that change the energy cost or access pattern of a given workload and 2) lack the ability to dynamically load code or data into SRAM at runtime, limiting the realized benefit when SRAM is scarce. As embedded programs grow in both complexity and size, a new approach is needed to maximize performance on embedded NVRAM systems.

One general-purpose approach is presented by Miller et al. [33] as an alternative to hardware-based caches for embedded systems which store code in external DRAM. They instrument each basic block in application code at compile time to redirect control-flow instructions to a caching runtime, which loads the subsequent basic block into SRAM and rewrites the original control-flow instruction to point to the now-cached block. Cache management overhead—selecting a location in SRAM, updating metadata, and rewriting the old instruction—is masked by the latency of external DRAM accesses. We evaluate this approach alongside *SwapRAM* and show that the highly-integrated and resource-constrained systems we target are a poor fit for the *basic-block caching* technique owing to its high memory and runtime overhead (Section 5).

2.4 Memory Access Patterns

Typical low-power embedded systems execute code directly from flash and store program data entirely in on-chip SRAM. While flash-based systems can realize a performance improvement by copying code to and executing it from SRAM [45], code in SRAM competes with volatile data which

Benchmark	Binary Size (B)	RAM Usage (B)	Code/Data Access Ratio
Stringsearch (STR)	12232	7586	1.620
Dijkstra (DIJ)	21956	8324	4.679
CRC	1470	562	3.448
RC4	3724	4444	1.944
FFT	23014	4768	3.749
AES	9608	674	3.947
LZFX	11085	10794	2.656
Bitcount (BIT)	4344	720	2.740
RSA	6331	332	2.530
Average Access Ratio			3.035

Table 1. Proportion of accesses to code and data space for embedded benchmarks we evaluate *SwapRAM* on.

cannot be remapped anywhere else—limiting the amount of already-scarce SRAM available for code storage. Embedded NVRAMs eliminate this restriction: software can map both code and volatile data to relatively-plentiful NVRAM, freeing *all* of the smaller but higher-performance SRAM to handle "hot" memory accesses. This paper demonstrates how to use this insight to 1) *eliminate the performance bottleneck inherent to current NVRAMs by shifting the maximum number of accesses to SRAM* and 2) do so entirely in software to address existing and near-future hardware-constrained systems.

SRAM's scarcity means that systems must be selective in choosing what to offload from lower-performance NVRAM. What kinds of memory accesses dominate in embedded software? We port nine benchmarks from the MiBench2 embedded benchmark suite [18, 20] (the subset that fits on our evaluation platform, Section 4) to the MSP430, a low-power architecture available on COTS NVRAM systems [43]. We then run each benchmark in a modified version of the open-source mspdebug [6] simulator tracking all memory accesses, and categorize each access as one to code space or data space.

Table 1 describes the proportion of code to data accesses for the benchmarks we evaluate. In every case, software accesses code space significantly more often than data space—on average over 3× more frequently, and in some cases up to nearly 5× as often. The dominance of code accesses is a consequence of a register-based processor architecture: software stores commonly-used values in architectural registers to minimize memory accesses, but each instruction fetch requires an access to code memory (or the instruction cache when one exists). Most instructions are register-to-register operations that only access memory to fetch the instruction. Our MSP430-based results likely *underestimate* the proportion of instruction fetches on the memory bus for microcontrollers in general: MSP430 instructions can operate on memory directly, while many architectures only access memory through dedicated load/store instructions [2, 31].

The energy/performance overhead of instruction supply is one motivating factor for novel architectures (e.g., vector-dataflow execution [17]), but remains an obstacle for current low-power systems. In this paper, we extend the concept of multi-use memory—previously introduced by work which repurposes cache as directly addressable memory [12, 14]—to existing systems by using SRAM main memory as an instruction cache. We also build on prior work which developed software caches for DRAM-based systems [33] where the inclusion of a hardware-based cache is deemed prohibitively design- or area-intensive. We demonstrate how a software caching approach tailored to low-power, highly-integrated systems can eliminate the majority of NVM accesses (an average reduction of 65% in our experiments) by caching code in higher-performance on-chip SRAM, maximizing performance on current and near-future NVRAM-based devices.

3 Design

At its core, *SwapRAM* implements an instruction cache in software. *SwapRAM* exploits the same temporal and spatial locality that motivates hardware-based caches, predicting future accesses in order to amortize the cost of data movement. However, a software-level implementation exposes new opportunities and limitations demanding different design choices to maximize performance improvement. Three high-level observations inform our design:

1. **Software cache loading is expensive:** Loading instructions into SRAM through software is a heavyweight process. While a hardware cache uses dedicated circuits to quickly populate fast memory on demand, a software approach must sequentially redirect control flow to a copy function, copy code into SRAM, update metadata, and redirect execution to the SRAM code copy. *SwapRAM*'s high performance depends on the insights below to maximize hit rate and amortize overhead over many SRAM executions.
2. **Rich application-level information provides a unique source of direction for caching decisions:** Hardware approaches achieve generality and performance by exploiting fine-grain, address-level locality; *SwapRAM* additionally exploits high-level trends based on semantic information, program analysis, and insights about the C runtime[1] not readily available at the architectural level. In particular, we demonstrate how function-level locality and call-stack monitoring allow *SwapRAM* to predict memory accesses and accurately pre-fetch code for execution far in the future.
3. **A flattened memory architecture enables flexibility in memory accesses:** Application code can fetch instructions from either NVM or SRAM: code in a flat memory architecture does not need to be cached in order for the processor to read and execute it. *SwapRAM* can deliberately avoid caching rarely accessed code when doing so would evict more frequently used code,

[1] *SwapRAM* targets C-based systems, which dominate the embedded software ecosystem today.

when pre-existing hardware caches would effectively handle incoming execution, or when the transfer overhead outweighs the savings from SRAM execution.

3.1 *SwapRAM* Overview and Interface

SwapRAM consists of two central components: a compile-time, assembly-level pass and a run-time module. Figure 2 shows an overview of *SwapRAM*'s compile-time pass and runtime behavior. *SwapRAM*'s static pass patches application code to render it safely runtime-relocatable and determines the size of code blocks to be cached (Section 3.2). The *SwapRAM* runtime interposes on regular execution and redirects calls to uncached functions to a cache miss handler, which:

1. Decides whether and where to cache the called function based on cache state.
2. Determines which, if any, currently cached functions need to be evicted to clear space.
3. Copies the target function into SRAM.
4. Updates metadata and redirection information (Section 3.2) for the newly cached and evicted functions.
5. Branches to the newly cached function.

SwapRAM typically requires no application code modifications (with the exception of jump tables, see Section 4) and is programmer-transparent, although we include a blacklist option to exclude specific functions from caching when they have strict timing requirements or are known to execute infrequently. Otherwise, users only need to integrate *SwapRAM*'s instrumentation scripts and runtime library into the compile toolchain.

3.2 Dynamic Function Redirection

Hardware caches normally operate on fixed-size blocks set at processor design time by the size of a cache line. *SwapRAM*'s software-based design shifts block size from a hardware-design-time decision to a software-run-time one, allowing program-specific and flexible block sizes. However, a software-level implementation also means *SwapRAM* cannot directly examine memory requests and must instead embed code throughout the application to predict which instructions will be executed. Maximizing end-to-end performance using software instrumentation is a balancing act: inserting more code increases execution overhead, but also allows *SwapRAM* to more accurately predict program flow.

The simplest way to predict execution is to instrument basic blocks: straight-line code sequences terminated with a control flow decision. Unfortunately, the size and number of basic blocks in typical programs (thousands of blocks with 5-6 instructions per block [18]) means block-level instrumentation dramatically increases both code size and execution overhead (an intuition we evaluate as part of Section 5).

To better balance instrumentation overhead and caching accuracy, *SwapRAM* caches code at the granularity of functions. Function-level instrumentation offers several advantages that make it effective for *SwapRAM*: first, function calls are relatively infrequent (compared to basic blocks) but still effectively indicate execution path. By caching an entire function on call, *SwapRAM* captures temporally and spatially adjacent code accesses (i.e., to the instructions in the function) without overloading execution with instrumentation. Second, software can modify function call sites to point to different functions during execution. *SwapRAM* uses this to insert calls to its instrumentation and caching code without increasing instruction count, and to remove calls to *SwapRAM* code once a function is cached—reducing *SwapRAM*'s impact on common-case execution once a function is cached.

Figure 3 illustrates how *SwapRAM* modifies function calls to support dynamically relocating code into SRAM. *SwapRAM* inserts code at the assembly level and compiles the modified application into an intermediate binary to determine final function size and account for changes made by linker optimizations (Section 4). *SwapRAM* replaces all calls—which are typically calls to absolute addresses in the original binary—with indirect call instructions to addresses stored in designated memory locations. On startup, these addresses point to *SwapRAM*'s cache miss handler rather than the original function. At each call site, *SwapRAM* also inserts an instruction to write a function-specific `funcId` value to a global memory location to signal to the cache miss handler the function to be cached. The first time program flow reaches the call instruction, *SwapRAM*'s miss handler is invoked.

3.3 Cache Miss Handler

Figure 4 illustrates the low-level behavior of the cache miss handler, which copies functions into SRAM, updates relocation data, and evicts functions as necessary. The simplest case is for a small function and an empty cache. Because *SwapRAM* replaces call instructions at the assembly level and is oblivious of function signatures, the first step is to save registers containing function arguments as defined by the platform calling convention. *SwapRAM*'s miss handler then uses the `funcId` value and a lookup table generated by the static pass to determine the size and NVM address of the target function. *SwapRAM* uses this information along with the current cache state to determine where to place the function (for an empty cache, *SwapRAM* begins with the lowest SRAM address). Finally, *SwapRAM* performs the copy, updates the call address to point to the SRAM copy, restores argument registers, and branches to the SRAM function copy. Later calls to the same function point directly to the cached version, bypassing the miss handler and incurring only the overhead of the added call-site instructions.

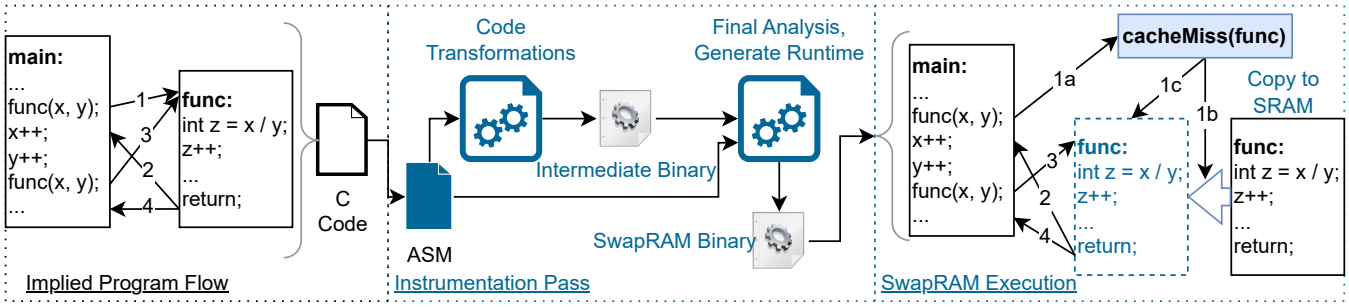

Figure 2. Overview of *SwapRAM*'s code generation, instrumentation, and basic execution model. Numbers indicate order of edges taken during program execution.

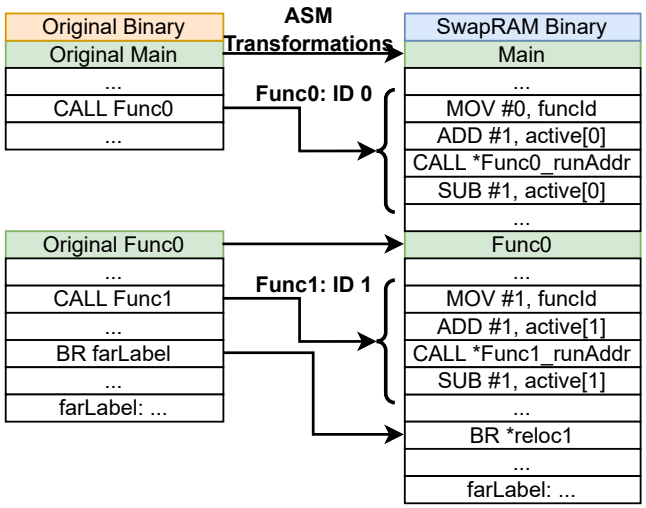

Figure 3. Assembly-level changes made by *SwapRAM*'s static instrumentation pass.

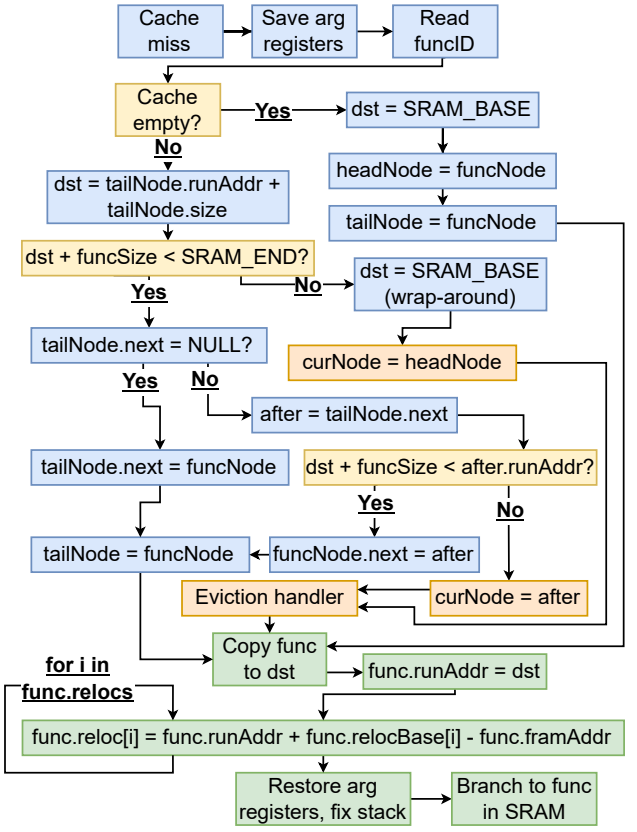

Figure 4. *SwapRAM* runtime component invoked upon cache miss for a circular-queue cache structure.

3.3.1 Branch Relocation.

Dynamic code relocation introduces several challenges for *SwapRAM* to keep execution correct and performant; one complicating factor is PC-modifying instructions. Embedded architectures typically modify control flow using both PC-relative and absolute branches:

- **PC-relative branches** encode an offset to add to the current PC and are *position-independent*, i.e., they point to the correct instruction regardless of the absolute address of the entire function. Relative branches work as intended when *SwapRAM* moves functions, but have a limited range because such instructions are traditionally used for conditional branching and spend part of the instruction word specifying a condition.
- **Absolute branches** have a longer range but set the PC to a fixed address regardless of the instruction's location. Without modification, executing an SRAM copy of an absolute branch causes program flow to branch back into NVM and incur the performance penalty of executing from slower NVRAM.

SwapRAM renders absolute branches relocatable using a value stored in memory rather than an instruction-encoded immediate (Figure 3). Instead of an absolute address, *SwapRAM* represents each branch target as an offset $ofs = branchTarget - fnBase$ from the base address of the

function containing it. *SwapRAM*'s static pass associates each absolute branch with a separate `reloc` memory entry and adds this offset to the function's base address in SRAM during a cache miss (Figure 4). When execution reaches the cached version of the branch instruction, the PC receives the previously calculated `reloc` value pointing to the cached version of the branch target—keeping execution in higher-performance SRAM.

3.3.2 Function Eviction. Eventually, SRAM will reach capacity and *SwapRAM* will need to clear space for a newly-called function. *SwapRAM* evicts functions only when attempting to cache a new one; after deciding where in SRAM to place the new function (Section 3.4), *SwapRAM*'s runtime steps through the metadata and flags for evicting each function that overlaps with the memory range for the incoming function to be cached. Flagging a function for eviction does not guarantee that it will actually be evicted (Section 3.3.3)—if all flagged functions *can* be evicted, *SwapRAM* loops through the flagged functions again and finishes the eviction process. Function eviction consists of resetting the function's metadata: namely, updating the cache data structure, pointing the function redirection entry back to the cache miss handler, and resetting the branch relocation entries (which must be reset in case *SwapRAM* later executes the function out of NVM, see Section 3.3.3).

3.3.3 Call Stack Integrity. Function eviction requires special care when cached code calls other functions. Subsequent calls to different functions will be redirected to *SwapRAM*'s miss handler if the new function is not cached, which may in turn evict the original calling code from the cache. If *SwapRAM* evicts and overwrites code on the call stack, however, execution will crash after the PC returns to the original call site that no longer contains the expected code.

We ensure call stack integrity using a per-function `active` counter. *SwapRAM* inserts instructions to increment/decrement a function's active counter[2] before/after it is called (shown in Figure 3) and does not evict functions with non-zero active counters during execution. If during a cache miss *SwapRAM* attempts to evict an active function, *SwapRAM* aborts the caching operation and executes the function that triggered the miss from NVRAM. This approach guarantees correctness at a typically low runtime cost but in principle gives rise to the *pathological* case of a function repeatedly failing to evict its own caller, reducing performance when *SwapRAM* invokes the miss handler each time.

3.4 Cache Memory Structure

SwapRAM's software implementation enables flexibility in how and where functions are cached in SRAM. Two challenges inform *SwapRAM*'s memory organization: first, caching different-sized functions causes fragmentation if *SwapRAM* introduces gaps between each function space—wasting already-scarce SRAM. Second, *SwapRAM* must organize functions in the cache so as to avoid evicting soon-to-be-executed code or attempting to evict code on the call stack. *SwapRAM* must balance these demands with complexity to minimize runtime overhead.

We represent each function using a node containing the relevant metadata for that function: the function's address in NVRAM, the current address to execute from, size, and active

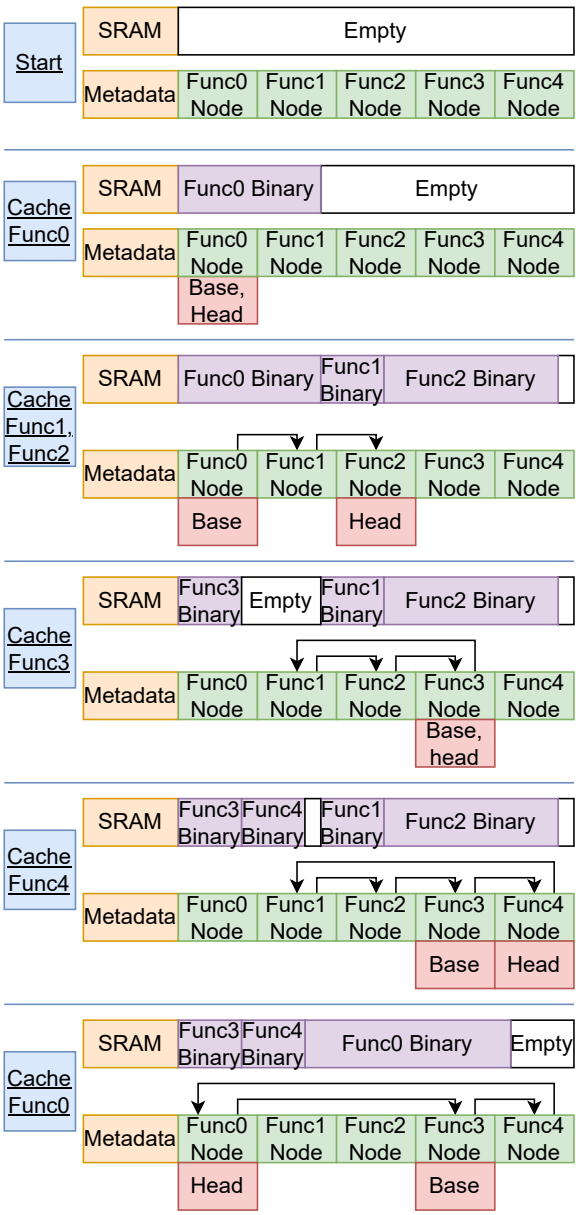

Figure 5. Function memory layout in SRAM throughout execution using a circular queue.

[2]Using a counter rather than a binary flag allows *SwapRAM* to support recursive programming where one function may have multiple stack frames.

Listing 1. Original

```
  CMP R12, R13
  JGE block2
block1:
  MOV #1, R14
  ...
block2:
  MOV #0, R14
  ...
```

Listing 2. Transformed

```
  CMP R12, R13
  JL label1
  BRA block2
label1:
  BRA block1
block1:
  MOV #1, R14
  ...
block2:
  MOV #0, R14
  ...
```

Figure 6. Transformations for block-level caching and extending the range of conditional CFIs on the MSP430.

counter. Organizing these nodes, and their corresponding SRAM function copies, into a certain data structure defines how *SwapRAM* addresses the challenges described above. This data structure effectively sets *SwapRAM*'s replacement policy. For example, storing functions and their nodes as a stack maximizes memory efficiency by placing functions at contiguous addresses in SRAM but compromises on eviction behavior by forcing a "most-recently-cached" replacement policy (which is counterproductive when considering both temporal locality and that recently-cached code is likely to be on the call stack and therefore be un-evictable).

Our proof-of-concept design uses a circular queue to balance data density and eviction behavior. Figure 5 illustrates how *SwapRAM* stores and evicts functions in SRAM throughout execution. Compared to a stack, a queue sacrifices a small degree of data density—allowing empty space both at the end of SRAM and following the head of the queue—for improved eviction behavior. A queue's first-in-first-out structure results in a "least-recently cached" replacement policy, better aligning with the temporal locality of code accesses and reducing the chance that functions attempt to evict their ancestors. A circular queue supports high data density and efficient eviction behavior with a relatively simple implementation; more sophisticated data structures would enable more intelligent caching decisions (e.g., a cost function to discourage evicting large functions using a priority queue), although we leave this exploration for future work.

4 Implementation

We develop *SwapRAM* as a set of Python scripts both for code instrumentation/transformation and for generating *SwapRAM*'s C runtime code. *SwapRAM*'s static instrumentation script takes unmodified assembly code and processes it in two passes. The first pass applies the code transformations, inserts *SwapRAM*'s instrumentation instructions, and compiles an intermediate binary. The second pass uses the intermediate binary to determine function sizes and base offsets for each branch relocation, generating the runtime and finally producing a standalone binary ready for deployment.

Target Platform: Our *SwapRAM* implementation targets the Texas Instruments MSP430FR line of devices, commercial microcontrollers with embedded FRAM NVM [34]. The MSP430 is a scalar, in-order 16-bit architecture optimized for low power draw commonly found in mobile systems and lightweight sensing and processing tasks [1, 26, 36]. We first evaluate *SwapRAM* on a version of the open-source mspdebug simulator [6] modified to record memory accesses and CPU cycle counts in order to track fine-grain execution statistics not readily available on physical platforms. We then evaluate *SwapRAM*'s real-world performance improvement using a physical MSP430FR2355 [41, 43] with 32 KiB of FRAM and 4 KiB of SRAM. The MSP430FR2355 runs up to a maximum CPU clock frequency of 24 MHz with 8 MHz FRAM, which we use to explore how baseline FRAM-based execution and *SwapRAM* scale with clock frequency.

SwapRAM is architecture-agnostic, except for code to preserve calling convention behavior: our MSP430 platform passes arguments in registers R12-R15 [44], so the miss handler preserves these registers. The MSP430-GCC linker attempts to implement all branches as PC-relative, which execute faster than absolute branches; branches outside of the -511/+512 word PC-relative range are reverted to absolute branches. *SwapRAM* iteratively scans the intermediate (post-instrumentation) binary for absolute branches to insert relocation code when generating the final binary, as *SwapRAM*'s code may push branches outside of the PC-relative range.

Benchmarks: We evaluate *SwapRAM* using MiBench2 [18, 20], a collection of embedded benchmarks representing workloads for resource-constrained systems. We select the subset of nine MiBench2 programs that fits on our evaluation platform, detailed in Table 1. Eight of the nine benchmarks require no source code modification; however, the bitcount benchmark uses a jump table to select from multiple functions for counting set bits in a number. Because *SwapRAM*'s instrumentation pass requires the destination of each call instruction at compile time, we replace the jump table call with a switch statement to call the correct function using the original table index. Future iterations of *SwapRAM* could include an interface for the programmer to explicitly inform the runtime of "dynamic" function calls, although we do not explore this option here.

Baseline Systems: Our baseline implementation executes code from FRAM using the integrated hardware cache to reduce FRAM reads (2-way associative with four 8-byte cache lines [43]). Five of the nine benchmarks require more program memory than our MSP430 has available SRAM. We first run all benchmarks mapping program memory space

into FRAM using the unified memory model NVRAM technologies enable (Section 5.4). We then re-evaluate the four benchmarks that can fit program memory into SRAM to explore how *SwapRAM* performs when data and code coexist in SRAM (Section 5.5). We compile all benchmarks using the msp430-gcc toolchain [44], running each benchmark 10 times to capture common-case performance after *SwapRAM* populates the cache.

We also port the block-based caching technique introduced by Miller et al. [33] to our MSP430 platform for comparison against *SwapRAM*. This approach splits the SRAM into evenly-sized cache slots and caches application code at the granularity of basic blocks rather than functions as in *SwapRAM*. Every Control-Flow Instruction (CFI) terminating a basic block initially points to a runtime miss handler (to identify target blocks, each CFI is associated with a unique entry point into the runtime); when execution reaches the miss handler, it places the target basic block into a cache slot, stores the location of the cached block in a hash table, and branches to the cached block. To reduce branches into the runtime, the runtime *chains* flow between blocks by overwriting the initial CFI to point to the newly-cached target block (analogous to our call redirection approach in Section 3.2).

For brevity, we only evaluate here the highest-performance implementation detailed in the original paper: maximizing chains between cached blocks and flushing the cache when it is full to eliminate the need for bookkeeping to undo chains. We experimented with the physical placement of the metadata and runtime code for managing the block cache and found that keeping them in FRAM maximized performance. Placing them in the SRAM (as in the original design) caused a high degree of thrashing and subsequent performance loss by reducing the memory available for caching application code. We compare *SwapRAM* to this *best-effort* port of the block cache which reserves the entire SRAM for application code caching.

We instrument application code for block caching at the assembly level similar to our *SwapRAM* implementation, with additional passes to identify basic blocks and modify CFIs accordingly. Because conditional branches on the MSP430 are limited to the -511/+512 PC-relative range and thus cannot span the range of the SRAM, we replace each conditional CFI with an absolute branch to the original destination preceded by an instruction to skip that branch if the *opposite* condition is true (Figure 6 gives an example). We also insert branches at the end of basic blocks when execution would normally fall through to the subsequent block, as blocks are not guaranteed to remain contiguous in the cache. For the hash table we follow the original implementation of a 0.5 load factor and use the *djb2* hash [50] as it uses exclusively shift/add operations, native instructions to the MSP430.

Library Instrumentation: Many embedded programs use precompiled library functions from binaries generated as part of the toolchain rather than the application code (e.g., floating-point math functions), but *SwapRAM*'s assembly-level instrumentation pass cannot operate on compiled binaries. To include these functions in *SwapRAM*'s (and the baseline block cache's) runtime as candidates for caching, we combine the objdump utility available as part of msp430-gcc with a script to generate gcc-parsable MSP430 assembly code corresponding to each library function and integrate that assembly into the *SwapRAM* workflow as with normal source code. While disassembly loses some semantic information, the information *SwapRAM* needs—intra-function branch destinations and function boundaries—can easily be recovered programmatically. Large libraries benefit from the function blacklist interface described in Section 3.1 to exclude un- or infrequently-used code from caching, as *SwapRAM* must reserve space in the metadata array for each potentially-cached function (Section 5.2).

5 Evaluation

We evaluate *SwapRAM* in two stages to explore its effect on system behavior and performance across a diverse set of embedded benchmarks. Our simulation-based evaluation captures how *SwapRAM* affects low-level behavior—memory access patterns and overall cycle count—while our deployment on a physical FRAM platform allows us to compare real execution time and energy consumption between systems with and without *SwapRAM*. The results of this evaluation serve to answer the following key questions:

1. How effectively does *SwapRAM* reduce NVRAM pressure by dynamically shifting instruction fetches into smaller but higher-performance SRAM?
2. To what degree does *SwapRAM* significantly improve end-to-end performance by amortizing transfer overhead over SRAM execution?
3. Does *SwapRAM* represent an improvement over prior software-based cache approaches?

5.1 *SwapRAM* Maintains Program Flow

SwapRAM interposes on and redirects control flow to execute code from SRAM, but should not otherwise modify program behavior. We validate that *SwapRAM* maintains semantically correct execution by running each benchmark with random sequences of input data, comparing the output and final program memory state of each benchmark on the baseline and with *SwapRAM*. Each benchmark we use to evaluate *SwapRAM* contains a check-sequence to print the results of the benchmark (e.g., a checksum for the CRC benchmark), which we print over the on-chip UART to a desktop computer. We compare the output of each benchmark running on the MSP430FR2355 using both the baseline system and

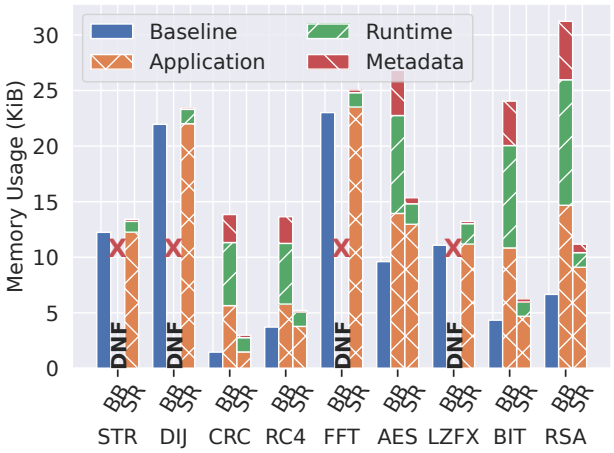

Figure 7. Block-based (BB) and *SwapRAM* (SR) NVM usage, showing contribution of transformed application code and system components.

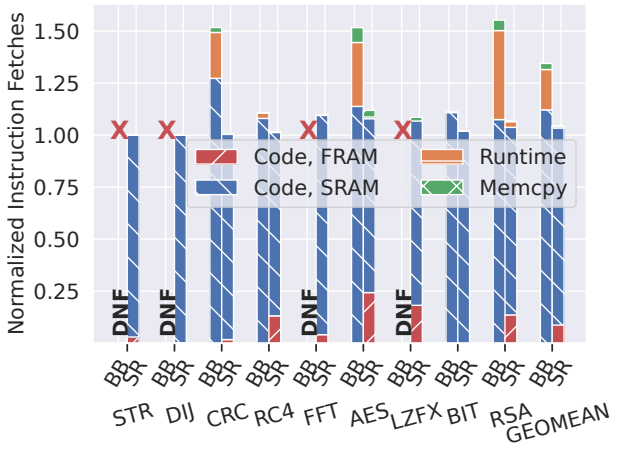

Figure 8. Dynamic instruction breakdown for each benchmark execution using *SwapRAM* and block-based caching, normalized to unified-memory operation.

SwapRAM, validating that *SwapRAM* does not affect the behavior of typical embedded programs. We remove the print statements and disconnect the serial bus when measuring benchmark energy and runtime.

5.2 Binary Size and Memory Usage

Transforming application code and adding a runtime system are only practical if the resulting binary still fits on the target platform, an important consideration for the resource-constrained low-power devices we target. Instrumentation inlined into the application code is particularly harmful as it increases the pressure on the SRAM cache when the added instructions must also be cached. We quantify the effect *SwapRAM* and the block-based approach have on overall binary size in Figure 7, which illustrates the size increase introduced by both systems. The Application and Runtime bars illustrate the executable binary size for transformed user code and runtime management code, respectively, while the Metadata bars illustrate the size of the structures needed to track the state of each cache system. We omit the size of the application's data area (Table 1) from Figure 7 for clarity as it does not change with either caching system. Block-based caching scales poorly with application size: four of our nine benchmarks do not fit on the evaluation platform after running the transformation/instrumentation passes, and we mark these benchmarks as DNF (Does Not Fit).[3] Across the remaining benchmarks, the block-based cache increases total NVM usage by an average 368%. Several sources contribute to the overall increase: adding instructions for each basic block to branch to the runtime and other cached blocks (Figure 6) on average approximately doubles application size, while the metadata and runtime code combined consume a similar amount of memory. While designers could trade off performance and memory consumption by reducing the size of the hash table, the dominant factor in the remaining memory consumption is the jump table used to inform the runtime of the target block during a cache miss.

SwapRAM's coarser-grain, function-level instrumentation results in an overall average binary size increase of 27% across all benchmarks. *SwapRAM*'s effect on application code size depends mainly on the number of calls to instrumented functions and ranges from a 0.1% to a 37% increase. In many cases, the addition of the cache miss handler dominates *SwapRAM*'s binary size increase. Miss handler size varies with the total number of relocatable branches (Section 3.3.1) because we insert code to calculate each relocation upon caching or eviction, but eviction logic makes up the majority of the function. The total handler size ranges from 972 to 1844 bytes, with an average across our benchmarks of 1378 bytes.

5.3 *SwapRAM* Eliminates NVRAM Accesses

SwapRAM achieves its performance improvement by shifting code accesses that would normally be issued to NVRAM into SRAM. We use the simulator described in Section 4 to determine how well *SwapRAM* reduces NVRAM pressure and detail the results in Table 2. *SwapRAM* eliminates on average 65% of NVRAM accesses by shifting instruction reads into SRAM—indicating that *SwapRAM*'s approach of using SRAM primarily for code execution successfully reduces NVRAM pressure.

[3]We exclude these benchmarks from average calculations for the block cache throughout our evaluation.

Benchmark		STR	DIJ	CRC	RC4	FFT	AES	LZFX	BIT	RSA	Geo. Mean Δ
FRAM Accesses (10^6)	Baseline	1224.4	275.9	0.6	8.8	5766.6	23.2	31.8	3306.1	0.7	
	Block-based	DNF	DNF	0.4 (-33%)	4.4 (-50%)	DNF	20.8 (-10%)	DNF	1304.7 (-61%)	0.8 (+7%)	-34%
	SwapRAM	498.4 (-59%)	52.6 (-81%)	0.2 (-75%)	3.8 (-57%)	1969.4 (-66%)	13.9 (-40%)	11.0 (-65%)	946.6 (-71%)	0.4 (-50%)	-65%
CPU Cycles (10^6)	Baseline	1033.5	388.3	0.7	8.0	6924.5	29.8	31.8	4173.7	0.9	
	Block-based	DNF	DNF	1.4 (+100%)	11.2 (+40%)	DNF	46.5 (+56%)	DNF	5244.3 (+20%)	1.4 (+52%)	+52%
	SwapRAM	1033.5 (+0.0%)	389.1 (+0.2%)	0.7 (+0.2%)	8.0 (+0.0%)	7538.2 (+8.9%)	37.0 (+24%)	35.3 (+11%)	4367.0 (+4.6%)	1.1 (+16%)	+6.9%

Table 2. NVRAM access and software cycle counts for *SwapRAM* and block-based caching on our simulation platform. *SwapRAM* significantly reduces FRAM accesses in exchange for a marginal increase in total (unstalled) cycles, while block-based caching has a diminished effect on FRAM pressure and significantly increases software effort.

Table 2 also shows the cycle count increase for each benchmark execution due to *SwapRAM*'s instrumentation and caching code.[4] *SwapRAM*'s cycle overhead scales with 1) the number of calls an application makes to instrumented functions due to *SwapRAM*'s code transformations and 2) the cycles spent executing runtime code and copying instructions to SRAM. While *SwapRAM* increases the cycles to completion for each application—an average 6.9% increase, and in the worst case 24% for the `AES` benchmark—the performance improvement from executing application code out of SRAM outweighs *SwapRAM*'s software overhead (Section 5.4).

SwapRAM yields a significant improvement over the block-based approach, which reduces FRAM accesses by an average 34%. While the block-based cache is designed never to execute application code from FRAM, block-level instrumentation causes a large number of both execution branches into the runtime and accesses to the metadata (which we store in FRAM to maximize end-to-end performance, Section 4)—limiting the overall reduction in FRAM pressure. The result is that the block-based approach ultimately underperforms the baseline by overloading execution with fine-grain instrumentation, increasing the unstalled cycles to completion by an average 52% over the baseline.

Finally, we extend the simulator to further break down program execution using *SwapRAM* and the basic-block cache. Figure 8 illustrates the source of each instruction fetched during each benchmark, separated into application code fetched from FRAM or SRAM, each system's miss handler, and memcpy calls to place functions in SRAM. Block caching entirely avoids code execution out of FRAM at the cost of significantly increasing the dynamic instruction count—an average 36% increase[5]—owing both to frequent jumps into the runtime and transformations that add instructions to each basic block in the application (Figure 6). *SwapRAM*'s relatively lightweight inline instrumentation increases dynamic instruction count by 0-10% for each benchmark, while the contribution from *SwapRAM*'s runtime is less than 3% for all benchmarks. Because we expect *SwapRAM*'s cache miss handler to be called relatively infrequently, we always execute both it and memcpy from FRAM.

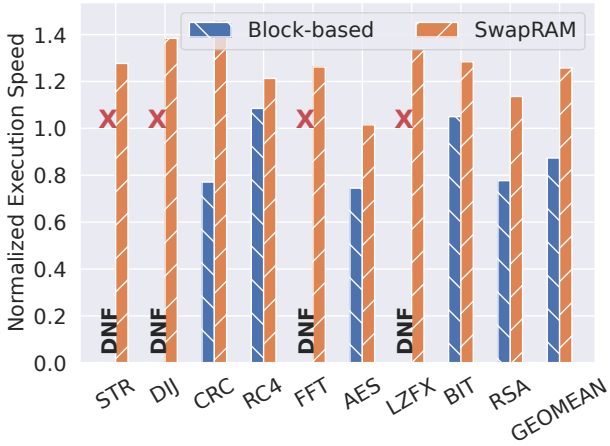

Figure 9. Benchmark execution speed at 24 MHz for *SwapRAM* and block-based caching, normalized to unified-memory operation. *SwapRAM* significantly increases execution speed compared to both the baseline and block-based approaches.

5.4 *SwapRAM* Energy and Runtime Performance

To determine how *SwapRAM*'s execution model affects performance on a real system, we evaluate each benchmark on an MSP430FR2355 development board [41]. We isolate the MSP430 using the on-board jumpers and power the device

[4] Our simulation platform does not account for the added cycles from FRAM wait states, which are captured in Section 5.4 as part of overall execution speed; the results in Table 2 represent unstalled cycles.

[5] Increases in dynamic instruction count and cycle count differ because different MSP430 instructions execute in different numbers of cycles.

through a sense resistor, across which we measure voltage and current using an oscilloscope. Each benchmark toggles a digital pin during execution, which we monitor on the oscilloscope to trigger power measurements.

We evaluate each system at a clock frequency of 8 MHz—the highest frequency at which the CPU can access FRAM without wait states—and 24 MHz, the maximum clock frequency and the most energy-efficient operating point for the digital core [43]. Figure 9 shows *SwapRAM*'s and the block cache's performance normalized to baseline unified-memory operation at 24 MHz. The CPU inserts a three-cycle delay for each FRAM access when operating at 24 MHz and depends on the cache hardware (Section 4) to access code/-data without stalls. These FRAM wait states and access cost significantly limit overall execution speed and efficiency.

Both *SwapRAM* and the block cache circumvent these wait states by executing application code directly from SRAM. While the block-based cache realizes a marginal improvement on two benchmarks—`RC4` and `bitcount`—the increased execution overhead associated with caching at the block granularity in general outweighs any performance improvement for the block-based cache, degrading execution speed compared to the unified-memory baseline by an average 13% while increasing energy consumption by 12%. *SwapRAM* largely eliminates instrumentation and runtime overhead by caching at the function level and taking advantage of program semantics, improving effective execution speed by an average 26% while consuming 24% less overall energy. *SwapRAM* realizes a similar but slightly smaller improvement at 8 MHz, increasing execution speed by an average 13% and reducing energy consumption by 20%, while the block-based cache's performance further degrades to 21% slower and 19% higher-energy than the baseline.

While all other benchmarks realize a 13-46% execution speed improvement and 16-36% reduction in total energy consumption, the `AES` benchmark is an outlier: *SwapRAM* only reduces total energy consumption by between 0.5% and 3.6%, and actually reduces execution speed by 6.4% at 8 MHz. Table 2 and Figure 8 explain *SwapRAM*'s relative underperformance on `AES`. *SwapRAM* increases the cycles to completion for `AES` by 24.1%, over twice that of the next-highest overhead benchmark (`LZFX`)—and executes the lowest proportion of application code out of SRAM (77.5%). The large increase in cycles/instructions executed indicates that `AES`'s function call patterns cause thrashing as *SwapRAM* repeatedly evicts and re-caches code, while the proportion of application code executed from FRAM indicates that *SwapRAM* often fails to evict active functions and must use the fallback strategy of FRAM execution. While *SwapRAM* significantly improves most benchmarks we evaluate, our results for the `AES` benchmark suggest strategies to detect and reduce thrashing (e.g., by temporarily pausing eviction to

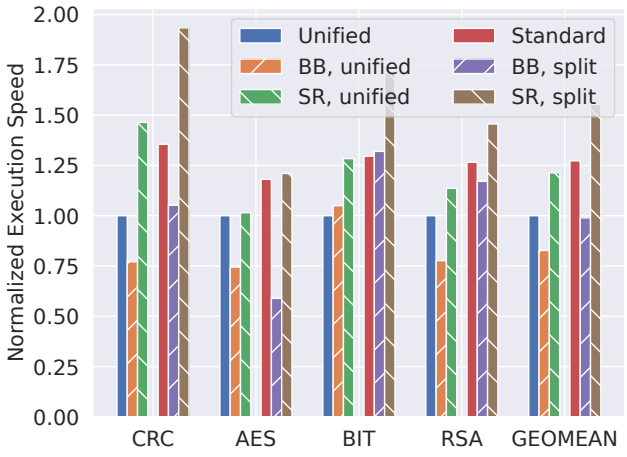

Figure 10. Execution speed at 24 MHz across unified- and split-memory benchmarks, normalized to baseline unified memory operation.

"freeze" cache state) or more aggressive code transformations to reduce software cache contention are compelling directions for future work.

SwapRAM improves performance at 24 MHz by removing expensive FRAM accesses from the execution path, but also realizes a similar improvement even at 8 MHz: the frequency at which FRAM's access speed no longer limits execution. Although SRAM's lower access energy partially explains the energy reduction, the main benefit of software caching at lower operating frequencies is eliminating hardware cache contention (Section 2.2). While unified memory execution forces the hardware cache to alternate between reads to disjoint code and data sections—causing simultaneous accesses which result in wait states *regardless of clock frequency*—*SwapRAM* reduces hardware cache pressure by offloading code accesses to SRAM, improving locality of access for the hardware cache and enabling systems to exploit NVRAM for program memory storage without the associated performance penalty.

5.5 Split SRAM Execution

SwapRAM primarily targets platforms that use NVRAM in a unified memory configuration to avoid the size constraints of on-chip SRAM. While memory size is a major driver of system design and cost, other considerations often determine final microcontroller choice (e.g., accelerators or analog features)—leading to over-provisioned SRAM when program memory usage is small. This underutilized SRAM represents an opportunity to further improve performance using *SwapRAM*. We extend our evaluation of *SwapRAM* and the block-based cache using the four benchmarks that can fit program memory in on-chip SRAM—`CRC`, `AES`, `bitcount`, and `RSA`—by splitting the SRAM into two regions: one for

program memory and one for the code cache. For each benchmark, we allocate sufficient SRAM to contain program memory (Table 1) and reserve the rest for caching. We compare this implementation to the baseline of executing code using FRAM/cache and storing program memory in SRAM.

Figure 10 illustrates *SwapRAM*'s execution speed improvement over the baseline systems in each configuration at 24 MHz, normalized to the original unified-memory results for context. In the split-memory configuration, *SwapRAM* realizes a 22% speedup and 26% energy reduction compared to the standard FRAM/SRAM approach—while the block-based approach at best meets the performance of the standard configuration, and significantly reduces performance on `AES` due to thrashing in the smaller cache. *SwapRAM* yields a similar 21% energy reduction at 8 MHz by eliminating relatively high-energy FRAM accesses, but only an 8% speedup as the baseline needs no wait states for FRAM accesses. These results demonstrate that *SwapRAM*'s approach nearly entirely eliminates the performance penalty of unified-memory operation when compared to the standard memory configuration, and is an effective way to maximize performance for *any* application on FRAM systems with underutilized SRAM.

5.6 Discussion

Prior work using the block-based cache we evaluate here struggles on resource-constrained systems as a result of overly-fine instrumentation demanding both a heavyweight runtime management system and a large number of instructions inserted throughout the application. By taking advantage of high-level observations about program semantics (i.e., instructions/blocks grouped together as part of a function are likely to execute together), *SwapRAM* sacrifices a small degree of precision for the ability to restrict instrumentation to only function calls. Because typical software has relatively few functions compared to basic blocks, *SwapRAM* also reduces the amount of state required to maintain the cache—saving scarce on-chip memory. While *SwapRAM* improves performance across a variety of benchmarks, its small impact on the `AES` benchmark suggests there is still a class of applications for which the *SwapRAM* approach can be improved. Our comparison of *SwapRAM* to the block-based system suggests that more closely integrating program semantics is a promising direction for improving performance, motivating future extensions to *SwapRAM* that take advantage of deeper static analysis or runtime code profiling.

6 Conclusion

Novel NVRAM technologies enable memory-intensive applications on small, ultra-low-power devices—but sacrifice common-case performance in both execution speed and energy consumption. We design *SwapRAM*, a software system that mitigates the performance penalty of NVRAM-based operation by caching and executing application code in underutilized on-chip SRAM. *SwapRAM* transparently interposes on software execution and redirects function calls to cached copies of user code, with an intelligent runtime system to minimize data transfer overhead. Experiments on real hardware show that *SwapRAM* increases execution speed by an average 26% while decreasing total energy consumption by 24% across a diverse set of benchmarks, eliminating the NVRAM performance bottleneck with a transparent software update.

SwapRAM demonstrates the potential of new approaches to general-purpose embedded system design that consider the capabilities and limitations of new memory technologies. These results motivate a range of future research directions further adapting systems to best leverage emerging memories at the language, compiler, and architectural levels.

7 Acknowledgments

We thank our shepherd Peter Chen for his guidance and the anonymous reviewers for their helpful suggestions. The project depicted is sponsored by the Defense Advanced Research Projects Agency. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Approved for public release; distribution is unlimited. This material is based upon work supported by the National Science Foundation under Grant No. 2240744.

References

[1] Miran Alhaideri, Michael Rushanan, Denis Foo Kune, and Kevin Fu. The Moo and Cement Shoes: Future Directions of A Practical Sense-Control-Actuate Application, September 2013. Presented at First International Workshop on the Swarm at the Edge of the Cloud (SEC'13 @ ESWeek), Montreal.

[2] ARM. ARM7TDMI Technical Reference Manual, November 2004. https://developer.arm.com/documentation/ddi0210/c/Memory-Interface/About-the-memory-interface?lang=en.

[3] Jean-Luc Aufranc. Texas Instruments Announces Ultra-Low Power MSP430 "Wolverine" MCU Series, February 2012. https://www.cnx-software.com/2012/02/29/texas-instruments-announces-ultra-low-power-msp430-wolverine-mcu-series/.

[4] Abu Bakar, Tousif Rahman, Rishad Shafik, Fahim Kawsar, and Alessandro Montanari. Adaptive Intelligence for Batteryless Sensors Using Software-Accelerated Tsetlin Machines. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, SenSys '22, page 236–249, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters*, 7(1):15–18, 2015.

[6] Daniel Beer. Debugging Tool for MSP430 MCUs, 2022. https://github.com/dlbeer/mspdebug.

[7] Andrea Bejarano-Carbo, Hyochan An, Kyojin Choo, Shiyu Liu, Qirui Zhang, Dennis Sylvester, David Blaauw, and Hun-Seok Kim. Millimeter-Scale Ultra-Low-Power Imaging System for Intelligent Edge Monitoring. *arXiv*, 2022.

[8] Juan Pablo Bello, Claudio Silva, Oded Nov, R. Luke DuBois, Anish Arora, Justin Salamon, Charles Mydlarz, and Harish Doraiswamy. SONYC: A System for the Monitoring, Analysis and Mitigation of Urban Noise Pollution, 2018.

[9] George Boateng, Vivian Genaro Motti, Varun Mishra, John A. Batsis, Josiah Hester, and David Kotz. Experience: Design, Development and Evaluation of a Wearable Device for MHealth Applications. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Yvonne Wang Bradley D. Nelson, Salil Sidharthan Karipott and Keat Ghee Ong. Wireless Technologies for Implantable Devices. In *Sensors*, Sensors, 2020.

[11] Tsai-Kan Chien, Lih-Yih Chiou, Shyh-Shyuan Sheu, Jing-Cian Lin, Chang-Chia Lee, Tzu-Kun Ku, Ming-Jinn Tsai, and Chih-I Wu. Low-Power MCU With Embedded ReRAM Buffers as Sensor Hub for IoT Applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 6(2):247–257, 2016.

[12] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-Specific Memory Management for Embedded Systems using Software-Controlled Caches. In *Proceedings 37th Design Automation Conference*, pages 416–419, 2000.

[13] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. Write-Light Cache for Energy Harvesting Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Jason Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman, and Yi Zou. An Energy-Efficient Adaptive Hybrid Cache. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 67–72, 2011.

[15] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. Camaroptera: A Long-Range Image Sensor with Local Inference for Remote Sensing Applications. *ACM Trans. Embed. Comput. Syst.*, 21(3), may 2022.

[16] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 199–213, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 670–684, New York, NY, USA, 2019. Association for Computing Machinery.

[18] 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 *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001.

[19] Matthew Hicks. Clank: Architectural Support for Intermittent Computation. In *International Symposium on Computer Architecture*, ISCA, pages 228–240, 2017.

[20] Matthew Hicks. MiBench2, 2023. https://github.com/impedimentToProgress/MiBench2.

[21] Andrew P. Hill, Peter Prince, Jake L. Snaddon, C. Patrick Doncaster, and Alex Rogers. AudioMoth: A low-cost acoustic device for monitoring biodiversity and the environment. *HardwareX*, 6:e00073, 2019.

[22] Gary Hilson. Flash Wearout Drives Tesla Recall, 2021. https://www.eetimes.com/flash-wearout-drives-tesla-recall/.

[23] Umesh Chand Jagan Singh Meena, Simon Min Sze and Tseung-Yuen Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9:526–559, 2014.

[24] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 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*, pages 330–335, 2014.

[25] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in IoT Edge Devices. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 264–269, 2016.

[26] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, page 256–269, New York, NY, USA, 2004. Association for Computing Machinery.

[27] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture Exploration for Ambient Energy Harvesting Nonvolatile Processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 526–537, 2015.

[28] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent Execution Without Checkpoints. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 96:1–96:30, October 2017.

[29] Kiwan Maeng and Brandon Lucia. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 129–144, November 2018.

[30] Andrea Maioli and Luca Mottola. ALFRED: Virtual Memory for Intermittent Computing. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, page 261–273, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Microchip Technology. Section 50. CPU for Devices with MIPS32 microAptiv and M-Class Cores, July 2015. https://ww1.microchip.com/downloads/en/DeviceDoc/60001192B.pdf.

[32] Microchip Technology. Low-Power, 32-bit Cortex-M0+ MCU with Advanced Analog and PWM, December 2021. https://ww1.microchip.com/downloads/aemDocuments/documents/MCU32/ProductDocuments/DataSheets/SAM-D21-DA1-Family-Data-Sheet-DS40001882H.pdf.

[33] Jason E. Miller and Anant Agarwal. Software-based Instruction Caching for Embedded Processors. *SIGARCH Comput. Archit. News*, 34(5):293–302, oct 2006.

[34] Mouser. Texas Instruments MSP430FRx Ultra Low-Power FRAM MCUs, 2022. https://www.mouser.com/new/texas-instruments/ti-msp430fr-fram-mcus/.

[35] Guillaume Patrigeon, Pascal Benoit, Lionel Torres, Sophiane Senni, Guillaume Prenat, and Gregory Di Pendina. Design and Evaluation of a 28-nm FD-SOI STT-MRAM for Ultra-Low Power Microcontrollers. *IEEE Access*, 7:58085–58093, 2019.

[36] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, 2008.

[37] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *15th International Symposium on System Synthesis, 2002.*, pages 213–218, 2002.

[38] S. Steinke, L. Wehmeyer, Bo-Sik Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 409–415, 2002.

[39] Robert Strenz. Review and Outlook on Embedded NVM Technologies – From Evolution to Revolution. In *2020 IEEE International Memory Workshop (IMW)*, pages 1–4, 2020.

[40] Texas Instruments. FRAM FAQs, January 2014. http://www.ti.com/lit/ml/slat151/slat151.pdf.

[41] Texas Instruments. MSP430FR2355 LaunchPad Development Kit (MSP--EXP430FR2355), 2018. https://www.ti.com/lit/ug/slau680/slau680.pdf.

[42] Texas Instruments. MSP430FR698x(1), MSP430FR598x(1) Mixed-Signal Microcontrollers, 2018. http://www.ti.com/lit/ds/symlink/msp430fr6989.pdf.

[43] Texas Instruments. MSP430FR235x, MSP430FR215x Mixed-Signal Microcontrollers, 2019. https://www.ti.com/lit/ds/symlink/msp430fr2355.pdf.

[44] Texas Instruments. MSP430 GCC Toolchain, 2020. https://www.ti.com/lit/ug/slau646f/slau646f.pdf.

[45] Texas Instruments. MSP430F552x, MSP430F551x Mixed-Signal Microcontrollers, 2020. https://www.ti.com/lit/ds/symlink/msp430f5529.pdf.

[46] Texas Instruments. Low-Power FRAM Microcontrollers and Their Applications, September 2023. https://www.ti.com/lit/wp/slaa502a/slaa502a.pdf.

[47] Texas Instruments. MSPM0L130x Mixed-Signal Microcontrollers, 2023. https://www.ti.com/lit/ds/symlink/mspm0l1306.pdf.

[48] Harrison Williams, Xun Jian, and Matthew Hicks. Forget Failure: Exploiting SRAM Data Remanence for Low-Overhead Intermittent Computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 69–84, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Joel Van Der Woude and Matthew Hicks. Intermittent Computation without Hardware Support or Programmer Intervention. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 17–32, November 2016.

[50] Ozan Yigit. Hash Functions, 2006. http://www.cse.yorku.ca/~oz/hash.html.