System-wide Batch Peripheral Scheduling in Multi-tenant Embedded Systems

Marshall Clyburn
marshallc@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Victor Cionca
victor.cionca@mtu.ie
Munster Technological University
Cork, Ireland

Brad Campbell
bradjc@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Abstract

Multi-tenancy, the co-location of applications on a single device, increases the utility of embedded systems and lowers deployment costs. Support for multiple applications has even reached lower-power battery-powered devices which benefit from modularity and multitasking, like other computing platforms. But energy remains a concern for these devices, and multi-tenancy can increase energy usage as applications execute uncoordinated with each other; their CPU and peripheral usage wake the device from low-power modes and collectively increase its active time. To address this, we exploit the inherent uncertainty in multi-tenant systems and dynamically shift peripheral activity to increase energy efficiency. Our approach executes within the OS, requiring no modifications to applications. We explore policies that trade off energy efficiency and latency and design one that accommodates latency-sensitive operations by considering operations to execute ahead of time. Our evaluation shows our approach outperforms equivalent traditional monolithic applications. With this approach, low-power embedded systems can benefit from multi-tenancy without sacrificing energy performance.

CCS Concepts

• **Computer systems organization** → **Embedded and cyber-physical systems**; **Embedded software**; *Sensors and actuators*; • **Software and its engineering** → **Scheduling**; **Power management**.

Keywords

embedded systems, operating systems, batching, peripherals

1 Introduction

Embedded hardware now supports multi-tenancy: the co-location of multiple, distinct software applications on a single device. Multi-tenant embedded systems support applications in wearables and the Internet-of-Things, including tracking personal health and fitness activity [7, 8], providing city-scale sensing [10], and monitoring building infrastructure [11]. Multi-tenant systems allow the system designer (responsible for integrating the hardware and the embedded OS) and application developers to be completely distinct entities. Multi-tenant systems enable increased flexibility, lowered deployment costs, and expanded utility over single-application devices. For example, a user can install applications and extend the functionality of their smart wearable device, or a local government may extend a traffic light sensors to assess the city's busiest routes, obviating the need for new hardware. Like conventional computers embedded hardware is fulfilling more general-purpose roles. The many sensors and accelerators available on-chip make them suitable as multi-purpose devices.

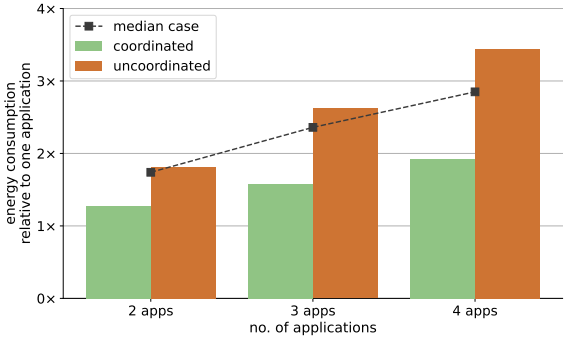

Figure 1: Energy usage variation of a device running applications. The applications sense light, sample with the ADC, refresh a display, and encrypt data. Consumption is relative to the light-sensing app. Working independently, these applications rarely align their execution, yielding a high median energy consumption.

However, energy remains a leading concern for these systems. Careful energy management is critical to ensure sustainable operation on limited energy budgets. Low-power "sleep" modes enable these devices to power off components and stop clocks, a significant contribution to energy consumption [14], vastly reducing power draw. Traditionally, embedded systems fit the purpose of a single application, allowing a single developer to tune the application and control energy usage. In multi-tenant systems however, multiple applications work concurrently and independently. This complicates energy consumption behavior of the device as applications execute uncoordinated with respect to each others' activity.

Applications make use of the peripherals available in embedded hardware to sense, actuate, and process data. Their uncoordinated activity keeps the device active in disjoint spans of time, often for a single application. Even a small set of simple applications can exhibit a wide range of energy consumption, as Fig. 1 illustrates. There is a high baseline energy cost for a device to be active, so active time is a large factor in energy usage. Though a CPU scheduler can control application code execution, applications can initiate peripheral operations, which run independent of the CPU. Just like the CPU, peripherals require clocks to function, meaning applications can indirectly inhibit low-power modes beyond the control of a CPU scheduler. The CPU scheduler alone is not sufficient to control applications' energy usage, leaving unpredictability in energy consumption. Bringing coordination to applications in multi-tenant systems will not only allow these devices to last longer but also allow them to *do more* with the energy they have.

To control energy usage arising from peripheral activity, we introduce a batching system designed specifically to control peripheral operations. Batching peripheral operations makes it possible to parallelize their execution, thereby reducing the amount of time the system is active for only a single application or peripheral. However, batching as an energy management technique presents challenges. Most obvious of them is the additional latency it imposes on applications. Applications will experience delays in starting peripheral operations and in receiving resulting data or notifications from interrupts. It is also necessary to define a policy that determines when to withhold operations from execution and when to execute them. This affects the system's responsiveness as well as its energy efficiency. These concerns run counter to each other; a responsive policy will batch less effectively and consume more energy, but an energy-conservative policy will be less responsive.

To provide efficient, responsive batching to embedded systems, we present a batching framework that integrates with the embedded operating system (OS) and a batching policy that takes advantage of the periodic nature of applications to execute latency-sensitive operations efficiently. Though individual applications may batch their own operations themselves, batching at the system level enables considering multiple applications' activity for batching. An important benefit to this approach is that applications are unaware that batching is happening. By making it entirely a concern of the underlying system, application code need not change to work with the batching system.

The batching system aggregates peripheral operations and executes them according to the policy. The batching policy determines when to execute on peripheral operations it accumulates. Given the importance of low latency to some applications, we design a batching policy that gives special consideration to latency-sensitive operations and maintains energy efficiency when responding to unpredictable interrupts. This policy identifies peripheral operations as latency-sensitive, batchable, or pre-executable and maintains awareness of upcoming operations to execute some ahead of time with latency-sensitive operations to create ad-hoc batches.

We evaluate the batching system in terms of energy efficiency, latency, and overheads across several application setups. It achieves better efficiency than equivalent, monolithic applications and a state-of-the-art harmonizing scheduler while also accommodating latency-sensitive operations. The evaluations also characterize the effect of the ahead-of-time batching policy on energy efficiency to discern the conditions in which it is and is not effective.

2 Related Work

Energy management is a prominent topic in computing. This work expands on existing work by highlighting the importance of peripherals to energy usage in multi-tenant embedded systems and presents a scheduler to reliably control their energy usage. Here we examine related work in scheduling for energy efficiency in low-power embedded devices, the application of batching for energy efficiency, and managing peripheral hardware energy usage.

Existing literature places much focus on CPU-only workloads. Most prominent are techniques that use DVFS to adjust power to the CPU [20, 24]. Niu et al. present a peripheral-aware DVS algorithm [19], but it uses peripheral power state to determine whether to run a job. In this work, we focus on manipulating the timing of peripheral operations to reduce energy usage.

The benefit of batching to energy efficiency is well-known. In [15], the authors suggest it as an effective technique for power-hungry components. Davies et al. present a scheduling algorithm for balancing energy and flow time [13], but it is an offline algorithm and requires the number of batches to create as input. Gupta et al. batch data transmissions from multiple applications to reduce energy usage in [16], but it is only scoped to radio transmissions. Rowe et al. present the energy-saving rate harmonic scheduler [21, 22] to harmonize the execution of periodic tasks to reduce energy usage but focus only on the CPU and do not consider aperiodic operations. We use batching to improve the execution of peripheral operations, but the batching policy we present also executes latency-sensitive operations without delay while maintaining efficiency.

There are also works in mobile computing that target efficiently using power-hungry peripherals like GPS and cell modems. APE [18] is a middleware that allows an application developer to write rules that the OS uses to decide how to run specific operations with energy-intensive components. Cinder [23] allows a user to budget energy between apps, and apps pool their energy to run energy-expensive operations. These techniques that may be applicable in the low-power embedded domain, but in this work, we develop a batch peripheral scheduling system for multi-tenant embedded systems that does not require developers to specially adapt their software to work on the platform.

3 Modeling and Opportunity

To understand the efficacy of batching, we build a mathematical model of batch peripheral scheduling and discuss results of a motivational experiment. The model shows the potential of controlled peripheral operation scheduling, and our experiment results show the additional efficiency possible with system-level batching.

3.1 Modeling batch peripheral scheduling

To understand the theoretical limits of energy efficiency through batching, we model the energy consumption of executing operations with and without batching. Modeling batch operation provides an approximate upper bound on energy efficiency, reveals thresholds at which batching becomes unhelpful, and indicates whether an implementation functions correctly. Our mathematical model consists of the following:

$$E_{\text{idle}} = P_{\text{idle}} \times t_{\text{idle}} \tag{1}$$

$$E_{\text{marginal}} = \sum_{i=0}^{N} P_{\text{periph. i}} \times t_{\text{periph. i active}} \tag{2}$$

$$E_{\text{baseline}} = P_{\text{base}} \times \sum_{i=1}^{N} t_{\text{periph. i active}} \tag{3}$$

$$E_{\text{batching}} = P_{base} \times \max(T_{\text{periph. active}}) \tag{4}$$

$$F = 1 - \frac{E_{\text{batching}} + E_{\text{marginal}} + E_{\text{idle}}}{E_{\text{baseline}} + E_{\text{marginal}} + E_{\text{idle}}} \tag{5}$$

Table 1 lists the model variables. The model considers a span of time during which N peripherals execute operations. Each peripheral operation executes for some duration ($t_{\text{periph. i active}}$) and causes the

notation	description
E_{idle}	energy consumed during low-power operation
E_{marginal}	peripheral's contribution to energy usage
P_{idle}	power of low-power operation
t_{idle}	time spent in low-power mode
$P_{\text{periph. i}}$	power of peripheral i
$t_{\text{periph. i active}}$	duration peripheral i is active
E_{baseline}	energy consumed without batching
E_{batching}	energy consumed with batching
P_{base}	device active power
$T_{\text{periph. active}}$	set of all peripherals' active durations
F	energy consumption reduction

Table 1: List of energy efficiency model variables.

peripheral to consume energy ($P_{\text{periph. i}}$). The total consumption of that duration is the sum of three values. Two of the three are shared between the batching and non-batching cases: energy consumption in a low-power state (E_{idle}, equation 1) and energy consumption to solely power a peripheral (E_{marginal}, equation 2).

Energy consumption between the batching and non-batching cases differs due to the concurrent execution of multiple peripheral operations, reducing the total time the system is active. The non-batching case uses E_{baseline} to compute energy consumption, modeling peripheral operations that execute in disjoint spans of time. The batching case uses E_{batching} to compute energy consumption, modeling a system that concurrently executes all peripheral operations and remains active only for as long as the longest peripheral operation to execute. We obtain the theoretical reduction in energy by comparing the total energy consumption of the batching and non-batching cases (F, equation 5).

The model estimates energy efficiency of batching with some assumptions that allow the model to remain simple. It assumes that peripherals start at the same instant which guarantees the device is active no longer than the longest operation. Actual start times are separated by a few microseconds, which is a relatively small fraction of time compared to the tens or hundreds of milliseconds that peripheral operations typically last. Secondly, it represents peripheral operations as continuous events with a single start and stop time although some operations may consist of bursty activity and exhibit a non-uniform power trace. Also, it does not consider the energy and time spent shifting between active and low-power states. These transitions take little time, on the order of single-digit microseconds, to complete. Nevertheless, the model is useful for analyzing the potential of batch peripheral scheduling.

3.2 Application- and system-level batching

To validate the model and reveal real-world energy efficiency gain, we performed a preliminary study. We ran an application performing four tasks: transmitting over UART, encrypting with an AES accelerator, querying a temperature sensor, and sampling with the ADC. We ran the application on Tock OS [17] on the Hail [1] development board and measured energy consumption. The application executed each task 100 times over 30 seconds, batched in three ways: all four tasks, two groups of two tasks (ADC + AES, UART + temp.), and one group of three tasks with one isolated task (ADC task

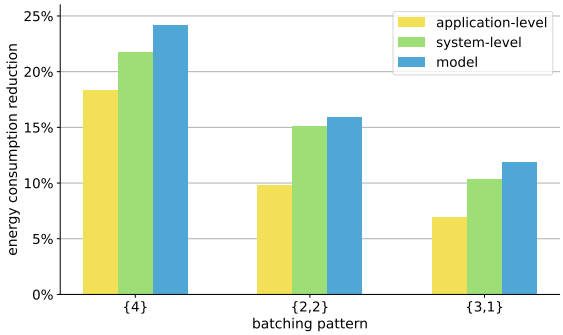

Figure 2: Reduction in energy consumption by batching peripheral activity. Application-level batching batches within application code, and system-level batching works within the embedded OS. Each pattern is a tuple describing how many operations are in each batch. Executing batches at the system level further improves efficiency.

alone). A small delay separated each batch. In this case, batching is happening at the *application* level.

We then modified Tock to perform batching at the *system* level (within the OS) and re-ran the same application. In this case, the kernel withheld peripheral operations until the batch was complete and then dispatched the calls in quick succession. As a baseline, we run the four tasks 100 times with a small delay between all tasks. Figure 2 shows the results. It is particularly notable that batching even just two peripheral operations together provides a worthwhile gain (approx. 15%) in efficiency. The results show that batching not only improves energy efficiency but also that batching at the system level offers further improvement. System-level batching performs better than application-level batching in all three setups.

But batching places a latency cost on the completion time of operations. Applications will experience delay in receiving results from operations. This latency cost and the reduction in energy usage should be in balance to make batching as applicable as possible. Having control of the trade-off between latency and energy is a desirable aspect for batching in embedded systems.

4 Design

The main focus for the batching system is the reduction of energy usage, but there are other key goals for the system:

- *Support latency-sensitive applications.* Embedded devices are often event-driven. Even though batching is opposed to this goal, supporting a wide variety of devices requires accommodating these applications.
- *Operate transparently.* Applications should not have to interact with the batching system. This allows developers to keep applications code portable. It also reduces the implementation complexity in memory-constrained hardware.
- *Be configurable.* It should be straightforward for the system designer to adjust when and how the batching system works: how often to execute batches, how long to retain peripheral operations, which operations to batch, etc. It must also be possible to simply disable the batching system.

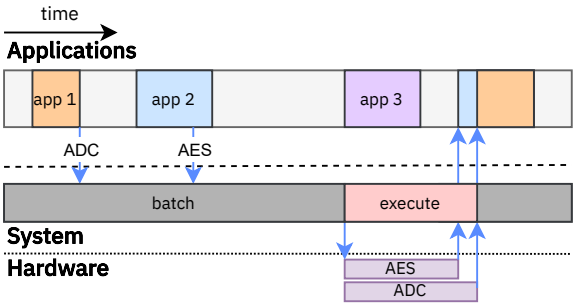

Figure 3: Overview of system-wide batching. The system exploits asynchrony to batch peripheral operations. The batch contains requests for the ADC and the AES accelerator. The ADC and AES operations execute simultaneously, and applications receive the results through callbacks.

This design exists entirely within the embedded OS and specifically targets parallelizing the operation of the peripherals found in the device. With the batching system as a foundation, we design a batching policy to optimize for lower latency and higher energy efficiency while still allowing time-sensitive tasks to execute without delay. To maximize the opportunity to improve energy efficiency, we focus on general control of all peripherals and do not target a specific class. Optimization is possible for specific types of peripherals, such as for MCUs that actually feature multiple blocks of ADC sampling hardware [4], but this is a narrower optimization.

4.1 Requirements for system-level batching

To make batching at the system level work, the system must mediate all interactions between applications and peripherals. This separation enables manipulating when applications' calls reach drivers. Applications, the system, and drivers must also support asynchrony for batching to be effective. Applications must support receiving the results of peripheral operations through callbacks. Respectively, the system must allow applications to continue executing after issuing a driver call, otherwise, a batching system could only consider up to a single operation per application. Also, peripheral drivers must not poll hardware and waste execution time. Drivers must instead rely on interrupts so that the system and CPU run independent of peripherals.

Manipulating the progression of driver calls is the key to batching the resulting peripheral operations. The system must be able to stop calls from reaching drivers at the original time of invocation to delay operations for batching. The system must also be able to then issue the call at any time after receiving it from the application to determine when to execute batched operations.

When an application makes a peripheral driver call, the system receives the request and stores data necessary to make the call to the driver later. This information includes the originating application, the target driver, and the arguments necessary to make the call. The system then returns control to the application, continuing execution. Once the system decides to execute a batch, the system issues the withheld calls in quick succession, effectively parallelizing the peripheral operations. As each completes, drivers deliver results to applications through callbacks. Figure 3 illustrates this in action.

4.2 Integration with the OS

Modifying the OS kernel offers a strong basis for the batching system, providing the requisite control to perform batching and also supporting the goal of operation transparent to applications. The context information necessary to execute batched operations and to return their results is readily accessible from within the kernel (e.g., the reason for userspace-kernel control transfer, or which application a call originates from). This simplifies implementation compared to an approach where the kernel is completely unaware of the batching system. However, instead of directly modifying the kernel to perform batching, we pursue an approach where batching logic is separate from the kernel to achieve a modular, adaptable design. This separate logic implements the *batching policy* that the system designer customizes to suit their requirements.

The kernel makes calls to hook functions provided by the batching policy at specific points in its control flow. They allow the batching policy to interact with the kernel at runtime. The hook functions decide when to batch, what to batch, and when to execute. The first batching policy hook function determines whether the kernel should run an incoming driver call. It occurs when control flow returns from an application to the kernel after the kernel determines that the application is attempting to invoke a driver. The second batching policy hook function returns batched driver calls the policy is ready to have execute. The call to this function occurs at the beginning of the scheduling loop before the kernel selects an application to run on the CPU, a frequently-run area of code that will ensure batch executions begin with little delay.

Other batching system designs could achieve the transparency and configurability necessary but pose difficulties that make them infeasible for implementation. Implementing it as a new, separate layer between userspace and the kernel would result in a highly modular design, but it would suffer from a lack of context information about system activity. This approach must discern driver calls from other reasons for control flow switching between applications and the kernel, and it would have to track which applications are running as well as require code for interacting with drivers. The kernel already has facilities for this. Another approach would be to make batching a function of the existing CPU scheduler, which would make for a pluggable implementation but introduce unnecessary complexity by combining the concerns of selecting when applications can run their code and when peripheral operations occur. It forces the system designer to implement an entirely new scheduler to change either the scheduling or batching policy.

4.3 Batching units of work

The primary target of batching is the hardware operation of a peripheral. However, we also consider the steps preceding and following the operation to maintain transparent function and to avoid implementation complexity. Application execution naturally consists of driver calls, interrupts, and peripheral hardware activity. Control flow also involves hand-offs between different abstraction layers. This means that there are many possible definitions of a unit of work for batching.

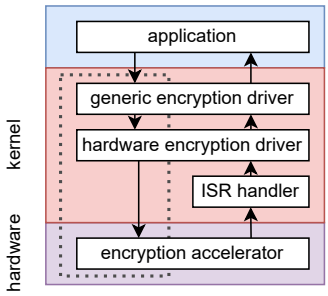

Figure 4: General execution flow of a driver call illustrated using an AES driver as an example. This model shows the software abstraction layers where batching is possible. The dashed rectangle outlines the focus of our batching system.

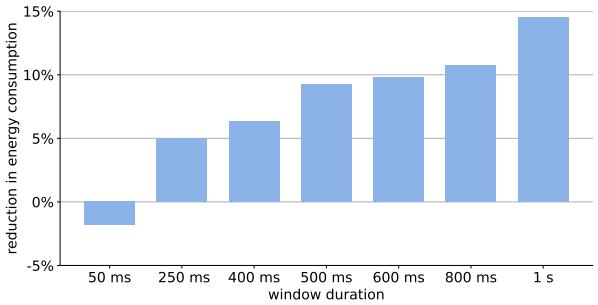

Figure 5: Efficiency of window durations in a system with a mean interarrival time of $\lambda = 428$ ms. There is a notable decrease in efficiency after shortening the window below this duration. This provides a useful guideline for adjusting batching window length for time-window-based policies.

The OS goes through several steps to execute a peripheral operation (Fig. 4). The kernel receives a driver call from the application and invokes the driver. The driver performs prerequisite configuration and runs hardware-specific code to start the peripheral operation. Upon completion, the peripheral will issue an interrupt. The hardware-specific code will, once again, execute to service the hardware followed by the rest of the driver code. The driver will update internal state and issue a callback to the application to notify it of completion, possibly delivering data as well. Drivers may deviate from this flow depending on how they or the hardware work, but this is a generally applicable model of the execution flow. Importantly, when running atop an embedded OS that utilizes a scheduler to select applications for execution, there is already no guarantee on the exact timing of these interactions. This makes it easier to interpose batching at points between these steps.

There are, however, problems that can arise that we are careful to not introduce so as to limit complexity. Separating generic and hardware-specific driver code can impede drivers that require accurate timing (e.g., the 1-wire protocol [9]), and delaying hardware servicing may result in lost or corrupted data. Therefore, drivers must have an accurate view of hardware state at all times, and they must be able to service hardware immediately after interrupts. Delineating batchable units of work between these points would require additional handling to resolve these issues.

Given general execution flow and the aforementioned potential complexities, we determine that a unit of work for the batching system includes the hardware-agnostic and hardware-specific portions of a driver and the initiation of the peripheral. This means the batching system will aggregate driver calls that the kernel receives to build batches and issue these calls later to execute the batch. It is also possible to consider the eventual callback to the application, but we focus on the peripheral operation in this work.

4.4 Efficient, responsive batching

Deciding when to execute a batch determines the efficiency the system can attain and the latency it imposes on applications. Even without considering latency-sensitive operations, the system should balance energy efficiency and latency. There are two simple policies that attain either extreme of energy efficiency and timeliness: time- and count-based batching. We take observations about each to build a third policy that can adapt to activity frequency.

The first simple policy is to batch for some fixed duration. The window begins when the first unit of work arrives. At the end of the window, the policy executes all batched work. This policy upper-bounds the latency of work. However, it does not provide efficiency if no more than one peripheral operation accumulates before the window expires. Tuning the window duration is key to attaining energy efficiency in this policy.

To observe the effect of window duration on energy efficiency, we ran three applications. These applications wrote data over UART every second, queried a temperature and a humidity sensor every two and three seconds, and sampled with the ADC every three seconds. Figure 5 shows the energy efficiency of each window duration. The longest window, 1 second, achieves the best energy efficiency. It is the minimum duration possible without overlapping the shortest period. Subsequently shorter windows achieve dwindling efficiency. There is a notable drop in efficiency between the 1-second and 800 ms windows, but there is also one between the 500 ms and 400 ms windows. Between 400 and 500 ms is the mean interarrival time, 428 ms, according to the Poisson distribution. Windows shorter than this value are noticeably less likely to batch more than a single operation than window sizes greater than the mean. These results suggest that the minimum of all periods and the Poisson mean are useful durations for deriving fixed time windows.

The other simple policy is to aggregate a fixed number of operations before executing. A fixed-count batching policy provides a strong guarantee of efficiency by forcing a number of operations to accumulate before the batch executes. The system designer can set the required batch size based on what they know about the activity of the system and its applications, but even just two operations give notable efficiency (Section 3.2). However, this policy provides no latency guarantees; an operation could await execution a long time before the system finally executes it.

Based on observations about the two simple policies, we design a hybrid policy that combines both to balance energy efficiency and latency. Embedded applications often function by periodically running code. This provides a good basis for deriving time windows.

As we previously observed, one duration we can use is the minimum period of all periodic activity. This is a window that avoids redundant requests from applications. We call this the *conservative* window. However, we also use the periodicity of embedded applications as a basis to gauge how often a single batchable operation arrives. By setting the duration to the mean interarrival time, we can likely form batches of at least two operations. We call this the *responsive* window. The hybrid policy can use either of these durations or one between them to balance efficiency and latency.

The hybrid policy also incorporates our observations about fixed batch sizes. When the hybrid policy accumulates two batchable operations, it executes on the batch regardless of the time remaining in the window. In so doing, applications can experience less latency at the expense of giving up additional energy efficiency possible from accumulating more than two batchable operations.

4.5 Ahead-of-time batching

The hybrid policy provides a balance between energy efficiency and responsiveness, but latency-sensitive applications must operate without delay. For example, interacting with a user navigating a UI, or occupancy detection in dangerous work areas [5]. The delay that separates these events is unpredictable and can occur before a single batchable event has arrived. When they occur after a single batchable event is awaiting execution, the hybrid policy can immediately execute on the batch and operate efficiently, but when they arrive before a batchable event has arrived, there is a loss of energy efficiency because there is no other operation to execute with the latency-sensitive operation.

Key to improving energy efficiency in the face of latency-sensitive events is the consideration that some applications may be amenable to having their peripheral-bound activity run sooner than originally scheduled. Though not possible for all peripheral operations, the OS can run some peripheral operations without any input from applications. This takes advantage of the periodic nature of embedded applications, which allows the OS to learn the schedule of operation requests by tracking when they execute and which applications make the requests. When a latency-sensitive operation must execute, the OS can use its knowledge of upcoming scheduled operations to anticipate and run another operation alongside the latency-sensitive operation. Then, when the application makes its request for the peripheral operation that occurred ahead of time, the OS delivers the result obtained from the expedited execution. We term this preemptive execution *ahead-of-time batching* (AoT).

Ahead-of-time batching requires that we classify operations the device is capable of performing. In our classification, We deem peripheral operations either latency-sensitive or batchable. Latency-sensitive operations are not subject to batching and execute immediately. Batchable operations are those the system can delay the execution time of to improve energy efficiency. Among all batchable operations is a subset with unpredictable inputs (e.g., encryption, transmitting sensor data) and are thus not executable until the application makes the driver call. All other batchable operations can execute ahead of their normally scheduled execution time independent of an application's call. When an application makes a request for an operation that was pre-executed by the policy, the OS immediately returns the cached data.

We extend the hybrid batching policy with the ahead-of-time batching capability to create the new batching policy. This changes the behavior of the hybrid policy in two cases: when receiving the first batchable operation in a batch and when a latency-sensitive operation arrives before a batchable operation. In the first case, upon receiving the first batchable operation of a batch the policy adds the operation to the batch but also consults the schedule of upcoming operations for the operation that is to occur the soonest. To avoid pulling forward progressively later and later scheduled operations, only operations scheduled with a time remaining less than their period are eligible. In the second case, upon receiving a driver call for a latency-sensitive operation, the policy pulls forward the operation that is to occur the soonest. Both the latency-sensitive and expedited operation run together as part of an ad hoc batch, allowing both the latency-sensitive operation to run without additional delay and the device to use energy more efficiently.

We call this policy the ahead-of-time (AoT) batching policy and provide pseudocode for its function in Algorithm 1. The AoT policy is one that executes latency-sensitive operations quickly yet still attempts to use energy efficiently. As just another batching policy, it interacts with the kernel and is one of the many possible strategies to perform batching.

Algorithm 1 Pseudocode of the ahead-of-time batching policy.

```
e ← current request
b ← currently batched requests
S ← upcoming scheduled events
procedure GET_AOT_OPERATION
    for each scheduled event, s, in S do
        if time_remaining(s) < period(s) then
            b += operation(s); break
        end if
    end for
end procedure
if e is latency-sensitive then
    b += e
    if |b| == 1 then
        b += GET_AOT_OPERATION
    end if
    run(b)
else
    b += e
    if |b| == 1 then
        begin_window()
        b += GET_AOT_OPERATION
    else if |b| ≥ 2 then
        run(b)
    end if
end if
```

5 Implementation

We implement the batching system on Tock [17]. Tock is an embedded OS for microcontrollers. Tock imposes a strong boundary between applications and the kernel; the code for each of the entities is separate, and drivers execute in the context of the kernel.

The kernel uses a scheduler to select which application to run on the CPU, meaning that applications are not always running and that the scheduler could delay a runnable application indefinitely.

Applications communicate with drivers through a standardized syscall interface. A syscall driving a GPIO pin high on one platform will look the same on every other platform Tock supports. Syscalls provide applications both high-level and fine-grained control over hardware (e.g., a single syscall can refresh a display with several SPI transactions or just trigger a single SPI transaction). Inspecting and manipulating syscalls is the key to performing batching in Tock.

5.1 Batching policy

The batching policy is defined by the `BatchController` interface. The interface requires the implementation of a handful of functions:

`check_enqueue(Process, Syscall)` decides if a call from an application should be batched, storing the call or indicating that it should be executed immediately.

`dequeue_syscall()` returns a batched syscall.

`state()` returns the current policy state.

The kernel uses these calls at runtime to determine its operation.

The batch controller is a state machine that switches between two states: `Batch` and `RunSyscalls`. It transitions between the two based on the policy design. For example, a fixed time window policy would start in the `Batch` state. After a syscall arrives, it sets a timer. Once the timer expires, it transitions to `RunSyscalls`. Once the kernel exhausts the batched syscalls with `dequeue_syscall()`, the batch controller transitions back to `Batch`. Determining when the batch controller alternates between the two states determines how the batching system works.

The batch controller maintains all information necessary to issue syscalls later. This includes the destination driver, the call being made to the driver, the arguments, and the calling application. When the batch controller switches to the `RunSyscalls` state, the kernel fetches pending calls and dispatches them to the respective driver.

5.2 Kernel changes

Our modifications to the Tock kernel add calls to the batch controller to integrate batching. These calls change the how the kernel handles incoming syscalls as well as when the kernel runs syscalls awaiting execution. The behavior of the kernel is dependent on the current state of the state machine batch controller maintains.

After receiving a syscall, the kernel calls `check_enqueue()` to determine how to handle it. This call occurs just before the kernel's syscall dispatch after control returns to the kernel. By doing this, all syscalls made by applications pass through the `check_enqueue()` function. The batch controller returns a decision: either that the kernel should execute the syscall immediately or that it is retaining the call, and the kernel should not execute the syscall.

The kernel queries the state of the batch controller state machine at the top of each iteration of the scheduling loop to determine if it should run batched syscalls. There is flexibility in where this can happen in the code, but we place this hook function call just before the kernel invokes the scheduler to select a process to run. If the call to `state()` returns `RunSyscalls`, the kernel calls `dequeue_syscalls()` until it exhausts the syscalls the batch controller has retained. As the batch controller returns pending syscalls, the kernel completes the calls to the drivers, running the delayed calls in quick succession.

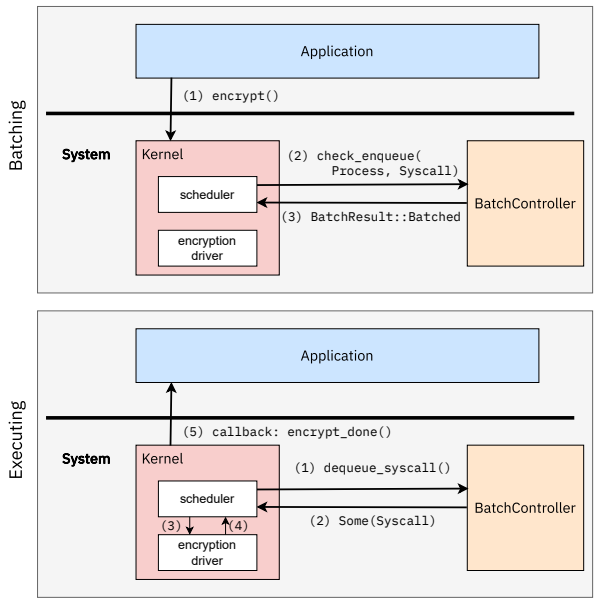

Figure 6: Call flow for batching and executing batched peripheral operation syscalls. When the batch controller's state machine is in the batching state, the kernel performs no action with peripheral operation syscalls. Once the batch controller switches to the execution state, the kernel initiates the batched operations in quick succession, returning results to the applications upon their completion.

5.3 Ahead-of-time batch controller

The implementation of the AoT batching policy adheres to the batch controller interface. To implement the time window-based functionality, the controller uses on-chip timer hardware. Since the controller retains the data about withheld syscalls, it is simple to provide the count-based batch execution behavior.

The AoT controller maintains a lookup table to determine how to handle syscalls that the kernel sends through `check_enqueue()`. By default, the controller performs no special action on a syscall. It passes through unaffected. It is easier to specify the syscalls that require special handling. Because Tock standardizes syscalls, the table and entries are portable across different hardware platforms. The system designer can easily add or remove entries to the table to configure syscall batching behavior of specific syscalls.

To make predictions for ahead-of-time batching, the batch controller must know which peripheral operations are upcoming and when those operations are scheduled to execute. Because applications interact with the alarm driver to schedule callbacks, the controller can learn applications' periodic schedules by inspecting alarm driver syscalls as they pass through `check_enqueue()`. It maintains a table storing these timers and their source application and tracks the time remaining for all alarms each time the kernel

application	operation(s)	batch?	AoT?
accelerometer	query accelerometer	✗	✗
biometrics	read temperature	✓	✓
	read GSR	✓	✓
	encrypt data	✓	✗
display	refresh screen	✓	✗
loudness	collect audio samples	✓	✓
synchronize	configure radio	✓	✗
	send data	✗	✗

Table 2: Evaluation applications, operations they execute, and whether they are batchable or executable ahead of time.

calls `check_enqueue()`. In this implementation, we use a naïve method to correlate a peripheral operation to an alarm. It assumes that any peripheral operation syscalls originating from the application in the execution of its timeslice following its alarm firing is the periodically occurring peripheral operation.

In Tock, applications have their own buffers and data they share with the kernel to move data to and from drivers, however, interacting with these at runtime without the knowledge of the application (e.g., an ahead-of-time execution placing ADC samples in a buffer while the application is performing analysis on the data currently in the buffer) can yield unexpected behavior. To safely execute peripheral operations before an application actually requests them, we implement a substitute process, `ThinProcess`, that implements the bare minimum function of a process that the kernel never actually schedules. The `ThinProcess` has its own grant memory and buffers that the controller configures for a single application and driver at a time. The controller executes syscalls through this structure, and `ThinProcess` retains results for expedited syscalls,

Syscalls differ in how they return data to applications (e.g., a return code or a return code and a value). We encode the return behavior of each ahead-of-time-eligible syscall in the `ThinProcess` implementation itself with another lookup table. The batch controller can forward data back to the application depending on how the syscall itself functions. Just as with syscall handling in `check_enqueue()`, this information is portable between platforms, as Tock standardizes syscall behavior. Regardless of what data a syscall returns, drivers deliver results of peripheral operations through callbacks. Therefore, to deliver the result of a pre-executed operation once the application actually makes the corresponding syscall, the controller queues a callback using an application's `Process` control structure.

6 Evaluation

We performed a series of evaluations to reveal the batching system's effects on energy consumption, latency, and system overhead. We performed our evaluations on the Hail [1] development board running an ARM Cortex-M4 MCU. In addition to using the onboard hardware, we connected a 2.13-inch Waveshare e-Paper display [6] to Hail via SPI. We used a Raspberry Pi 2B [3] and a Texas Instruments INA219 [2] current sensor to control evaluations and to record the total energy consumption.

6.0.1 Evaluation applications. The evaluations use applications that execute periodically and run in response to external events. These applications initiate various peripheral operations. Table 2 provides an overview of the applications and their function. They provide a mix of activities that one could find on a wearable device. **Accelerometer (accel)** is a pedometer. It registers threshold notifications with the accelerometer to count steps. Upon receiving a notification, the application validates vector values to track motion. This is a latency-sensitive operation. **Biometrics (bio)** monitors biometric data. It senses temperature and galvanic skin response (GSR). After collecting data for both, it encrypts the data. **Display (disp)** refreshes the e-ink display. **Loudness (loud)** monitors the loudness of the environment. It collects audio and uses it to calculate the noise level. **Synchronize (sync)** sends data over Bluetooth. It exchanges data with a separate, central Bluetooth device. Both configuring the radio and sending data are latency-sensitive, as wireless communication may be adversely affected by delay.

To run controlled, repeatable experiments involving *accel*, we instead deliver interrupts to a GPIO pin on Hail using the Raspberry Pi. These interrupts are jittered to mimic small variation in the pace of a human walking. The application still registers for vector threshold interrupts from the sensor and responds to the synthetic interrupts from the Raspberry Pi by running the same code to retrieve data from the accelerometer.

6.0.2 Batching implementations. We compare the ahead-of-time batching policy with other implementations. **Simple** is an in-kernel system that batches by monitoring the running state of processes, batching indirectly through the CPU scheduler. When a process is ready, the kernel waits for a period of time before running the application, accumulating other processes that become ready to run. **Time** is a fixed time window policy as described in Section 4.4. We implement this as a batch controller (Section 5.1). **Monolithic** combines the function of multiple applications into one and performs batching within the application code. This represents a traditionally-developed embedded application. **ES-RHS** [21] is a rate-harmonizing scheduler that aims to align the phase of scheduled operations to perform batching. Operations that arrive must wait until the next harmonizing period before executing, at which point all waiting operations will execute. It performs no special handling for latency-sensitive operations. We implement this as a batch controller (Section 5.1).

6.1 Coordinating application execution

We first revisit the experiment from Section 1 to show the batching system consistently coordinating applications. We use the same applications as before. These applications still exhibit variation in when exactly they schedule their periodic activities. Tock starts applications sequentially, so depending on how long applications run at startup, individual applications will naturally not align their own periodic executions.

Figure 7 presents the results alongside the original results (from Fig. 1). Across all three configurations, the AoT policy optimizes for lower energy consumption and lower energy consumption variation when executing multiple applications. Across 100 trials, the minimum and maximum energy consumption for AoT and the batch controller framework is only slightly higher than the coordinated scenario for the non-batching system.

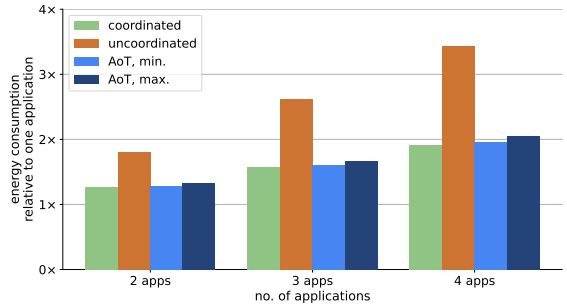

Figure 7: Coordinated and uncoordinated energy consumption by applications with no batching and the minimum and maximum energy consumption using the AoT policy. The AoT policy successfully coordinates application activity to reliably run efficiently.

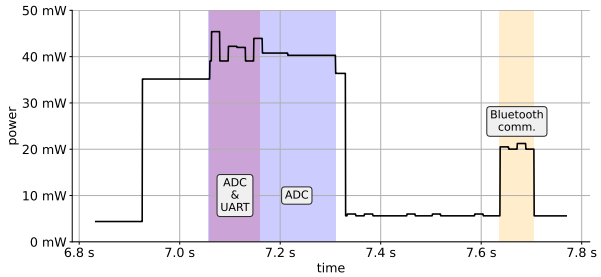

Figure 9: Power trace showing batching Bluetooth radio configuration (UART) and ADC sampling (ADC). Wireless communication is handled by a separate SoC and occurs outside the control of the system.

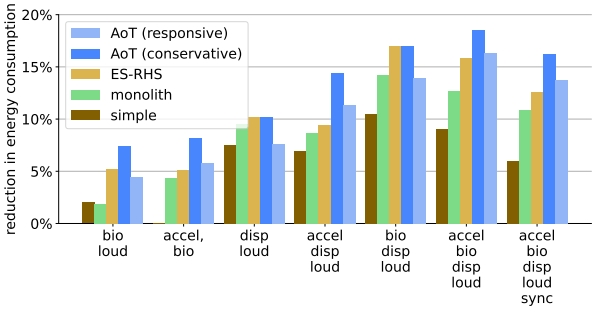

Figure 8: Energy consumption reduction (relative to no batching) of different batching policies when running different sets of applications.

6.2 Energy consumption reduction

We compare the energy consumption of several application sets running with different batching policies to gauge the effectiveness of our batching system. We also separately evaluate the benefit of AoT batching to energy consumption.

6.2.1 Energy efficiency of application sets. We first compare the energy efficiency of the batching implementations by installing different sets of applications (described in Section 6.0.1) on the evaluation device. The applications access peripherals at different rates. *Accel* accesses the accelerometer approximately every 750 ms. *Bio* reads temperature every two seconds and reads GSR every four seconds. *Disp* refreshes the display every second. *Loud* samples every three seconds. *Sync* communicates with a separate device every 20 seconds. We run each set with each batching policy in ten 5-minute trials and record energy consumption. For the baseline, we ran the same application sets on Tock 2.0 unmodified which does not perform batching. Using the baseline, we computed the reduction in energy consumption and present the results in Fig. 8.

Generally, running more applications offers more opportunity for reducing energy consumption. Energy efficiency trends upward going from two to four applications. Out of all policies, AoT performs the best in energy efficiency. ES-RHS matches AoT in efficiency except in application sets with latency-sensitive operations, like {accel, disp, loud}. The responsive AoT policy tends to outperform the simple strategy and the monolith, but the monolith sometimes achieves efficiency between the AoT policy variants.

Each of the two-application sets demonstrates how well batching improves energy efficiency in a simple scenarios of multi-tenancy. The {disp, loud} set is the simplest of the two-application sets to achieve energy reduction with. The batching policies must only align the display refresh with audio sampling, so all batching strategies attain notable energy efficiency. The {accel, bio} set shows that AoT is able to improve energy efficiency over the other approaches with the occurrence of latency-sensitive operations from *accel*.

The {bio, loud} set shows the effect of peripheral usage conflicts on energy efficiency achieved by each batching policy. Both applications use the ADC, resulting in a sequential usage of the ADC (only one sampling operation may occur at a time). Therefore, the ADC is active for longer and the second operation might not have another operation to run concurrently with. However, the AoT policy can avoid this conflict by sometimes pulling forward the samplings to make a batch with the encryption operation or temperature reading.

Notably, introducing *accel* to a set improves AoT's energy efficiency (e.g., {disp, loud} → {accel, disp, loud}), though the improvement between {bio, disp, loud} and {accel, bio, disp, loud} is less impactful. While other strategies allow the accelerometer query to run by itself, AoT pre-executes another operation concurrently, as long as one is eligible. This allows AoT to maintain a higher mean batch size. When applications do not issue latency-sensitive operations, AoT performs similarly to ES-RHS. We further explore the contribution of pre-execution to energy efficiency in Section 6.2.2.

Hail uses a separate SoC to perform Bluetooth connectivity. Software running on the main MCU has limited control over the wireless chip's activity compared to solutions that integrate the hardware into a single MCU. However it is possible to batch application- or driver-initiated operations (Fig. 9). We include *sync* with all other applications to show the effect on energy efficiency. We have an external Bluetooth-enabled device connect to and exchange data with the evaluation device and then disconnect, as a phone would with a wearable device. All strategies lose efficiency, but AoT achieves an approximate 15% efficiency improvement.

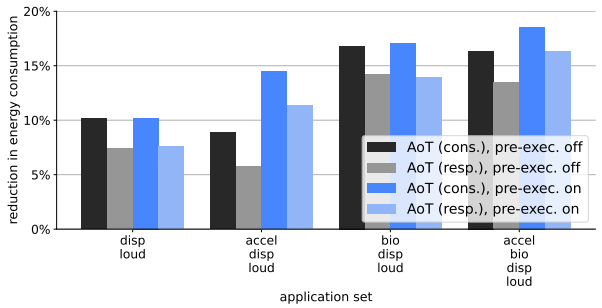

Figure 10: Reduction in energy consumption (relative to no batching) achieved by AoT with and without pre-execution. Pre-execution provides an improvement in energy efficiency after adding the latency-sensitive accelerometer application. Without it, there is a decrease in energy efficiency. The pre-execution aspect of AoT improves energy efficiency when running latency-sensitive applications.

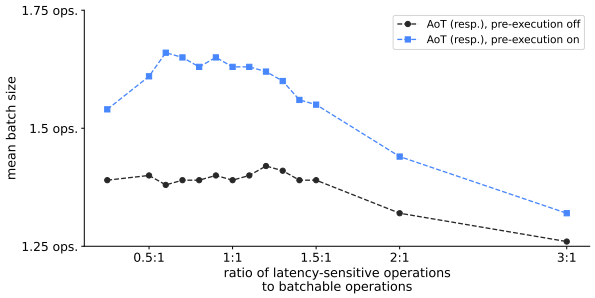

Figure 11: Mean batch size of AoT with pre-execution on and off running {disp, loud} and a synthetic application issuing aperiodic latency-sensitive operations. As the ratio of latency-sensitive to batchable operations increases, the gap in mean batch size decreases but full AoT maintains larger mean batch sizes. Full AoT maintains a higher mean batch size and outperforms AoT with no pre-execution. At higher ratios, the advantage of pre-execution decreases.

6.2.2 Benefit of ahead-of-time batching. Ahead-of-time batching enables energy-efficient execution of latency-sensitive operations. However, if the frequency of latency-sensitive events outpaces the frequency of batchable operations, the likelihood of batching at least two operations decreases. Knowing what benefit the pre-execution aspect of AoT offers and when that benefit applies helps understand what use cases it is best suited for. We now compare the efficiency of the AoT policy with a variant of itself that does not pre-execute operations to isolate the benefit of pre-execution. This variant is equivalent to the hybrid policy (Section 4.4).

We run {disp, loud}, {accel, disp, loud}, {bio, disp, loud}, and {accel, bio, disp, loud} and provide the results in Fig. 10. The policies with and without pre-execution perform similarly with {disp, loud} and {bio,disp,loud}. But with pre-execution, the conservative AoT policy improves energy efficiency going from {disp, loud} to {accel, disp, loud} (10.2% → 14.5% for AoT cons.) and gives a less significant gain from {bio, disp, loud} to {accel,bio,disp,loud} (17.0% → 18.6% for AoT cons.). Without pre-execution, energy efficiency drops going from {disp, loud} to {accel, disp, loud} and from {bio, disp, loud} to {accel, bio, disp, loud}. Both of these cases introduce *accel* to the set which adds a latency-sensitive operation to the system. This shows that the benefit from using the AoT policy applies when the system must handle latency-sensitive events; there is little difference in efficiency between pre-execution on and off in sets without *accel*.

To explore why AoT does not improve efficiency between {bio, disp, loud} and {accel, bio, disp, loud} as much as between {disp, loud} and {accel, disp, loud}, we performed an experiment to see how the frequency of latency-sensitive events affects batch size (and consequently, energy efficiency). We deployed {disp, loud} alongside a third application that issues timer interrupts at an approximate rate with ±5% jitter and reads the accelerometer. By varying the interval of the timer and tracking mean batch size, we obtain Fig. 11. As the rate of the latency-sensitive operations increases, the ratio of latency-sensitive to batchable operations also increases.

In this setup, AoT without pre-execution consistently achieves smaller batches. If an accelerometer operation arrives after the display refresh, both policies can batch them together. However, if the accelerometer operation arrives before the display refresh, the policy without pre-execution must execute it alone. Meanwhile, the full AoT policy maintains a higher mean batch size by sometimes pulling forward the ADC sampling operation. As the ratio increases, AoT maintains an advantage, but at much higher ratios, it will behave similarly with neither being able to keep up with the rate of latency-sensitive operation requests. If the rate of latency-sensitive operations far exceeds that of batchable operations, there is only a small additional benefit to energy efficiency.

activity	application behavior change from "default"
default	display updates every 10 s loudness metering every 5 min. temperature reading every 3 min. GSR readings every 10 min. send data via Bluetooth every 5 min.
sleeping	display updates disabled loudness metering every 10 min. temperature reading every 10 min. send data via Bluetooth every 30 min.
sitting	accelerometer event approx. every 30 s GSR readings every five minutes
walking	accelerometer event approx. every 800 ms loudness metering every one minute GSR readings disabled temperature readings disabled send data via Bluetooth every 30 sec.
eating	accelerometer event approx. every 5 s

Table 3: Variation in application behavior (Section 6.2.3). "Default" describes all application behavior under all activities. Other activities describe behavior changes from "default."

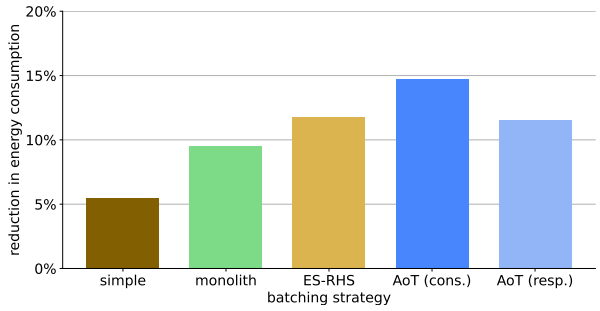

Figure 12: Energy consumption reduction of batching strategies when running applications that change the frequency of their activity over time.

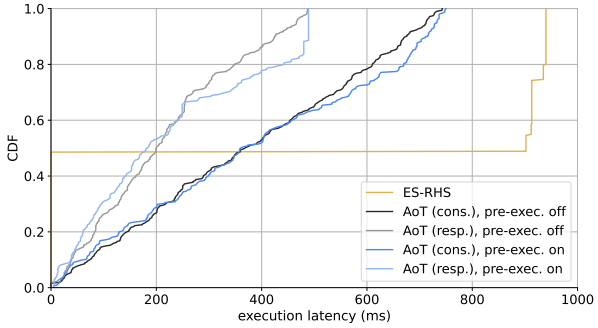

Figure 13: Batching latency while running the {accel, bio, disp, loud} set. Handling aperiodic operations causes the CDF for AoT to take on a nearly uniform distribution.

6.2.3 Dynamic applications. Embedded applications can exhibit dynamic behavior: starting or stopping their periodic activity, changing the periods of their activity, or changing which peripherals they use. To evaluate the effectiveness of the system in a more complex scenario, we alter the behavior of the wearable applications according to traces of a participant's everyday activity in the labeled open wrist activity dataset, CAPTURE-24 [12]. We use six hours of data that includes a total of: 70 minutes of sleeping, 128 minutes of sitting or office work, 46 minutes of eating, 47 minutes of walking, 46 minutes of eating, and 22 minutes of household chores. These activities are not contiguous. Applications modify their behavior according to the current activity. Table 3 details their behavior.

We run the applications in six-hour trials with each batching policy and compare their efficiency in Fig. 12. The result is consistent with the experiment using application sets. AoT maintains the best efficiency among these approaches when using the conservative window, and it achieves efficiency similar to ES-RHS when using the shorter, responsive window. This shows that the batching system and AoT can adapt to a common, dynamic use case.

6.3 Impact to applications

Although the AoT policy can execute latency-sensitive operations efficiently, it delays other batchable requests. To understand the impact the policy has on applications, we characterize the latency imposed on individual operations and the overhead latency of the batching logic. We instrument Tock and the batching code to profile the system and deploy the {accel, bio, disp, loud} set.

Figure 13 gives the CDF for latency experienced by operations withheld by batching under AoT and ES-RHS policies. This latency includes the time spent awaiting execution and the time the batch controller spends processing the operation (a small fraction of the total time). In this experiment, the batching window durations for conservative AoT and ES-RHS are equal at 1 second. The batching window duration for responsive AoT is 520 ms.

The CDF for AoT is approximately uniform, notably different from ES-RHS. This is due to the aperiodic operations for *accel* causing batches to also execute aperiodically. This occurs both with the full AoT policy and the AoT policy without pre-execution. This implementation of ES-RHS immediately executes latency-sensitive operations, leaving batched operations unaffected.

Though both policies upper-bound delay, AoT imposes a less predictable delay than ES-RHS. ES-RHS dictates a harmonizing interval, and as long as an application does not change its scheduling period, its operation will experience the same delay before execution. However, because AoT makes use of aperiodic events, batches may execute earlier than its window duration. The delay will be between zero and the duration of the window.

A fraction of time in these measurements is the time it takes for the kernel and batch controller to handle the peripheral operation. Every syscall passes through the batch controller, so every syscall will have this time added to its execution. When the AoT batch controller transparently passes a syscall, it adds about 3.15 µs to its execution time. When the AoT batch controller withholds a syscall, it adds 126 µs before returning to the application.

Summarily, AoT trades off predictability in delay to accommodate latency-sensitive operations and to improve energy efficiency. This has the additional effect of improving the latency of batched operations over an approach which does not consider latency-sensitive operations for batching, though this is dependent on the expected frequency of aperiodic events compared to periodic events.

6.4 System overhead

The batching system increases binary size and memory usage. Since embedded hardware varies widely, what may be a small cost to one device may overwhelm another. We evaluate overhead with respect to our evaluation platform, which has 64 KB of SRAM and 512 KB of storage. These specifications are representative of low-power embedded systems. Table 4 gives the code size of different batching implementations. The AoT policy is the largest, requiring an additional 1,956 B. At runtime, AoT uses 304 B of RAM, ES-RHS uses 200 B, the fixed time window uses 204 B, and simple batching uses 8 B. Hardware specifications in the embedded space varies, but our evaluation platform easily accommodates these changes.

7 Discussion

Sensor- and peripheral-rich platforms, like wearables and building monitoring infrastructure, can benefit greatly from controlling energy usage through peripheral operation batching. We specifically demonstrate its applicability to a low-power wearable platform, but

implementation	add. code size	total binary size
simple batching	896 B	114.7 KB (+0.78%)
time window batching	880 B + 592 B	116.5 KB (+1.55%)
ES-RHS	880 B + 608 B	116.6 KB (+1.70%)
AoT batching	880 B + 1076 B	116.6 KB (+1.71%)

Table 4: Code size increases for batching. For the time window and AoT policies, the first size is due to the batching framework; the second is due to the policy implementation.

any device with a breadth of sensing, processing, or communication hardware can leverage this technique effectively.

The ahead-of-time policy we present can serve a general-purpose use case, but two scenarios we do not cover in depth that may be important for system designers to consider are peripheral usage conflicts and redundant requests. In case of a conflict, a policy could retry on behalf of applications or determine which to serve based on some priority. And if applications submit identical requests, a policy could make a single request to the peripheral driver and return the result to both applications. This more sophisticated handling would come at the cost of additional code and memory usage.

Exerting system-level control for batching presents possibilities for additional research directions. While embedded applications tend to have simple workflows, factors other than time or a single event can trigger peripheral-bound activity. Applications may depend on the activity of others or implement more dynamic execution workflows. A deeper analysis of application behavior could improve both energy efficiency and latency in these cases. Additionally, considering other properties—such as peripheral operation duration or application priority—and accordingly adapting window durations or adding entirely new policy rules could improve energy efficiency, latency, or feasibility of batching for particular use cases.

8 Conclusion

As the importance of low-power embedded systems grows and multi-tenancy expands use cases, energy management on these devices must continue to improve for sustainable operation. The batching system we present coordinates applications in multi-tenant systems to control energy usage, enabling extended operation or expanded utility. Its adaptability makes it suitable for a variety of embedded hardware, and its system-level operation means applications remain unencumbered with the mechanics of batching. It can serve many use cases well, but having a deeper understanding of application behavior may serve more complex applications. This system-level energy management technique can provide a strong foundation to build on. To encourage collaboration and repeatability, we make our implementation and evaluation artifacts public upon publication.

Acknowledgments

We thank the anonymous reviewers and shepherd for their feedback that improved this paper. This work is supported by the National Science Foundation under Grant Nos. 1829004 and 2144940, as well as the Semiconductor Research Corporation (SRC) project HWS-3127.001.

References

[1] 2020. *Introducing Hail.* Retrieved June 22, 2023 from https://tockos.org/blog/2017/introducing-hail/

[2] 2023. *INA219 data sheet, product information and support.* Retrieved June 22, 2023 from https://www.ti.com/product/INA219

[3] 2023. *Raspberry Pi.* Retrieved June 22, 2023 from https://www.raspberrypi.com

[4] 2023. *STM32F446 - STMicroelectronics.* Retrieved June 22, 2023 from https://www.st.com/en/microcontrollers-microprocessors/stm32f446.html

[5] 2023. *STM32N6: Get a sneak peak at the future of AI-powered MCUs.* Retrieved June 18, 2024 from https://blog.st.com/stm32n6/

[6] 2024. *2.13inch e-Paper HAT Manual - Waveshare Wiki.* Retrieved May 30, 2024 from https://www.waveshare.com/wiki/2.13inch_e-Paper_HAT_Manual

[7] 2024. *Garmin Venu 3 | Fitness and Health Smartwatch.* Retrieved June 17, 2024 from https://www.garmin.com/en-US/p/873008

[8] 2024. *Oura Ring. Smart Ring for Fitness, Stress, Sleep & Health.* Retrieved June 14, 2024 from https://ouraring.com/

[9] 2024. *Overview of 1-Wire Technology and Its Use.* Retrieved June 18, 2024 from https://www.analog.com/en/resources/technical-articles/guide-to-1wire-communication.html

[10] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing. In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 188–199.

[11] Sudershan Boovaraghavan, Chen Chen, Anurag Maravi, Mike Czapik, Yang Zhang, Chris Harrison, and Yuvraj Agarwal. 2023. Mites: Design and deployment of a general-purpose sensing infrastructure for buildings. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 7, 1 (2023), 1–32.

[12] Shing Chan, Yuan Hang, Catherine Tong, Aidan Acquah, Abram Schonfeldt, Jonathan Gershuny, and Aiden Doherty. 2024. CAPTURE-24: A large dataset of wrist-worn activity tracker data collected in the wild for human activity recognition. *Scientific Data* 11, 1 (2024), 1135.

[13] Sami Davies, Samir Khuller, and Shirley Zhang. 2022. Balancing flow time and energy consumption. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 369–380.

[14] Analog Devices. 2001. *Using Power Management with High Speed Microcontrollers.* Retrieved February 21, 2025 from https://www.analog.com/en/resources/technical-articles/using-power-management-with-highspeed-microcontrollers.html

[15] Prabal K Dutta and David E Culler. 2005. System software techniques for low-power operation in wireless sensor networks. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.* IEEE, 925–932.

[16] Vikram Gupta, Eduardo Tovar, Karthik Lakshmanan, and Ragunathan Rajkumar. 2011. A framework for programming sensor networks with scheduling and resource-sharing optimizations. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, Vol. 2. IEEE, 37–40.

[17] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 234–251.

[18] Nima Nikzad, Octav Chipara, and William G Griswold. 2014. Ape: An annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*. 515–526.

[19] Linwei Niu and Gang Quan. 2007. Peripheral-Conscious Scheduling on Energy Minimization for Weakly Hard Real-time Systems. In *2007 Design, Automation and Test in Europe Conference and Exhibition*. 1–6. https://doi.org/10.1109/DATE.2007.364387

[20] Padmanabhan Pillai and Kang G. Shin. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 89–102. https://doi.org/10.1145/502059.502044

[21] Anthony Rowe, Karthik Lakshmanan, Haifeng Zhu, and Ragunathan Rajkumar. 2008. Rate-harmonized scheduling for saving energy. In *2008 Real-Time Systems Symposium*. IEEE, 113–122.

[22] Anthony Rowe, Karthik Lakshmanan, Haifeng Zhu, and Ragunathan Rajkumar. 2010. Rate-harmonized scheduling and its applicability to energy management. *IEEE Transactions on Industrial Informatics* 6, 3 (2010), 265–275.

[23] Arjun Roy, Stephen M Rumble, Ryan Stutsman, Philip Levis, David Mazieres, and Nickolai Zeldovich. 2011. Energy management in mobile devices with the cinder operating system. In *Proceedings of the sixth conference on Computer systems*. 139–152.

[24] Ruibin Xu, Daniel Mossé, and Rami Melhem. 2007. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Trans. Comput. Syst.* 25, 4 (Dec. 2007), 9–es. https://doi.org/10.1145/1314299.1314300