

Power Clocks: Dynamic Multi-Clock Management for Embedded Systems

Holly Chiang*, Hudson Ayers*, Daniel Giffin*, Amit Levy[†], Philip Levis*

*Stanford University, [†]Princeton University

{hchiang1@, hayers@, dbg@scs.}stanford.edu, aalevy@cs.princeton.edu, pal@cs.stanford.edu

Abstract

This paper presents Power Clocks, a kernel-based dynamic clock management system that reduces active energy use in embedded microcontrollers by changing the clock based on ongoing computation and I/O requests. In Power Clocks, kernel hardware drivers asynchronously request clocks, providing a set of constraints (e.g., maximum speed), which the kernel uses to dynamically choose the most efficient clock. To select a clock, Power Clocks makes use of the observation that though slower clocks use less power and are suited for fixed time I/O operations, faster clocks use less energy per clock tick, making them optimal for pure computation. Using Power Clocks, a networked sensing application consumes 27% less energy than the best static clock, and within 3% of an optimal hand-tuned dynamic clock strategy. Power Clocks provides similar energy savings even when there are multiple applications.

1 Introduction

Cheap microcontrollers (MCUs) make a variety of sensing applications possible. Medical sensors monitor patients' vital signs, and notify doctors of anomalies. Utility sensors monitor and control water, gas and power systems. Sensors find myriad other uses in transportation, public safety, and agriculture [25]. In many of these applications, deployed sensors lack access to power sources, while cost or other constraints limit battery sizes. Accordingly, energy efficiency defines the lifetime of many sensor systems, and application developers frequently make trade-offs between performance and application lifetime.

To conserve energy, sensor applications spend most of their time in deep sleep. They punctuate their sleep with periods of I/O and computation to gather, process, and communicate data. These brief active periods consume most of the system's energy. Low-power operation during active pe-

riods centers on minimizing the energy consumed executing CPU instructions and powering on-chip peripherals such as buses, sensors, and flash controllers.

Low-power microcontrollers have begun to introduce a new mechanism for energy efficiency: clock selection. These microcontrollers [27, 29, 16] provide multiple clock sources which vary in their frequency, power draw, and startup times. Different clocks can vary in current draw by more than an order of magnitude (e.g., 1.14mA vs. 26.70mA on an Atmel SAM4L): clock choice is an important component of system energy consumption. Which clock is optimal depends on whether ongoing peripheral operations are CPU or I/O bound.

Interestingly, and inverting the tradeoffs of dynamic frequency and voltage scaling (DVFS) in traditional processors, in these microcontrollers *fast clocks are more energy-efficient for CPU operations*. Slow clocks, in turn, are more efficient for I/O operations. In DVFS systems, a slower clock allows a lower voltage and therefore lower power; in microcontrollers, however, there is no voltage scaling and the clock itself is a significant energy cost. A faster clock consumes more power, but amortizes this cost over more CPU cycles: it draws *more power but less energy per CPU cycle*.

Choosing the most energy efficient clock requires satisfying many constraints and tradeoffs. Many peripherals have specific clock requirements, which means choosing a clock requires hardware-specific knowledge of each peripheral used. Additional energy savings can be achieved by dynamically changing the clock in response to application state, but doing so while ensuring the correct functionality of all ongoing peripheral operations is challenging. To avoid this complexity, applications often pick a single static clock that ensures correct functionality for all peripherals, sacrificing the potential for efficiency gains from dynamic clock

This paper introduces Power Clocks, a kernel subsystem that dynamically chooses a microcontroller's system clock in response to computation and I/O requests. With Power Clocks, kernel hardware drivers asynchronously request clocks, providing a set of constraints (e.g., maximum speed), which the kernel uses to dynamically choose the most efficient clock. The key insight Power Clocks uses is distinguishing between low power (cost over time) and low energy (cost per tick) clocks, and selecting which to use based on clock constraints of active and waiting peripherals.

Power Clocks is implemented in the Tock operat-

ing system [20] and on two microcontrollers, the Atmel SAM4L [27] and the NXP K66 [16]. The energy savings provided by Power Clocks are evaluated using two example sensor applications. We find that using Power Clocks, a sensing application can consume 27% less energy than the best static clock, and only 2% more than an application-specific kernel that dynamically changes the clock. For multiprogrammed OS kernels, Power Clocks can provide similar energy savings when there are multiple independent applications, an otherwise impossible task.

2 Choosing Optimal Clocks

A clock is a simple signal that goes between 0 and 1. The clock’s high frequency and high load, from driving all synchronous elements on the chip, means that the clock consumes a significant amount of power. The clock can be responsible for up to 40% of dynamic power consumption in synchronous digital circuits [4]. This number can be even more significant in low power microcontroller applications.

Modern Cortex-M MCUs allow software to choose between a number of clocks to drive the CPU, buses, and other peripherals. In addition to varying in frequency by orders of magnitude (115kHz to 180MHz on the SAM4L), these clocks can differ in type (RC oscillator, crystal oscillator, PLL/FLL). This allows the MCU to meet clock requirements ranging from specific frequencies, to fast startup times, to temperature stability. This section focuses on the trade-off between power and energy that different clocks provide, and introduces the benefits of and constraints on dynamically changing the clock.

2.1 Power vs Energy

Modern microcontrollers have sleep and deep sleep modes where power draw is minimal compared to active periods (μA vs mA). On Cortex-M MCUs, sleep modes stop the processor clock while deep sleep stops all high frequency clocks and disables most non-essential peripherals [11].

During active periods, reducing energy use usually means lowering power draw, or the rate at which energy is consumed. On CMOS chips, power can be divided into dynamic and static power. Dynamic power is dissipated each time a signal toggles, causing capacitors to charge and discharge. Static power is dissipated by current leaking through powered transistors, even when they are not actively switching. While static power is an increasing concern with smaller CMOS processes, dynamic power still dominates for microcontrollers. Dynamic power scales linearly with clock frequency, while static power is independent of clock frequency. Accordingly, a circuit’s power draw is a nearly linear function of frequency, with a significant constant component [3]. Slower clocks are *low power* since they draw less dynamic power, and therefore less power at a given instant. Faster clocks are *low energy*: as frequency increases, static power is amortized over more clock ticks, minimizing the energy per clock tick.

The typical sensor node consists of a sensing system to gather data, a processing system to locally process data, and a wireless communication system to transmit data [28]. Sensing and communication operations are usually I/O bound, while processing is compute bound. I/O bound op-

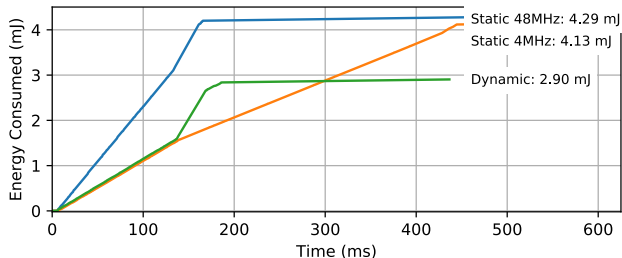


Figure 1: Cumulative energy used over time by an example sense-process-send application. Optimal dynamic clock management provides 32% energy savings over the best static alternative.

erations take a fixed amount of time regardless of the clock used; a faster clock does not lead to faster completion. To minimize energy, the low power clock should be used. Compute bound operations complete faster when a faster clock is used. To minimize energy, the low energy clock should be used. Since the low energy clock is effectively the fastest clock, compute operations are energy-optimal when they “race to idle” - they run at the fastest frequency in order to return to sleep as quickly as possible [3].

Many more powerful processors support DVFS, which aims to reduce the energy consumption on a single clock by reducing the clock’s frequency in order to perform voltage scaling. This paper focuses on systems which support only a fixed voltage. Further discussion of why the techniques described in this paper are fundamentally different from DVFS techniques can be found in Section 6.

2.2 Dynamic Clock Change Benefits

Minimizing energy consumption requires using the right system clock. Systems today typically use one of two algorithms to manage clock energy: they use a slow, low power static clock or a fast, low-energy static clock (race-to-idle). Systems using a slow static clock choose the lowest power clock that is fast enough for all peripherals in a given workload. This algorithm is easy to implement and safe but wastes energy during compute-bound operations. Race-to-idle, in contrast, always uses the fastest clock so as to minimize static power consumption by entering deep sleep sooner. This is energy-efficient for compute operations, but can waste energy if there is I/O: using a 48MHz clock rather than a 4MHz bus uses more power without reducing runtime.

Figure 1 shows these trade-offs for an example sense-process-send application: a static 4MHz clock is more efficient during sensing (the first 150ms), while a static 48MHz clock is more efficient for processing data. Relative to sense and process, the send operation is short enough that its energy usage is relatively unnoticeable on the graph. The sense operation is I/O bound and lasts the same length regardless of clock used, while the processing data operation is compute bound and takes significantly longer with the slower 4MHz clock. Dynamically changing the clock to use low-power clocks for I/O bound operations, and low-energy clocks for

computationally intensive operations, gives the best of both worlds, allowing the application to run almost as quickly as if the static 48MHz clock was used, while reducing energy consumption by 30-32% over the static clock choices.

2.3 Clock Change Constraints

Dynamic clock selection is necessary to ensure energy-optimal clocks are used for different operations. However, peripheral requirements must be considered at each opportunity for a clock change.

2.3.1 Clock Selection

Even when many clock options are available, the choice of clock is often constrained, with the CPU, buses, and other peripherals each having their own clock requirements. For example, a UART bus might need to operate at a specific frequency (e.g. baud rate of 115200), or chips connected to a SPI bus may limit how fast the bus can run (e.g. 1MHz, 8MHz). The ADC may require a minimum frequency in order to achieve its desired sample rate. Additionally, a peripheral’s clock requirements are often specific to its microcontroller.

A single clock cannot match the requirements of every peripheral, so most peripherals have a divisor that allows them to divide a clock to a lower rate. This is typically just a counter that triggers the peripheral clock after every n ticks of the input clock. For example, a bus with a maximum clock speed of 1MHz can be driven by an 8MHz clock after the bus divides the input clock by a factor of 8. However, this draws more power than using a 1MHz clock with no divider (the 8MHz clock draws more power).

It is not always possible to just use a faster clock. Peripherals may require specific clocks for reasons such as accuracy, or may only have access to a finite or integer divider.

2.3.2 Clock Change

In addition to choosing which clock is optimal for a given set of ongoing operations, a clock management system must also consider *when it is safe* to change clocks. Some peripheral operations can function correctly across a clock change, but others require a constant frequency while they execute.

Some peripherals, such as SPI and I2C, have a clock line that indicates when a receiver should read from the data line. These peripherals can tolerate frequency changes while an operation is ongoing because the clock line synchronizes the receiver to the sender: if a clock edge is delayed by a few microseconds, or even milliseconds, the receiver can still read data correctly. However, a SPI or I2C bus may be used to communicate with an external chip, which can have a lower maximum clock frequency than the main microcontroller. If this is the case, and the frequency on the bus’s clock line becomes higher than the maximum frequency of the external chip, then communication with the external chip will fail.

Other peripherals, such as the UART and ADC, rely on precise timing and can malfunction if the clock jitters. A clock change can corrupt a UART data transmission by sending at the wrong speed so that a receiver reads the wrong message. Similarly, changing the clock while an ADC is sampling can cause it to sample at the wrong rate. To ensure correctness, a system must not change the clock while any such timing-sensitive operations are in progress.

Table 1: The Power Clocks API. Each driver implements the ClockClient interface and informs the ClockManager by calling `register_client`, which calls the ClockClient’s `client_registered` in return.

	<code>register_client(client: &ClockClient)</code>
	<code>set_max_frequency(freq: u32)</code>
	<code>set_min_frequency(freq: u32)</code>
	<code>set_clocklist(clock_list: u32)</code>
ClockManager	<code>no_jitter(can_jitter: bool)</code>
	<code>make_clock_request()</code>
	<code>release_clock_request()</code>
	<code>client_registered()</code>
ClockClient	<code>update_clock_configs(new_f: u32)</code>
	<code>clock_request_granted()</code>

3 Power Clocks

Since applications can save energy through dynamic clock management, how should it be done? Performing dynamic clock management in the application is challenging: interleaving clock management code with application logic is likely to introduce bugs and makes it harder to port applications between chips. Furthermore, in the case of multiple applications, information would have to be shared between applications to coordinate clock changes, and the number of possible state combinations multiplicatively increases with each additional application. Instead, dynamic clock management should be handled in the kernel.

3.1 Design Goals

The ultimate goal of a clock management system is to make the device more energy efficient. However, as we saw in Section 2.3, there can be many peripheral-specific constraints on choosing the clock, and careless clock changes can result in incorrect peripheral behavior. From these points we propose the following design goals for a dynamic clock management system:

Efficiency: Achieve significant energy savings over the best static clock choice, be near optimal approach.

Flexibility: All peripherals should be able to precisely specify their clock requirements.

Correctness: Peripherals must continue to operate correctly after a clock change.

Notably, meeting timing deadlines is explicitly not a goal. This paper focuses on sensornet type systems with lax timing constraints, not control systems with real-time deadlines.

3.2 System Architecture

Power Clocks is a kernel API and subsystem that dynamically changes the clock in an energy-efficient manner. Rather than deferring global and chip-specific clock management decisions to applications, Power Clocks chooses clocks dynamically using aggregated *local* information from drivers.

Power Clocks imposes a split-phase [12] API between the ClockManager and ClockClients, as shown in Table 1. A centralized component, called the ClockManager, chooses the most energy-efficient clock, subject to the clock constraints of drivers with active or waiting operations. When

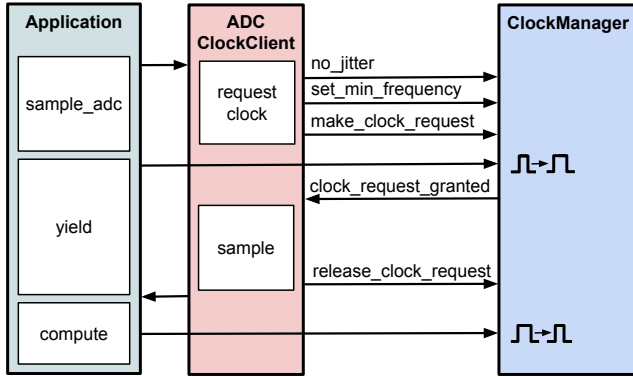


Figure 2: Example interaction of a ClockClient (the ADC driver) with the ClockManager. When an app requests ADC samples, the ADC driver interacts with the ClockManager

choosing the clock, the ClockManager also takes into account whether the current workload is compute or I/O bound.

Each peripheral driver serves as a ClockClient, and registers itself with the ClockManager during initialization. Each client informs the manager of its clock requirements before performing any operations that require a clock. Requirements can include the minimum and/or maximum frequency permitted for an operation, whether the peripheral can tolerate a clock change mid-operation, and optionally a specific list of allowed clocks. When the active clock is compatible with all of a client's requirements, ClockManager issues a callback to the client, signaling that it can begin.

3.3 Example Execution

Figure 2 shows an example of Power Clocks in action. An application requests ADC samples from an ADC driver that has implemented ClockClient. The client first saves the request parameters for use in the callback. Next, it reports the ADC's clock requirements to the ClockManager. These requirements include the minimum clock frequency it needs to achieve its desired sample rate (`set_min_frequency`) and that it cannot handle clock changes mid-operation (`no_jitter`). The client calls `make_clock_request`, asking the ClockManager for a clock that meets its reported requirements, and returns to wait for a compatible clock.

When the ClockManager is ready to change the clock, as described in Section 3.6, it uses the process discussed in Section 3.5 to choose the next clock. After the clock is changed, the ClockManager issues a callback to the client (`clock_request_granted`), signaling that it can begin operation using its saved parameters. Because the ADC client specified that it was a no-jitter peripheral, the ClockManager will not change the clock until the client releases this constraint, regardless of requests from other clients.

Once the client finishes ADC sampling, it releases its requirements (`release_clock_request`). With only compute operations left, the ClockManager races to sleep, switching to the low-energy clock to complete remaining computations so the chip can enter deep sleep.

3.4 Meeting Design Goals

The API between ClockManager and its clients enables Power Clocks to achieve its three design goals.

3.4.0.1 Efficiency

Power Clocks implements the energy-efficient strategy of using the low-power clock when there are any I/O bound operations, and using the low-energy clock when there are only compute bound operations. When there are both I/O and compute bound operations, the most energy efficient clock is the slowest clock that allows computation to finish before I/O operations. For such cases, Power Clocks uses the I/O operations' low-power clock because applications designed for low power nodes rarely overlap lengthy computation with I/O. Power Clocks tracks active I/O bound operations through peripheral calls to `make_clock_request` and `release_clock_request`. Section 5 evaluates Power Clocks's energy efficiency.

Power Clocks addresses the challenge of clock changes incurring time and energy overheads by waiting for a set time quanta before switching to the low-energy clock for compute operations. This delay reduces the energy wasted by always changing the clock for short computations.

3.4.0.2 Flexibility

Power Clocks gives clients flexibility in requesting clocks. Peripherals' clock requirements often vary per operation. For instance, the SAM4L's SPI generates its baud rate using a divisor between 1 and 255 to divide the clock. In order to achieve a 1,000,000 baud rate, the SPI needs a clock frequency between 1MHz and 255MHz. `set_max_frequency` and `set_min_frequency` allow peripherals to express their clock requirements as a range of allowed frequencies. Peripherals may also require specific clocks. For instance, the ADC may want to only use clocks that produce accurate sample times regardless of environmental conditions. `set_clocklist` would allow the ADC to limit its list of allowable clocks to those with higher stability.

3.4.0.3 Correctness

Ensuring correct peripheral operation requires allowing peripherals to adjust their clock related configurations prior to clock changes. Power Clocks does so through `update_clock_configs`, which the ClockManager calls on ClockClients with active operations, before the clock change if the new clock is faster, and after the clock change otherwise.

No-jitter peripherals such such as UART or ADC, which can malfunction if the clock frequency changes during an active operation, report their requirement with the `no_jitter` method. ClockManager will not change the clock unless all active clients support jitter.

3.5 Clock Selection Algorithm

If a client calls `make_clock_request` and its clock requirements are compatible with the current clock, the manager issues a `clock_request_granted` callback only if doing so will not further delay outstanding requests. This prevents starvation, which happens when a series of new requests cause an outstanding request to delay indefinitely.

A new request does not delay pending requests if two conditions hold. First, the requesting client cannot have `no_jitter` set: otherwise, it will lock in the current clock, prevent-

ing pending requests from changing the clock. Second, there must exist a clock that satisfies both the requesting client and all pending clients. This ensures that when the clock can be changed, if the requesting client is still active, the ClockManager can change the clock to one that services all of the pending requests. If a client's request does not meet these two conditions, the ClockManager enqueues it.

The request queue is a FIFO queue. When a clock change occurs, the ClockManager starts at the head of the queue and iterates through the requests, finding the intersection of each request's clock requirements with that of requests before it. The first request finds the intersection of its clock requirements with that of clients with active operations, ensuring that whatever clock is chosen will not interfere with their operations. If a request causes the intersection to be zero, the ClockManager skips it.

If the final intersection has multiple clocks, ClockManager chooses the lowest power one. If the request queue is empty when a clock change occurs, it selects the low energy clock. All of the pending requests that are not serviced keep their ordering in the queue: the first (oldest) request with zero intersection will become the first request in the queue and receives the highest priority.

3.6 Clock Change Conditions

At a high-level, Power Clocks aims to choose the most energy efficient clock at each instant of operation. However, eagerly changing clocks immediately in response to requests from ClockClients is sub-optimal because of two phenomena: *lockout* and *thrashing*. To address both of these, Power Clocks takes advantage of typical characteristics of low-power applications.

Lockout can occur when a client requires the `no_jitter` constraint, preventing clock changes while the client's operation is outstanding, and when subsequent requests have only partially-overlapping clock requirements. If the ClockManager uses a greedy algorithm, it might unnecessarily commit to a clock incompatible with subsequent operations, forcing them to be serialized and prolonging the MCUs active time.

For example, if an application initiates an ADC operation—which cannot tolerate clock changes and needs at least a 1MHz clock—immediately followed by an I2C operation—which needs at least a 4MHz clock—the ClockManager could commit to the 1MHz clock for the ADC operation and be forced to delay the I2C operation until the ADC completes. In this example, it would be more time and energy efficient to wait until both the ADC and I2C clients have submitted their clock requests before changing the clock. This would cause the ClockManager to choose the 4MHz clock, allowing both operations to occur in parallel.

To address such cases, Power Clocks waits to make more informed clock choices. Power Clocks assumes an asynchronous operation model (typical for low-power systems) where applications initiate I/O operations asynchronously and are notified of completion through callbacks after yielding their thread of execution. Power Clocks initiates a clock change once all applications are yielded, as this indicates that all I/O requests have been made. At this point, the ClockManager can make an optimal clock choice based on a global view of concurrent I/O operations. While this is not a perfect

solution, it works well for Power Clocks because it allows parallelizing peripheral operations despite no-jitter peripherals preventing clock changes. This strategy can also save energy by reducing the number of needed clock changes.

Thrashing is the result of frequently switching between clocks when the CPU is active for short bursts between I/O operations. Because clocks take time to stabilize, there is a fixed energy cost to clock changes that often outweigh the benefit of short bursts of more energy-efficient operations.

Power Clocks reduces the overhead of thrashing by using application time quantum expirations as a signal that the application is running a long compute bound operation. When the time quantum expires, if there are no I/O bound clients, Power Clocks changes to the low-energy clock. This heuristic is based on the observation that CPU utilization in low-power embedded systems is highly bimodal: it either is very short (e.g. lightweight I/O processing) or very long, taking hundreds of milliseconds (e.g. occasional signal processing or cryptographic operations).

3.7 Limitations

While Power Clocks performs well for the typical sense-process-send application, there are several edge cases for which other approaches may outperform it.

First, if a single static clock happens to be the most energy-efficient approach (e.g. if applications are almost entirely compute-bound), Power Clocks adds unnecessary overhead. Power Clocks is not useful for such applications.

Second, if a no-jitter operation never completes, Power Clocks will never change clocks. For instance, an application that samples the ADC continuously would prevent Power Clocks from ever changing the clock once the ADC starts. No-jitter peripheral operations can delay or even starve other peripheral operations waiting for a clock change. Though Power Clocks does not address continuously running operations that require a constant clock, energy-limited applications rarely have such peripheral operations.

Third, latency sensitive applications may fail to meet timing deadlines under Power Clocks. In addition to Power Clocks's code overhead, operations may experience latency waiting for the clock to change. Furthermore, peripheral driver code is processed by the CPU, so when a low power, slower clock is used to run a peripheral's I/O bound operation, some of that peripheral's driver code for things such as peripheral configuration or interrupt handling will also run on that slower clock. All this means that for an application using Power Clocks, tasks are likely to finish later than in a system using a fast static clock choice. There is a fundamental trade-off between minimizing power and maximizing throughput. Integrating real-time requirements into Power Clocks is an area of future work.

4 Implementation

Power Clocks is implemented in the Tock embedded operating system, which is designed to run multiple concurrent, mutually-distrustful applications on low-power microcontrollers [20]. Prior to this work, Tock used a single static clock.

Power Clocks is implemented on two microcontrollers: the Atmel SAM4L [27] on the imix development board, and

the NXP K66 [16] on the Teensy 3.6 development board. The SAM4L and K66 are chosen because they both have multiple clock sources that the user can directly select from, and support a range of peripherals that make clock choice nontrivial. The two clock systems also have differences that test Power Clocks’s ability to generalize to different architectures.

4.1 Tock

Tock’s kernel code is split into code that is portable across microcontrollers, and code that is specific to a chip or architecture. The ClockManager and ClockClient interfaces are in the portable section of the kernel. Adapting chip-specific drivers to work with Power Clocks requires at most a few dozen lines of code. This code informs the ClockManager of clock requirements and defers performing an operation until it receives a callback.

Power Clocks requires changes to Tock’s scheduler to have it notify the ClockManager when it’s time to change the clock. The scheduler requests a clock change in one of two situations. The first is just before the board goes to sleep (the scheduler is about to issue a wait-for-interrupt, or `wfi` instruction). This indicates that all applications have yielded and the full set of parallel I/O operations have been requested. At this point, the ClockManager chooses a clock based on the request queue, as described in Section 3.5.

The second situation is when an application has spent longer than a time quanta on computation, indicating long-running computation. The time quanta used by Tock is 10ms. When this happens, the scheduler notifies the ClockManager that the application is doing compute, and if the ClockManager determines there are no active peripheral operations, it changes to the low-energy (fast) clock.

4.2 Example Chip: SAM4L

The Atmel SAM4L has 64 kB of RAM and 512 kB of flash. The SAM4L has seven clocks that can drive the system clock, as shown in Table 2. Higher frequency clocks are more expensive (more current) but more efficient (less energy per clock tick).

Table 2: The current is measured with the SAM4L in RUN mode, where the CPU and peripheral clocks are on and the CPU is executing NOP instructions. Energy numbers are calculated assuming a 3.3V voltage source.

Clock	Freq (MHz)	Measured Current (mA)	Energy/ Clock Tick (pJ)
RCSYS	0.113600	1.14	33116
RC1M	1	2.90	9570
RCFAST	4.3	4.12	3162
	8.2	5.48	2205
	12	6.36	1749
OSC0	16	6.76	1394
RC80M	80	11.28	465
DFLL	20-150	6.56-22.80	1082-502
PLL	48-240	11.90-26.70	818-367

Some clocks are configurable. RCFAST can be configured to run at either 4, 8, or 12MHz. The PLL can be configured to a frequency in the 48-240MHz range, and DFLL

(Digital FLL) to a frequency in the 20-150MHz range. These extremely configurable frequency ranges can be useful if a driver requests a specific clock frequency that none of the constant frequency clocks can meet or be divided down to. None of the SAM4L peripherals or external peripherals on imix need this functionality, so our implementation does not use it. Handling such cases is future work, when we encounter use cases for them.

The CPU can run at frequencies up to 48MHz, so higher frequency clocks must be frequency divided before use. Therefore, though RC80M seems to have the lowest energy/tick cost at 465pJ/tick, its effective energy/tick cost is actually twice that, at 930pJ/tick, since it must be divided down to 40MHz before use. The PLL at 48MHz therefore has the lowest energy/tick cost, at 828pJ/tick. However, the PLL can take half a millisecond to start up, such that the energy overhead of changing to the PLL makes it inefficient for all but the longest computations. As a result, RC80M is used as the low-energy clock.

The Power Clocks implementation for SAM4L treats RCSYS as a special case. Normally if a process enters compute mode (it has spent longer than a time quanta on computation), Power Clocks will defer switching to a low-energy clock if there are outstanding peripheral operations. However, if a process enters compute mode when RCSYS is in use, Power Clocks will switch to a faster clock even if other processes have active peripheral operations (as long as none of those peripherals are no-jitter). RCSYS is so energy-inefficient that the associated compute cycles just for driver software become a significant cost. The only use case we’ve found RCSYS to be energy-efficient for is waiting on infrequent GPIO interrupts, where computing is a tiny fraction of application time.

4.3 Example Chip: K66

The second microcontroller we examine is the NXP K66. The K66 has 256kB of RAM, 1MB of flash, and its core can run at speeds up to 180MHz, its bus at up to 60MHz, and its flash at up to 28MHz. The K66 has seven clock sources which can be used to drive the CPU and peripherals. The primary clock in use is determined by the on-chip multi clock generator (MCG), which can be configured into one of eight different states, each enabling a different set of output clocks. Changing from one clock to another on the K66 requires transitioning along the MCG’s state diagram. Some states serve as transition states and have multiple clocks enabled, incurring correspondingly higher power draws. Power Clocks minimizes time spent in those states, only using them to transition between states with a single clock in use. In cases where transitioning from one clock to another requires passing through an intermediate clock, Power Clocks ensures that active peripherals are compatible with every intermediate clock that is used.

5 Evaluation

This section evaluates Power Clocks. It measures the energy savings provided by Power Clocks on the SAM4L over static clock choices by dynamically changing between low-power and low-energy clocks, and compares how Power Clocks does against optimal hand tuned clock changes. It

also measures Power Clocks’s energy savings when multiple independent applications run concurrently, something which is impossible with application controlled power management. We quantify Power Clocks’s cost in code size and clock cycles. Finally, we showcase Power Clocks’s generality by examining a port to a different MCU, the NXP K66, and by discussing Power Clocks’s applicability to other popular chips, STMicro’s STM32F303VCT6 and Nordic’s nRF52840.

5.1 Methodology

All experiments labeled “Power Clocks” use the implementation described in Section 4. Static clock experiments represent clock choices that optimize for either device or compute operations. For experiments using the static low-energy (fast) clock, we use the DFLL at 48MHz, because it uses slightly less energy per tick than RC80M when no clock changes are required, making for a fairer comparison. Hand tuned experiments use a version of Tock without Power Clocks that simply exposes the clock change mechanism. Kernel hand tuned experiments represent the optimal dynamic clock policy; they rewrite parts of the Tock kernel to implement the application-optimal policy. Application hand tuned experiments place optimal clock changes in the application, representing what might be done by a energy-conscious application developer with hardware-specific knowledge.

We measured power using a RIGOL DS4024 oscilloscope and voltage with a TI INA210-215 EVM current sense amplifier across a 1ohm resistor in series with the SAM4L’s VCC. We use two representative applications to measure clock management’s effects on energy efficiency.

5.1.1 Audio Classifier

The first application samples and processes audio to classify ambient noise events, for example to count vehicles on a university campus [1]. The application periodically samples 125ms of audio at 31.5kHz from a microphone connected to the ADC. It applies a Fourier transform to derive 7 frequency bands, computes summary statistics for each band, and sends them to a server using an IEEE802.15.14 radio. Between samples the system is in a deep sleep state.

The ideal policy for this application transitions through 7 different clock stages. In deep sleep, it uses the lowest-power clock, 32kHz RC32k. When it wakes up and starts sampling, it uses the lowest energy clock, 80MHz RC80M. ADC sampling requires a clock 32 times as fast as the sample rate, 4MHz RCF4M. Computing the Fourier transform uses RC80M. It sends the data to the radio over a 2MHz SPI bus using RCF4M. It waits for the GPIO interrupt from the radio with 1MHz RC1M. On the packet complete interrupt, it goes back to RCF4M to send SPI commands to turn off the radio, then returns to deep sleep in RC32k.

5.1.2 Groundwater Pollutants

The second application detects groundwater accumulation and analyzes it for pollutants. [8] The application spends most of its time in sleep, until rainfall generates enough groundwater flow to close a circuit in an external sensor and trigger a GPIO interrupt. It then measures pollutant levels with I2C connected sensors. It compresses sensor readings,

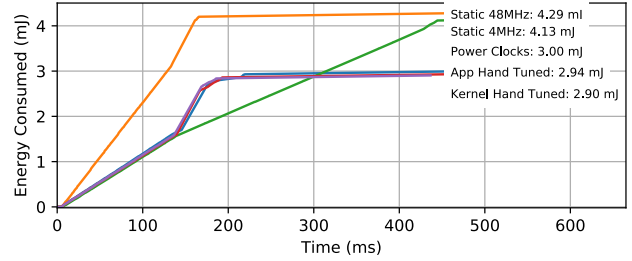


Figure 3: Cumulative energy of the *Audio Classifier* application using five clock policies. Power clocks is within 3% of the hand tuned in-kernel policy and consumes 27% less energy than the best static clock choice. Power Clocks performs worse than hand-tuned clock choices because the computation in the GPIO interrupt handler is much less efficient with a 115 kHz clock than with a 1MHz or 4MHz clock.

detects anomalies, and finally writes the results to flash. In our experimental version of this application, we emulate the groundwater flow detector with a GPIO interrupt and pollutant sensors with other I2C sensors supported by Tock: the result is nearly identical to the original application.

This application progresses through 4 different clock stages. Waiting for a GPIO interrupt uses a very slow clock (RCSYS). Communicating with the I2C sensor at 400 kHz requires at least a 4 MHz clock, so it uses RCF4M. The flash abstraction writes in place, so uses a read, erase, write cycle. The read uses the RC80M clock at 40 MHz while the erase and write use RC1M at 1MHz.

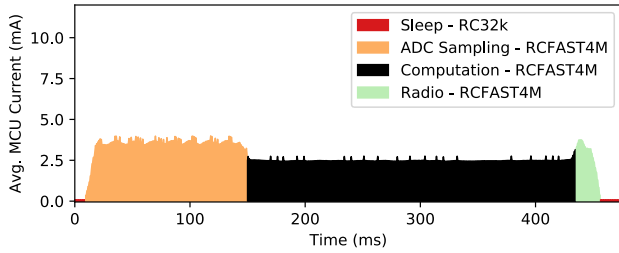
5.2 Application Energy Consumption

We measured the energy consumption of each application under Power Clocks and four other policies: statically choosing the lowest-energy clock (DFLL 48MHz) and the lowest-power clock (RCFAST at 4MHz), using an application hand tuned policy, and using a kernel hand tuned policy.

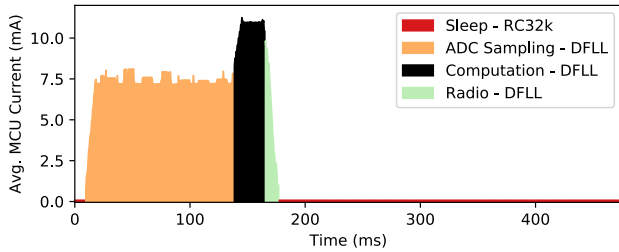
5.2.1 Audio Classifier

Figure 3 shows the cumulative energy consumed by each policy for the Audio Classifier application. Power Clocks consumes 30% less energy than the static low-energy policy and 27% less than the static low-power policy. Power Clocks consumes 2-3% more energy than hand-tuned policies. Figure 4 shows power-over-time profiles of four policies: lowest-power, lowest-energy, kernel hand tuned and Power Clocks. Figure 4a shows that even when the static RCF4M clock is used, power consumption varies depending on what is active. Most of the application’s energy is spent in computation because a low-power clock has a higher per-tick cost. Figure 4b shows the power profile for the application under a static DFLL clock. Sampling takes the same amount of time as under RCF4M but draws much more power. The computation, however, is 12x faster and uses 66% less energy. The low-power clock is more efficient for sampling, but the low-energy clock is more efficient for computation: a dynamic policy that transitions between them will be more efficient.

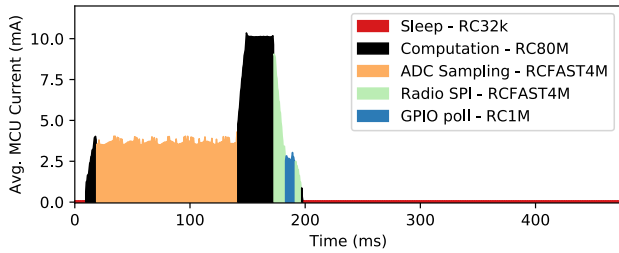
Figure 4c shows current draw under the ideal hand tuned



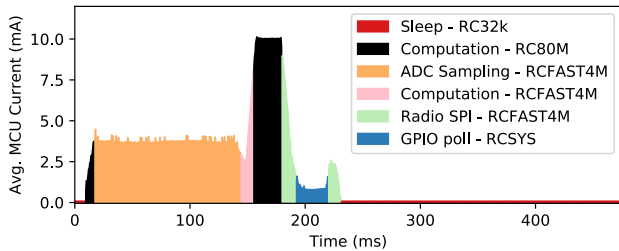
(a) Static RCFast (4 MHz) - Low Power Clock - Energy: 4.13 mJ



(b) Static DFLL (48 MHz) - Low Energy Clock - Energy: 4.29 mJ

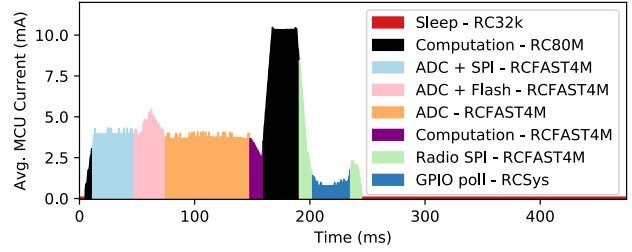


(c) Kernel Hand Tuned - Energy: 2.90 mJ



(d) Power Clocks- Energy: 3.00 mJ

Figure 4: Power profile of the Audio Classifier application with different clock policies. Power Clocks picks the same clock as the ideal hand-tuned policy, except that it waits for a timeslice expiration to switch to a fast clock when the ADC completes, and waits on the GPIO interrupt with RCSYS.



(a) Power Clocks- Energy: 3.51 mJ

Figure 5: Power profile with multiple applications when the active periods of each app overlap. Power Clocks chooses the correct clock even when devices with different preferred clocks execute simultaneously, as evidenced by use of the 4MHz clock when app 1 ADC sampling occurs simultaneously with app 2 flash writes, although flash would use the 1MHz clock when executing alone. With multiple applications, an OS abstraction for clock management is the only viable strategy for saving energy via dynamic clock selection.

Table 3: Energy consumption of the Groundwater Pollutants app with 4 different clock management schemes over different operational periods. “Active only” refers to the time before the application returns to waiting on an interrupt.

Scheme	Measurement Time		
	Active Only	300 ms	1 day
Static 4MHz	0.47 mJ	2.33 mJ	407 J
Static 48MHz	0.55 mJ	5.69 mJ	1729 J
Hand Tuned	0.40 mJ	1.06 mJ	150 J
Power Clocks	0.41 mJ	1.07 mJ	150 J

policy, while Figure 4d shows Power Clocks, which has nearly identical behavior to this ideal. There are two notable differences. First, Power Clocks computes for a 10ms time quanta using RCFast4M before transitioning to an 80MHz clock, while the hand-tuned implementation knows it is executing a long computation so switches to 80MHz immediately. This brief period of using a sub-optimal clock, plus the CPU cycles of clock manager logic, leads to about 0.5% energy overhead for Power Clocks. Second, Power Clocks waits for a GPIO interrupt from the radio using RCSYS instead of RC1M, adding another 2.5% overhead. The hand-tuned policy can exploit the knowledge that the GPIO interrupt from the radio will return very quickly, such that the energy cost of computing the interrupt handler with such an inefficient per-tick clock (RCSYS) outweighs the savings from RCSYS being lower-power while waiting. The application hand-tuned policy is less efficient than the kernel one due to the overhead of making system calls to change the clock and longer periods of executing the CPU at a lower-power clock due to these boundary crossings.

5.2.2 Groundwater Contamination

Table 3 shows the energy consumption of the Groundwater Contamination application under four policies as the

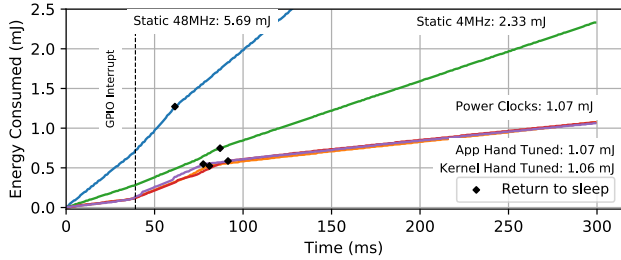


Figure 6: Cumulative energy of the *Groundwater Pollutants* application using five clock policies. All 3 dynamic policies perform within 1% of one another, while consuming 54-55% less energy than the best static clock choice over just a 300ms period. The savings come from no static clock choice being able to satisfy all device requirements *and* achieve lowest-power operation while waiting for a GPIO interrupt.

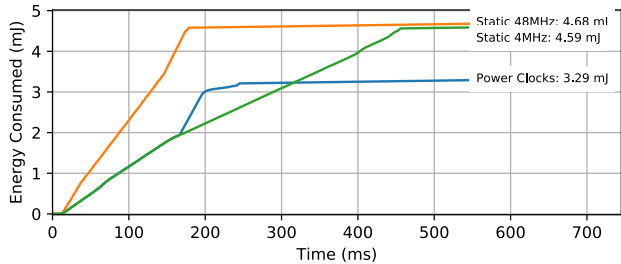


Figure 7: Cumulative energy of the multiple application example when application execution overlaps. Power Clocks uses 28% less energy than the best static clock and 6% less than running the applications sequentially.

frequency of event detection changes. Over 300ms, the three dynamic policies are all within 1% of each other, and consume 54-55% less energy than the best static approach. These savings come from the long periods when the application waits for a GPIO interrupt, during which the clock dominates power draw. Using a static clock, the clock must be fast enough for I2C operations (4MHz), while with dynamic clocks the system can drop down to RCSYS. Figure 6 shows the cumulative energy consumed by each policy for the Groundwater Contamination application when the sample interval is 300ms: the long periods of GPIO waiting (the long straight lines) dominate energy consumption. Power Clocks achieves equivalent or slightly better energy efficiency than hand-tuned approaches, without requiring application-level energy management logic or an application-specific kernel.

5.3 Multiple Applications

To evaluate Power Clocks by its ability to efficiently coordinate multiple simultaneously running apps, we flashed the two apps described in Section 5.1 onto the same board. We slightly modified the groundwater pollution app to collect pollution measurements on a timer instead of in response to a GPIO interrupt (otherwise waiting on GPIO interrupt dominates consumption). We measure two scenarios - when their active periods overlap and when they do not.

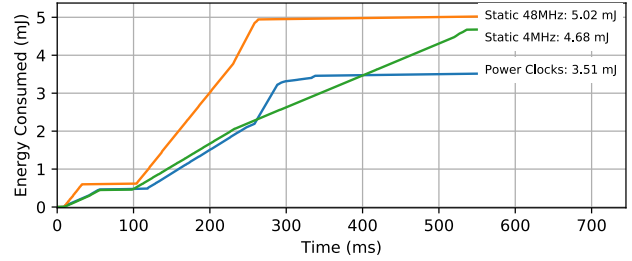


Figure 8: Cumulative energy of the multiple application example when the applications execute sequentially. Power Clocks consumes 25% less energy than the best static clock.

Table 4: Code size (bytes) comparison

	ROM	diff
Tock	132,072	-
+ClockManager	134,344	2272
+ADC	135,184	840
+Flash	135,416	232
+GPIO	135,680	264
+I2C	135,848	168
+SPI	136,168	320
+USART	136,420	252
Total		4,348

Figure 7 shows the cumulative energy used when the active section of both apps overlap. Over a 550ms period, Power Clocks uses 28% less energy than the best possible static clock choice. Figure 8 shows the cumulative energy used when the apps trigger sequentially rather than simultaneously (perhaps as the result of non-overlapping timer expirations). In this scenario, Power Clocks uses 25% less energy than the best possible static clock choice. This difference shows how Power Clocks can exploit overlapping device operations to reduce total energy consumption. This agrees with prior results that parallelizing I/O operations reduces energy consumption [17]. When Power Clocks is used, overlapping app execution and then sleeping uses 6% less energy than executing each app sequentially. Figure 5 shows the power profile when both apps execute simultaneously.

5.4 Power Clocks Overhead

This section quantifies Power Clocks's overheads in terms of code size and CPU cycles: code size is a limiting resource

Table 5: CPU cycle overhead of Power Clocks. Overhead scales with the number of clients because some calls iterate across all active clients. For our example applications, the most expensive operation is 1232 cycles, or 30 μ s at 40MHz.

Function	no jitter	jitter
make_clock_request	206-352	195-341
release_clock_request	124	194 + 30-36/client
clock change	812 + 210/client	812 + 210/client

in embedded systems and CPU cycles represent energy overhead. Table 4 shows the Tock ROM use as ClockManager and ClockClient support for drivers are added. ClockManager adds 2.3kB of code. Each ClockClient adds a further 168-840 bytes. These numbers are approximate because of inlining and link-time optimization. For example, the ADC adds the most (840 bytes) because of dead code elimination; when there are no ClockClients, the compiler elides some of ClockManager’s logic. Supporting all major SAM4L devices adds a total of 4.4kB of code (3.3%); doing so allows the kernel to automatically reduce energy consumption by over 30%. There is no increase in RAM use.

Table 5 shows the CPU cycle costs of Power Clocks operations. This includes the logic in ClockManager but not the configuration register writes to change the clock (since every dynamic policy has this cost). Some operations require iterating across pending or current requests (e.g., to calculate the best clock) so their overhead increases with the number of active clients. In the multiple application example (Section 5.3), the largest number of active clients is 2 (ADC + SPI). In such a workload most expensive operation in Power Clocks is `clock_change`, at 1232 cycles. On the 48MHz SAM4L microcontroller on `imix`, this is 25 μ s; compared to the hundreds of milliseconds that the applications execute, this is a tiny overhead.

5.5 Generality

So far, this evaluation has focused on our implementation of Power Clocks for the Atmel SAM4L. However, we believe Power Clocks is portable for use on other modern microcontrollers with different clock requirements. Here, we present an analysis of our implementation for the K66. The K66 has a complex clock system that is very different from the SAM4L, including a state machine of allowable clock changes that increases the overhead of dynamically changing clocks.

We evaluated our K66 Power Clocks implementation using a variation of the audio classifier application. The primary difference is that sending a radio packet is replaced with writing data to flash, as the Teensy 3.6 does not have an on-board radio. The FLL set at 96MHz is the low-energy clock, and a 4MHz internal clock serves as the low-power clock. However, the ADC can only divide the clock frequency by up to 16 times, placing an upper bound on the clock frequencies it can accept. As such, a 16MHz clock is the effective static low-energy clock for comparison. On this platform, Power Clocks achieves 35% energy savings when compared with the best static clock choice when running this example application, as shown in Figure 9.

Though we have not yet implemented Power Clocks on any additional platforms, we can consider two other representative chips from popular families of different vendors. The STMicro STM32F303VCT6 (the core for the STM32F3Discovery platform) [29] is representative of the STMicro STM32 family and is a popular selection for ultra-low cost systems. The Nordic nRF52840 [23] is representative of the Nordic nRF5x family, a popular selection for end-user applications such as iBeacon tags due to its clear documentation and higher levels of hardware abstraction.

The STM32F3 chip has a similar clock model to the

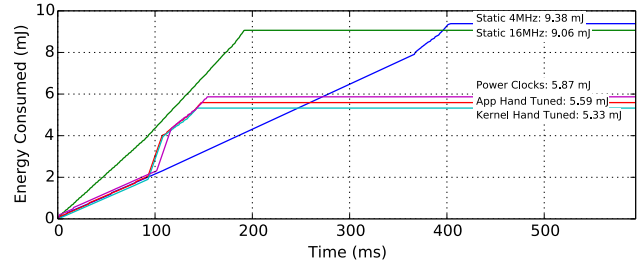


Figure 9: Cumulative energy of the *Audio Classifier* application with flash instead of radio for the K66. Power Clocks consumes 35% less than the best static clock. Power Clocks uses more energy than hand-tuned since it waits for a 10ms time quanta before switching clocks during compute, and has an extra transition to/from the low-energy clock at the end.

SAM4L: it has RUN, SLEEP, and STANDBY power modes, and three clock sources that can drive the system clock. Clock control is left to software. Given this similarity, we believe that Power Clocks could offer similar gains on this platform as was demonstrated on the SAM4L and K66.

The nRF52840 generates two primary clocks: a high frequency (64MHz) and a low frequency (32kHz) clock. The high frequency clock drives the CPU, and is also divided to generate 1/16/32MHz peripheral clocks. Power Clocks cannot be applied as the nRF52840 uses a hardware clock controller that automatically distributes clocks to peripherals based on their clock requirements. Software only needs to ensure the clock source is enabled before a peripheral requires the clock, which can be done once during initialization. However, the nRF52840’s clock management system is limited in its energy savings. Generating slower peripheral clocks from a 64MHz clock source is equivalent to always using the fastest clock and dividing it down with peripheral specific clock divisors. As long as there are active peripherals, the 64MHz clock must always be generated, even if those peripherals only require a 1MHz clock.

6 Related Work

Reducing energy consumption in computer systems has been an active area of research since the late 1990s. Power Clocks complements and builds on this body of work to be the first system to automatically minimize the energy consumption of active devices in embedded systems. Prior work falls into four major categories: minimizing duty-cycle, trading off performance for efficiency, energy management by the OS, and clock configuration tools.

6.1 Minimizing Duty Cycle

Active power draw, such as time spent executing instructions or transmitting on a radio, is usually 1-2 orders of magnitude higher than other system states. The typical first step to reduce energy consumption is duty-cycling: minimizing time spent in the active state. Embedded operating systems such as TinyOS [19] and Tock [20] place the microcontroller in a sleep mode whenever the processor is idle, while ConTiki [5] leaves controlling the duty cycle to the application.

Integrated concurrency and energy management (ICEM)

uses asynchronous I/O to minimize device active time, so the kernel can group and schedule requests efficiently. [18] Power Clocks's asynchronous clock request API borrows from ICEM's use of asynchronous locks to protect shared resources. ICEM, however, was designed for older embedded systems with a single clock source: it saves energy by minimizing completion time. Power Clocks generalizes this to manage multiple clocks and their interactions with devices.

Dynamic Power Management (DPM) is a refinement of duty-cycle based techniques to support a variety of low-power modes on modern microcontrollers. DPM allows an OS to automatically transition between different low power modes based on the requirements of device drivers and applications. [2] TI-RTOS [14] is an example of a commercial RTOS that provides robust DPM support. TI-RTOS drivers declare dependencies on hardware resources and allowable sleep modes, and a centralized power manager automatically configures clock gates and power domains. TI-RTOS also handles transitions into low power modes, respecting dependencies of certain drivers on higher power sleep modes.

DPM-like approaches represent the extent of *automatic* low power approaches on embedded OSes today. Zephyr [24] allows for automatic transitions between low power states, but any more complex power control mechanisms are left to the application. Contiki and FreeRTOS leave clock management to applications.

Generally, DPM differs from Power Clocks in several significant ways. DPM does not reduce active power by configuring the core clock speed - a single static clock is used for all wake periods. Further, kernel-based DPM power managers generally run in the idle loop, and thus cannot make any optimizations based on peripheral operations completing if those peripheral operations trigger any other work.

6.2 Trading Off Performance for Efficiency

In traditional systems, CPU active power can dominate a device's energy budget. Dynamic voltage and frequency scaling (DVFS) allows a device to trade off performance for efficiency. [31]. The fundamental problem in DVFS is predicting future CPU utilization and correctly handling periodic (e.g., multimedia) workloads [9]. Vertigo [6] and GRACE-OS [32] demonstrated that if applications provide information about their future intentions, the operating system can be more energy efficient without degrading application performance. These techniques have been widely adopted in sensor networks, laptops [7], smart phones [10], and cloud workloads [15].

What Power Clocks does is superficially similar to DVFS, since both result in the system dynamically changing frequency in order to save energy. However, Power Clocks changes the frequency because low power clocks are more energy-efficient for I/O bound operations and low energy clocks are more energy-efficient for compute operations, whereas DVFS changes the frequency because doing so allows to save energy by lowering the voltage.

[2] describes a wealth of energy-aware scheduling algorithms for real time systems, based on DVFS algorithms. These algorithms focus on using the minimum clock frequency (and thus voltage) without ever missing application specified deadlines. In addition to being inapplicable on

MCUs that do not support voltage scaling, these techniques miss many of the benefits of Power Clocks. Most real-time DVFS algorithms require a periodic task model, a requirement which most WSN OSes (Zephyr, Contiki, TinyOS, Tock) avoid because of the burden it places on application developers. Further, these algorithms largely require that all tasks being scheduled can operate at any of the available processor speeds, and that the tasks are robust to clock-speed changes even while a task is running. On the systems that Power Clocks targets, these assumptions only hold if separate dedicated clocks are used for peripherals, which eliminates the power savings from sharing a single clock.

6.3 Energy Management

Another body of related work explores how an operating system can manage energy as a resource. The seminal paper in this body of work proposes "Currentcy" as a new resource to consider when scheduling, akin to disk quotas. [33] Implemented in ECOSystem, Currentcy is much simpler than prior approaches like Nemesis OS that are grounded in economic models. [22] The Cinder operating system extends these ideas further, simultaneously allowing an OS to manage both energy and power. [26]

Applications supporting a range of output quality levels ("approximate computing") interact with an OS to choose a trade-off between quality and lifetime. In Odyssey, a power manager "viceroy" provides feedback to adjust the output quality of a variety of applications to meet a system lifetime goal [7]. More recently, JouleGuard introduces a crisp control theoretic framework with provable guarantees. [13]

Pixie OS applies these techniques to embedded systems, proposing resource aware programming. Applications in Pixie OS are structured as dataflow graphs, with individual processing elements reacting to available resources [21] Power Clocks complements this work; while they seek to manage how quickly energy is consumed, Power Clocks seeks to reduce the cost of individual operations, thereby allowing a given energy budget to perform more work.

6.4 Clock Configuration Tools

Many modern microcontrollers have complex clock trees, where sets of devices can run off different clock sources, with complex constraints as to which clocks and frequencies can be used. Given the complexity of correctly configuring a system, there exist tools such as STM32CubeMX [30] for STM32 microcontrollers, which adjusts bus frequencies, timers, and devices to work with the user's selection of clock sources, frequencies, and divider values.

Power Clocks is similar to STM32CubeMX in that it adjusts bus frequencies, timers, and devices to work with different clock sources. Unlike the STM32CubeMX, which does this configuration once in expectation of a static clock source, Power Clocks does dynamic configuration as the clock source changes. Further, STM32CubeMX allows for configuration of multiple clock sources to reduce response times, while Power Clocks only uses one system clock at a time in order to minimize energy usage.

7 Conclusion + Future Work

Power Clocks is designed to reduce a microcontroller's energy usage by allowing it to dynamically change its clock

to reflect changes in application state. It selects an energy-optimizing clock for any combination of application states at the cost of requiring that all operations are serviced asynchronously. Power Clocks is implemented entirely in the kernel and requires no changes to application code. Power Clocks achieves significant ($> 25\%$) energy savings in both single and multiple app scenarios.

Prior work has shown that deadlines can be used by RTOS systems to improve energy efficiency. Power Clocks relies on heuristics for estimating task duration, but creating a deadline aware Power Clocks algorithm is an interesting area of future work that could provide further efficiency improvements. Power Clocks shows that kernel-managed, dynamic clock-source selection is a promising method for achieving energy savings in low power devices.

8 References

- [1] J. Adkins, B. Ghena, N. Jackson, P. Pannuto, S. Rohrer, B. Campbell, and P. Dutta. The signpost platform for city-scale sensing. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN 2018, Porto, Portugal, April 11-13, 2018*, pages 188–199, 2018.
- [2] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):1–34, 2016.
- [3] S. Dawson-Haggerty, A. Krioukov, and D. E. Culler. Power optimization—a reality check. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-140*, 2009.
- [4] D. W. Dobberpuhl, R. T. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R. A. Conrad, D. E. Dever, B. Gieseke, S. M. N. Hassoun, G. W. Hoepfner, K. Kuchler, M. Ladd, B. M. Leary, L. Madden, E. J. McLellan, D. R. Meyer, J. Montanaro, D. A. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam. A 200-mhz 64-b dual-issue cmos microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, 1992.
- [5] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004, IEEE EmNetS-I*, Nov. 2004.
- [6] K. Flautner and T. Mudge. Vertigo: automatic performance-setting for linux. In *Proceedings of the 5th symposium on Operating systems design and implementation, OSDI 02*, pages 105–116, New York, NY, USA, 2002. ACM Press.
- [7] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP '99*, pages 48–63, New York, NY, USA, 1999. ACM.
- [8] J. Frigo, H. Ayers, V. Kulathumani, S. Hinzey, S. Sevanto, M. Priocou, X. Yang, K. McCabe, A. Saari, and K. Sentz. Novel wsn hardware for long range low power monitoring. In *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 89–92, June 2017.
- [9] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking, MobiCom '95*, pages 13–25, New York, NY, USA, 1995. ACM.
- [10] D. Grunwald, C. B. Morrey, III, P. Levis, M. Neufeld, and K. I. Farkas. Policies for dynamic clock scheduling. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4, OSDI'00*, Berkeley, CA, USA, 2000. USENIX Association.
- [11] A. Guldahl. <https://www.embedded.com/understanding-mcu-sleep-modes-and-energy-savings/>.
- [12] J. L. Hill, R. Szweczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000.*, pages 93–104, 2000.
- [13] H. Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*, pages 198–214, New York, NY, USA, 2015. ACM.
- [14] T. Instruments. Ti-rtos: Real-time operating system (rtos) for micro-controllers (mcu), 2017.
- [15] C. M. Kanga. Cpu frequency emulation based on dvfs. *SIGOPS Oper. Syst. Rev.*, 47(3):34–41, Nov. 2013.
- [16] Kinetis K66 Series Microcontrollers. <https://www.nxp.com/docs/en/data-sheet/K66P144M180SF5V2.pdf>.
- [17] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. *SIGOPS Oper. Syst. Rev.*, 41(6):251–264, Oct. 2007.
- [18] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, pages 251–264, New York, NY, USA, 2007. ACM.
- [19] P. Levis, S. Madden, J. Polastre, R. Szweczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.
- [20] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, pages 234–251, New York, NY, USA, 2017. ACM.
- [21] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource aware programming in the pixie os. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems, SenSys '08*, pages 211–224, New York, NY, USA, 2008. ACM.
- [22] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis OS. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20-23, 2001, Elmau/Oberbayern, Germany*, pages 67–72, 2001.
- [23] NRF52840 System on Chip. <https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840>.
- [24] Z. Project, 2019.
- [25] T. Rault, A. Bouabdallah, and Y. Challal. Energy efficiency in wireless sensor networks: A top-down survey. *Computer Networks*, 67:104–122, 2014.
- [26] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems, EuroSys '11*, page 139152, New York, NY, USA, 2011. Association for Computing Machinery.
- [27] A. SAM4L. <https://www.microchip.com/wwwproducts/en/ATSAM4LC4C>.
- [28] F. K. Shaikh and S. Zeadally. Energy harvesting in wireless sensor networks: A comprehensive review. *Renewable and Sustainable Energy Reviews*, 55:1041–1054, 2016.
- [29] STM32F3 ARM Cortex-M4 Microcontrollers. <https://www.st.com/en/microcontrollers-microprocessors/stm32f3-series.html>.
- [30] STMicroelectronics. Um1718 user manual. https://www.st.com/resource/en/user_manual/dm00104712.pdf, 2019.
- [31] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Mobile Computing*, pages 449–471. Springer, 1994.
- [32] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, pages 149–163, New York, NY, USA, 2003. ACM Press.
- [33] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 43–56, 2003.