Dynamic Clock Reconfiguration for the Constrained IoT and its Application to Energy-efficient Networking

Michel Rottleuthner
HAW Hamburg
michel.rottleuthner@haw-hamburg.de

Thomas C. Schmidt
HAW Hamburg
t.schmidt@haw-hamburg.de

Matthias Wählisch
Freie Universität Berlin
m.waehlisch@fu-berlin.de

Abstract

Clock configuration takes a key role in tuning constrained general-purpose microcontrollers for performance, timing accuracy, and energy efficiency. Configuring the underlying clock tree, however, involves a large parameter space with complex dependencies and dynamic constraints. We argue for clock configuration as a generic operating system module that bridges the gap between highly configurable but complex embedded hardware and easy application development. In this paper, we propose a method and a runtime subsystem for dynamic clock reconfiguration on constrained Internet of Things (IoT) devices named ScaleClock. ScaleClock derives measures to dynamically optimize clock configurations by abstracting the hardware-specific clock trees. The ScaleClock system service grants portable access to the optimization potential of dynamic clock scaling for applications. We implement the approach on the popular IoT operating system RIOT for two target platforms of different manufacturers and evaluate its performance in static and dynamic scenarios on real devices. We demonstrate the potential of ScaleClock by designing a platform-independent dynamic voltage and frequency scaling (DVFS) mechanism that enables RIOT to autonomously adapt the hardware performance to requirements of the software currently executed. In a use case study, we manage to boost energy efficiency of constrained network communication by reducing the MCU consumption by 40 % at negligible performance impact.

Categories and Subject Descriptors

D.4.9 [**Operating Systems**]: Systems Programs and Utilities; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Management

Keywords. Embedded Systems, Energy, DVFS

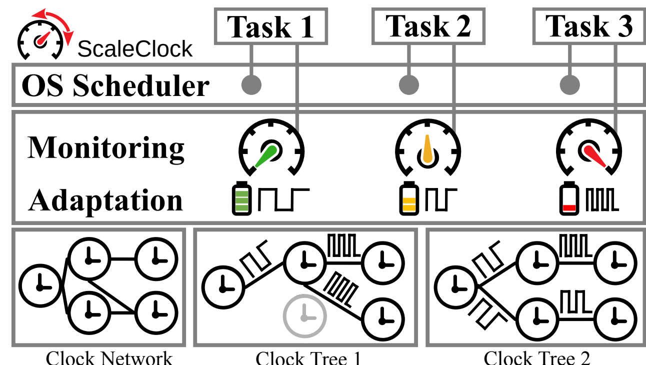


Figure 1: ScaleClock derives tasks characteristics at runtime and adapts the system performance to application needs. Its clock tree abstraction it is able to dynamically reconfigure the hardware for more efficient operation and saving energy.

1 Introduction

Embedded systems based on microcontroller units (MCUs) and diverse peripherals are omnipresent today, and the rapidly evolving IoT deployment turns them into networked devices. Hardware abstractions available from various (open source) IoT operating systems [1–3] facilitate to assemble and maintain portable software on heterogeneous embedded platforms. Nevertheless, key performance characteristics (*e.g.,* accuracy, energy, connectivity, lifetime) have conflicting impacts, which calls for advanced optimization and control of the interplay between hardware and software.

Energy availability is severely constrained on wireless IoT devices and mandates for efficiency improvements on each layer [4, 5]. The clock configuration of a system comprises a control knob for balancing between high performance and low power as every switched signal inevitably affects power consumption [6]. Adaptive software can exploit this to save energy (*e.g.,* via dynamic voltage and frequency scaling (DVFS)), making the clock configuration fundamentally important whenever energy efficiency is a concern [7].

Frequency scaling requires access to the clock configuration of the underlying hardware. This should be supported by the hardware abstraction layer (HAL) of the operating system. Current operating systems for embedded devices, though, lack abstractions for clock trees and hence are unable to manage the full system clock configuration space.

They cannot dynamically manage clock configurations for optimized performance and energy efficiency [8–10]. This wastes energy in particular on low-power, wireless IoT devices, where peripheral buses are often one order of magnitude slower than the CPU and common radio interfaces have even lower data rates (*e.g.,* IEEE802.15.4 with 250 kbit/s and LoRa with <1 kbit/s).

In this paper, we introduce ScaleClock, a lightweight generic clock configuration solution for constrained IoT devices. ScaleClock enables platform-independent control of complex, MCU-internal clock settings during runtime. ScaleClock configures the *clock network* of a device, the set of clock sources, modifiers, and consumers, such that the effective *clock tree* reflects application needs (see Figure 1).

Dynamic power management with reconfiguration of clock trees on IoT devices is challenging for two reasons. First, constrained devices are restricted by processing power and memory. This prevents transferring approaches from prior work such as Linux. Second, most MCU manufacturers provide a variety of internal and external clock sources. These sources exhibit diverse properties supporting orthogonal objectives. We want to integrate those for the sake of a versatile, platform-independent IoT.

ScaleClock overcomes these challenges by generic methods for assessing tasks and identifying their optimized clock configuration. It utilizes a layered architecture consisting of composable base types for clocks and unified configuration interfaces that decouple from the underlying hardware. Applications do not need to interact with our framework because the ScaleClock transition manager autonomously interacts with the OS scheduler and clock configuration. ScaleClock is implemented as part of the open-source operating system RIOT and validated on two common IoT platforms (STM32 and EFM32). Despite its flexibility, the overhead remains small (*e.g.,* 4% more memory than the platform-specific clock tree). It optimizes energy during runtime and explores all possible clock configurations when needed. This operation is fast and results are cached. It provides DVFS as a lean OS-centric service and saves more energy than the race-to-idle strategy often used on constrained devices.

In summary, our main contributions are:

1. A system service for clock-tree exploration and dynamic reconfiguration, which uses a novel method to proactively assess task characteristics for optimizing DVFS control. (§ 3)
2. A flexible abstraction for a light-weight cross-platform modeling of MCU clock-trees. (§ 4)
3. Comprehensive evaluations, including the validation of ScaleClock on two independent target platforms. (§ 5)
4. A case study demonstrating how ScaleClock improves energy efficiency of low-power communication. (§ 6)
5. A ScaleClock open source implementation on RIOT.

In the remainder of this paper, we introduce the problem space and potentials of a reusable model of clock trees with (re-)configurable clocks in § 2. We present our core contributions in § 3–§ 6, discuss related work in § 7, and summarize our findings and present an outlook in § 8.

2 The Clock Tree and its Forest of Problems

Clock trees comprise the low-level clock networks within MCUs. They are responsible for distributing and conditioning the clock ticks required for operating almost every internal component. They substantially differ in complexity and implementation – not only between vendors but also between MCU series and models of the same vendor. In addition, clock trees often come with very complex dependencies that are entangled with hardware configurations and constraints, use of peripherals, environmental conditions, and application demands. Modeling the clock tree configuration as generic reusable component is therefore challenging.

Figure 2 illustrates a simplified example of an MCU clock tree. Left are various clock sources of the tree. Sources provide clock signals to intermediate nodes such as gates, muxes, and scalers to eventually feed a consumer node (*e.g.,* a timer or the CPU). Intuitively it may seem appropriate to just put configuration logic of each part to respective peripheral drivers. Drivers, however, only control local parts (*i.e.,* leaf nodes) and cannot manage the overall coordination of clocks that affect multiple devices. The latter requires a global component on the kernel level.

We surveyed the clock configuration features of eleven popular embedded IoT operating systems. The majority (*i.e.,* Contiki, FreeRTOS, LiteOS, Mbed, Mynewt, NuttX, RIOT, TinyOS) avoid this problem by implementing the clock configuration in static code that is only configurable before compilation. While being efficient w.r.t. code size and execution time, this static pattern does not allow for dynamic reconfiguration at runtime. It effectively neglects optimization potentials, since a single static configuration that operates in the energetic sweet-spot of every task and hardware does not exist. Other OSs (*i.e.,* ChibiOS, Tock, Zephyr) only provide partial dynamic features. Specifically, ChibiOS offers an application programming interface (API) for switching between static platform specific presets by re-initializing the entire clock subsystem, but lacks support for exploration, topology control, and performance assessment. Tock provides an API for clock gating (*i.e.,* enable and disable control), which can be used for automated peripheral power management. Zephyr has an interface to access frequency properties of specific clocks. However, none of them considers an abstract model of the clock tree, topology control, exploration of possible configurations, nor dynamic performance adaptation.

To the best of our knowledge, there is currently no viable solution for handling advanced clock (re-)configuration features even though many operating systems could profit from dynamic optimizations.

2.1 Common Building Blocks for Clock Trees

Clock trees involve many properties that largely vary between devices. Still, most entities (*i.e., clock nodes*) that compose a clock tree share common functionality to serve specific high-level purposes. They either manipulate the topology of clock signals (*i.e.,* the routing to different destinations) or change properties of a clock, such as its frequency, calibration, or duty-cycle. The most primitive form of a clock node is a *gate*. Its sole purpose is to enable or

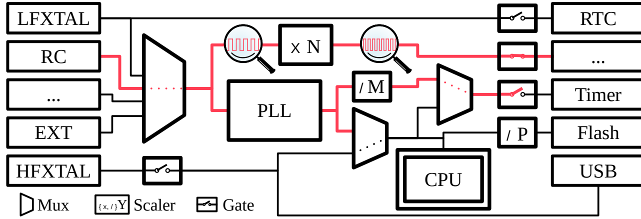


Figure 2: Example clock tree consisting of various clock sources (left), muxes, scalers, gates, and consumers (right).

disable the forwarding of its input clock to its output. A *mux* selectively passes one of its multiple input clocks to its single output. Multipliers (*e.g.,* based on phase-locked loops (PLLs)) and dividers scale the frequency of the clock signal. We therefore collectively refer to them as *scalers*.

The exact features and implementations of those clock nodes differs between platforms but their high level objective and semantics are shared. Respectively, the hardware interfaces admit many commonalities. For example, settings are commonly programmed by writing values to specific memory mapped configuration registers, which also contain bit flags to trigger and observe operations and state changes.

2.2 Configuration Space(s)

Management and encoding of configuration parameters that describe the clock tree and its hardware properties face challenges. Inherent features of hardware or its individual composition with external components must be visible to the configuration tools. We differentiate between *static properties*, *static configuration* and *runtime configuration*.

Static Properties. Static properties are immutable for a specific target platform. Here *target platform* refers to a system model consisting of a particular MCU and permanently connected components such as sensors and crystal oscillators (*e.g.,* a particular smartwatch or evaluation board). Some properties are inherent to the MCU model, *e.g.,* the possibility to connect an external crystal. This optional oscillator may then be present on a specific target platform or not. Overall, these static properties relate to different levels of abstraction and should be defined on the topmost level of commonality. This can be MCUs with a specific instruction set, an MCU family, an MCU model, or an actual user device.

Static Configurations. Static configurations concern parameters that are adjustable per se but their static implementation enforces a fixed value at runtime. Adjusting those parameters therefore requires recompilation. For example, a device that reads data from an external pen drive could be statically configured to enable a 48 MHz clock for driving the MCU-internal Universal Serial Bus (USB) peripheral.

Runtime Configurations. Runtime configurations apply new system parameters during operation. Data describing valid configuration values and adaptation logic must be part of the firmware. For the above USB example, runtime configuration of the peripheral clock becomes highly preferable in battery powered scenarios as it reduces power consumption while the USB-port is unused.

These simple examples of different configuration types illustrate that there is no universal solution that serves all scenarios. Though, a self-adaptive system that shall adjust to its current needs, requires runtime configuration.

2.3 Complex Clock Configuration Transitions

Procedures for changing the configuration of a clock (*i.e.,* its topology or frequency) can vary significantly in complexity. This applies to the execution of the transition itself but also to side effects when other parts of the clock tree are affected by a change. In simple cases the clock may for example be updated on-the-fly by just writing a new prescaler value to the corresponding configuration register. The change is immediately applied and a new frequency is in effect. Contrarily, we refer to a *complex transition* if it consists of multiple phases. There are several reasons why complex transitions are needed. A Transition may affect parts further down in the clock tree, demanding for pre- or post-operations to make the new configuration applicable. Examples are adjustments to voltage range or flash access parameters. There are also changes which take time to complete and temporarily produce unstable frequencies, *e.g.,* on PLLs that require some time to stabilize. Other clocks are simply not adjustable on-the-fly. This is a particular problem when users of that clock are at the same time uninterruptible. Complex transitions solve this by a temporary switch to an alternative source, performing the adjustment, and then switching back.

Determining transitions and target configurations online foremost requires semantic encoding of what each involved step does in order to detect if they can be applied or raise conflicts. Since this kind of online exploration can impose significant overhead, pre-calculating complex clock transitions is considered beneficial for fast repeated execution.

2.4 Hard Resource Limits

It is worth highlighting the significant challenges associated with scaling down the overhead of a generic solution to constrained IoT devices. In those small embedded systems data structures to encode hardware properties and runtime management of configurations can significantly impact the memory footprint. Dynamic loading of hardware description files is out of scope and even dynamic memory allocation is preferably avoided for keeping a fixed memory budget [11].

Overall, its complexity poses a severe challenge on the design of run-time methods to (re-)configure clock trees.

3 ScaleClock Approach for Self-Optimization

This section introduces the core concepts of ScaleClock and describes the mechanisms with which ScaleClock enables the system to assess tasks and optimize their energy level. Built on a generalized clock subsystem control, ScaleClock introduces the ability to dynamically adapt clock frequencies to execution demands and thereby leverage potential energy savings without sacrificing application demands.

Dynamic Power Consumption. Essentially, power consumption of Complementary Metal-Oxide-Semiconductor (CMOS) circuits can be separated into a *dynamic* and a *static* part. We focus on scenarios, in which an MCU performs computations, and the dynamic share dominates the power consumption—as we will also show in § 5. The dynamic power consumption of a switched circuit is described by $P = \alpha \cdot C \cdot V^2 \cdot f$, where C is the capacitance of the switched circuit, V the voltage, and f its frequency. The transistor

switching activity (*i.e.,* number of switched transistors) is reflected by α. C is a property of the specific device, and the executing application mostly defines α. Hence we can only influence the parameters V and f to reduce power consumption of a given application on a device. Both are subject to the following conceptual and practical limits. (*i*) The minimal voltage depends on the frequency; thus, voltage cannot be reduced independently. (*ii*) The available range of voltage adjustment is significantly confined. On modern microcontrollers, core voltage often ranges from 1.8 V down to around 1 V, whereas frequencies can be scaled from hundreds of MHz down to kHz. (*iii*) Time overhead can differ significantly for scaling voltage vs. frequency. Voltage scaling incurs a relatively static time overhead in the order of several µs per 10 mV. Frequency scaling can be almost instantaneous (*e.g.,* when switching a mux or adapting a scale value) to taking multiple ms (*e.g.,* when cold-starting an oscillator). (*iv*) At very low frequencies the static power consumption becomes more relevant, which reduces efficiency. For a more comprehensive background reference, we refer the reader to Castagnetti *et al.* [7] and Eyerman *et al.* [6].

3.1 Resource Demands are Dynamic

The potential for dynamic energy optimization (*e.g.,* from DVFS) is significant because applications rarely utilize the full performance provided by the MCU. A mismatch between the performance configuration of the hardware and the utilization by the software inevitably wastes energy. Full computing performance is not required, for example, at execution phases in which the MCU waits for an operation to complete, such as reading an external sensor, erasing a flash page, or transmitting data. This leaves potential to trade underutilized performance for energy savings.

On resource constrained devices race-to-idle is commonly employed because of its easy implementation, but it often lowers energy efficiency compared to more adaptive methods [12]. With this in mind, the demand for dynamic performance adaptation as a generic system service becomes apparent. Its largest challenge is to define a generic mechanism for dynamic clock configuration on constrained devices.

3.2 Assessment of Resource Utilization

A versatile feedback loop for dynamic runtime optimization needs to acquire precise knowledge about the system conditions via a simple, yet expressive metric. Consider a time slice t, of which the scheduled task utilizes the CPU for the fraction t_{busy}, then

$$Load = \frac{t_{busy}}{t_{busy} + t_{idle}} \quad (1)$$

defines a simple utilization metric often employed on MCUs. We argue that this metric is not well suited for dynamic performance control because it is insensitive to tasks which only *appear* to utilize the CPU but in reality are limited by other operations. Instead, we propose a utilization metric that compares the actual utilization at two different frequencies F_1 and F_2. For any scheduled time slice t, Equation 2 relates the ratio between busy times at different frequencies to the frequency change ratio.

$$PU = \frac{t_{busy}(F_1)}{t_{busy}(F_2)} \cdot \frac{F_1}{F_2}, \quad F_1 < F_2 \quad (2)$$

For perfectly scalable tasks the busy time reduces by the same ratio as the frequency increases. On the converse, tasks with low performance utilization (PU) values scale worse but show high potential for energy savings. Notably, this definition drops the explicit use of the idle time, because the ability of a task to (not) scale well with frequency is independent of its idle time—in contrast to the global system load. Idle time is considered implicitly as the time allocated to the idle task.

3.3 Dynamic Scaling

In ScaleClock, we devise an online PU assessment that executes on the device itself. It instruments the operating system to collect the context switching count together with busy- and idle times for each task. The core frequency is then opportunistically adapted while collecting measurement points for the PU metric. Based on these measurements a task-specific, energy-optimized frequency is selected. For a given set of possible core frequencies corresponding clock tree configurations are determined in an exploration phase, which runs once on system init and on topology changes.

The dynamically assessed target frequency is set up before scheduling the next task. Core voltage and flash wait state adaptation follow the frequency selection according to static constraints which encode clock-node and frequency-specific hardware limits. A policy setting governs whether fast flash or low voltage is preferred in cases where they mutually exclude each other. We evaluate this concept in § 5 for the RIOT [3] operating system on real hardware.

3.4 Interfacing Functional Capabilities

Clock types implement different sets of capabilities, which we reflect by an individual assignment. A clock can for example be *scalable*, *muxable*, *gateable*. Each capability provides its own interface functions such as accessing and configuring a scaling factor for a *scalable* clock, configuring the parent for a *muxable* clock, or simply enabling or disabling a *gateable* clock. Drivers that only differ in details can reuse most capability code by only replacing the selective parts of the interfaces that differs. Capability implementations are explicitly separated from the mapping functions that translate register content to logical values in order to reuse code sections wherever possible.

All properties described so far are provisioned as separate static data structures to support static memory allocation and selective inclusion. These static definitions also open the door for *a priori* encoding tweaks towards our design goals to minimize memory consumption and maximize execution speed. Combining these principles allows an easy to understand modeling while still being expressive, memory efficient, and flexible to adjust for optimizations.

4 ScaleClock Implementation

A suitable level of abstraction is essential for a reusable design that hides hardware specifics but grants sufficient access. In § 2.1, we identified a significant part of common behavior at the level of individual base elements (*i.e., gate*, *mux*, and *scaler*). The hardware facing part of the interface sits at this level so that higher order clock trees can be flexibly orchestrated from separate clock instances.

Albeit uncommon clock nodes exist (*e.g.,* PLLs with multiple stages and outputs), even among devices with very com-

```
const gclk_t *clk = gclk_get(gclk_get_cnt() - 1);
bool enabled = gclk_is_enabled(clk);
unsigned factor = gclk_get_current_factor(clk);
uint32_t freq = gclk_get_current_freq(clk);
const gclk_t *parent = gclk_get_current_parent(clk);
```

Listing 1: Get clock properties.

```
gclk_disable(clk);
gclk_enable(clk);
gclk_set_factor(clk, 42);
gclk_set_freq(clk, 10000000);
gclk_set_parent(clk, other_clk);
```

Listing 2: Set clock properties.

```
uint32_t *freqs;
unsigned cnt = gclk_manager_get_dfs_freqs(&freqs);
// iterate available DFS frequencies
for (unsigned i = 0; i < cnt; i++) {
  gclk_manager_scale_core_freq(freqs[i]); }
```

Listing 3: Set different core frequencies.

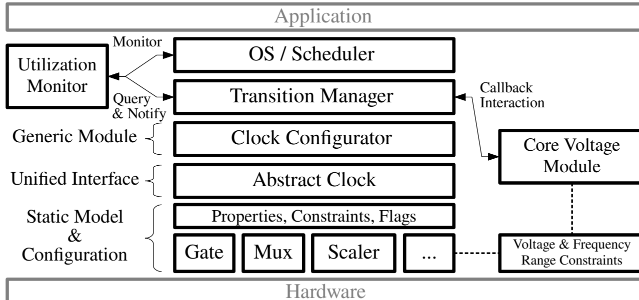


Figure 3: The ScaleClock architecure incorporates a unified configuration interface to individual clocks. The transition manager uses OS statistics to self-adapt the system for low energy or high performance operation.

plex clock trees we never encountered any that could not be modeled by combining the aforementioned base elements. Moreover, custom clock types can still be added, and the design is widely applicable to MCUs with memory mapped configuration registers. We thus argue that these clock types form a reasonably reduced but expressive set to model all typical MCU clock trees. We will demonstrate its utility later in § 5. Key design decisions are presented next, for further details we refer to our publicly available code (see § 8).

4.1 Layered Architecture for Flexibility

Our layered architecture puts common functionality and patterns to a higher utility layer and strictly separates data and code. We observe that clock configuration can be done by adapting *frequency* or changing *topology*. Topology operations may affect the frequency whereas frequency operations shall never affect the topology, as this may have significant impact (*e.g.,* on clock availability and accuracy). We explicitly separate this functionality to retain direct control.

Figure 3 gives an overview of the ScaleClock architecture. An application never has to interact with the clock framework directly, unless it aims for manual control. Operations on the *Abstract Clock* interface are decoupled from the underlying hardware. Platform specific configuration options and hardware access description are provided as static data, separated from the base clock functions, which are defined as independent driver functions. Individual mapping functions convert between semantic values (*i.e.,* multiplier, enable state, clock instance, *etc.*) and configuration register values. Clock base types (*Gate*, *Mux*, and *Scaler*) are provided for reuse by platform specific implementations. Custom clocks can be modeled by separately orchestrating and decorating primitive operations, value descriptors, and mapping functions. Tree variants are constructed from a list of parent choices of each clock. Pointers to unique static descriptors of clock instances serve as identifiers. For easy traversal, getting the current parent of a clock returns a pointer to its static descriptor (see Listing 1). A topology entry datatype stores the logical configuration state of a clock instance for virtual representation of topology settings.

The clock configurator can query and reconfigure hardware state, *e.g.,* to get the current frequency or set a new scaling factor (see Listing 1 and Listing 2). The transition manager handles high-level tasks such as core frequency updates (Listing 3), complex transitions as introduced in § 2.3 (Listing 4), and triggering notifications on configuration changes. The utilization monitor gathers scheduler statistics (idle-, execution-time, context switches) to meter the performance of running threads and provides feedback to identify bottle-necked operations for dynamic optimization. Energy is then saved by aligning the system performance to the task demand. Listing 5 shows a manual invocation of the PU-assessment mechanism in a multi-threaded application.

```
const gclk_t *clk = gclk_manager_get_core_clock_handle();
uint32_t max_len = gclk_get_clk_subtree_max_depth(clk, 0);
clk_topology_entry_t topology[max_len]; topology[0].clk = clk;
for (unsigned ti = 0; ti < gclk_get_topology_config_cnt(clk); ti++) {
  size_t size = gclk_get_nth_topology(topology, max_len, ti);
  gclk_cmp_func_t cmp = closest_leaf_freq; // match to closest frequency
  uint32_t f_match = gclk_manager_switch_topology(clk, ti, f_target, cmp);
  printf("switched to topology %u, running at %u Hz\n", ti, f_match); }
```

Listing 4: Code excerpt to switch between core topologies.

```
for (unsigned i = 0; i < thread_cnt; i++) {//request assessment per thread
  gclk_manager_enable_pu_stat_request_for_thread(thread_ids[i]); }
gclk_manager_enable_pu_assessment(true);//enable assessment
gclk_manager_start_freq_cycler(MIN_DURATION, MIN_SCHEDULES);//sweep freq.
for (unsigned t = 0; t < thread_cnt; t++) {
  int pu = gclk_manager_calculate_pu_factor(thread_ids[t]);
  printf("PU of thread %u: %d\n", thread_ids[t], pu); }
gclk_manager_enable_dynamic_frequency_scaling(true);//enable PU-based DFS
```

Listing 5: Code excerpt to assess performance utilization.

Porting ScaleClock to new platforms only requires provisioning of hardware-specific static data (register mappings, constraints etc.), *i.e.,* layers below the abstract clock interface in Figure 3. Generic low-level drivers can be reused directly in most cases and manual effort to extend capability drivers of specific clocks (as described in § 3.4) is rarely needed. Higher layers of ScaleClock require no porting.

4.2 Property Encoding

Clock nodes can have many (configurable or immutable) properties, such as valid input sources or divider values. Those must be accessible to upper layers, whereas respective register values and their addresses must be available to the hardware facing layer. Reconfiguration constraints (*e.g.,* on-the-fly adaptability of scaling factors) must be stored too.

Numerical values are encoded using an extendable set of mapping types to ease modeling of sets of distinct values or ranges with associated semantics. Memory overhead is further reduced by using indicators for common implicit value encoding patterns and modifiers. In many cases this halves the storage for mappings between logical and register values.

Instead of saving configuration registers and masks separately for every clock node we leverage that only a subset

of registers are typically responsible for clock configuration and combine them into a lookup table. A register is then identified with only a few bits and a single integer can be shared for storing the register id, masking information and bit indexes for common flags (*e.g.,* enable or ready states). Respectively, a frequency multiplier descriptor that accepts factors between 1 and 8 by writing values from 0 to 7 into a specific register field can be encoded by the bounds (1,8) and a *zero-based* implicit register value modifier. Using the condensed configuration register descriptor, only two 32-bit integer values are needed to encode how to access the hardware register and which values it may attain.

4.3 Notification & Transaction Mechanism

Interaction between ScaleClock and other modules requires methods to request, indicate or block changes. While ScaleClock itself can handle dependencies and constraints internal to the clock tree, this does not cover dependencies on internal state of peripheral drivers or application logic. A clock, for instance, can be safely turned off if no other component is using it. If it is used, *e.g.,* by a peripheral driver, it can block or allow transitions depending on its operation.

External modules can prepare for clock transitions or trigger reconfiguration procedures afterwards via *pre*- and *post*-hooks, which can be registered for any existing clock instance. Shared peripherals can lock access for the duration of the transition via the *pre*-call. During the *post* hook, the respective module must be put back to normal operation mode.

5 Evaluation of ScaleClock

We are now ready to evaluate key performance metrics of ScaleClock regarding *functionality*, *performance*, and *overhead*. First, we describe the core features enabled by ScaleClock and compare them to alternative mechanisms. Second, we benchmark the effect of configuration parameters in static scenarios (*i.e.,* running different tasks at preset frequencies). Third, we evaluate the performance of dynamically applying ScaleClock (*i.e.,* changing frequency at varying application needs). We measure the energy savings and the temporal overhead. Later in § 6 we will assess the energy savings in a realistic case study of low-power wireless networking.

Experiments are conducted on the Nucleo-L476RG by STMicroelectronics and the SLSTK3402A EFM32 Pearl Gecko PG12 board by Silicon Labs. Both boards are unmodified and our experimentation firmware sources are publicly available (see § 8) to ease reproducibility. All stated current values relate to a static supply of 3.3 V. We use a highly accurate Keithley DMM7510 digital sampling multimeter [13], connected to the power headers (IDD and BAT respectively).

5.1 ScaleClock Core Functions

ScaleClock supports active exploration of the clock subsystem and increases visibility and usability of hardware capabilities. Developers can use the same unified API on all target devices to explore available clock configurations and how they can be operated—instead of studying data sheets for each target platform. The system itself exploits this knowledge to dynamically optimize the hardware configuration in concordance with the changing application needs.

Reducing Power Consumption. Applications facing (temporary) constraints on peak power consumption can instruct

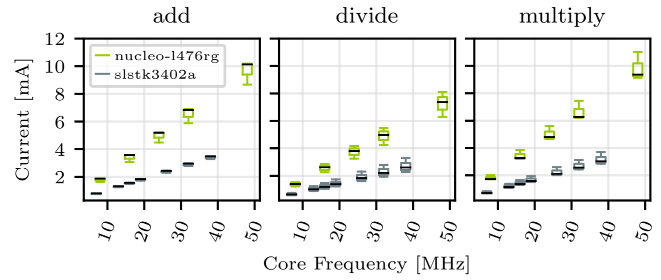


Figure 4: MCU current draw behavior for tasks that execute loops of different integer instructions (add, divide, multiply) at varied clock frequencies on two devices. IQR: 25th-75th percentile, whiskers: Q1-1.5*IQR and Q3+1.5*IQR.

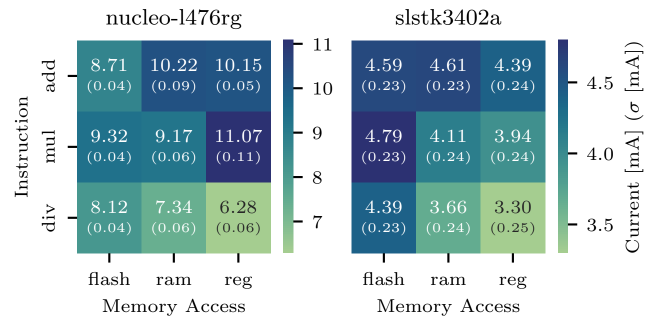


Figure 5: MCU current draw behavior of two devices when executing tasks which loop different integer instructions (divide, add, multiply), using data from distinct memory locations (flash, RAM, registers); left: 48 MHz, right: 40 MHz.

ScaleClock to throttle clock speed. For example, systems facing critical supply voltage may thereby limit power consumption to reliably maintain operation of the voltage regulator. Energy harvesting systems with varying power supply [14, 15] can use this to significantly extend the runtime [9]. Power consumption can be reduced further by gating (*i.e.,* disabling) unused clock sources, sub-topologies of the clock tree, or input clocks of inactive peripherals. Active power management can apply this dynamically [16].

We determine the magnitude of enabled power reductions by executing several micro-benchmarks on our test systems. Figure 4 shows the current drawn at different CPU frequencies set up by ScaleClock. We use the most compatible topologies in this experiment, in which the system is clocked by a scalable resistor-capacitor (RC) oscillator. Three aspects stand out. (*i*) a roughly linear relationship between frequency and power consumption in all configurations; (*ii*) both platforms show significantly lower consumption for division. This can be traced back to divisions requiring more cycles, which reduces the proportion of other CPU activities (*i.e.,* fetch, decode, memory access), lowering the switching activity; (*iii*) the ratio between current and frequency (*i.e.,* the slope) is significantly different on both targets.

Beyond instruction execution, the consumption is also affected by the memory access required to fetch data. Figure 5 shows this variation of power consumption when the same instructions use data from different memory locations. This effect cannot be generalized into simple rules, as it has

Table 1: Time overhead for transitioning from a static low power mode to executing a function.

Board	LPM	Exit Transition Time	Current
nucleo-l476rg	PM0	9.3 ms (to main)	410.3 nA
nucleo-l476rg	PM0	2.4 ms (to system init)	410.3 nA
nucleo-l476rg	PM1	9.6 µs (to callback)	8.4 µA
slstk3402a	PM0	18.9 µs (to callback)	7.4 µA
slstk3402a	PM1	18.8 µs (to callback)	7.8 µA

diverse impact, *e.g.,* for the *nucleo-l476rg* using data from registers reduces consumption for division while increasing it for multiplication. Both devices show a notable difference in how flash access affects the consumption. The generally more efficient *slstk3402a* exposes a much higher relative consumption when flash access is involved. Overall, these results highlight two relevant insights. First, operations of the executed task change the total consumption in a considerable magnitude. Second, precise knowledge of task behavior and device characteristics provide valuable runtime information for deciding whether a specific frequency configuration is energetically more efficient for that task.

MCU power consumption is commonly reduced with duty cycling via low-power modes (LPMs) in which any execution is stopped completely. This binary on-off operation mode lacks gradual control over performance and consumption during active operation. The ScaleClock dynamic frequency adaptation can be used in complement to achieve additional energy savings during the active periods, in which duty cycling has no effect, and it does not interfere with duty cycling applicability. Table 1 puts our previous consumption statistics of a slowed down (but still actively processing) MCU into perspective with the static consumption of low power modes, which completely stop ongoing execution. It also lists the time overhead incurred when transitioning from the sleeping LPM state back to operation mode as measured via general-purpose input/output (GPIO) instrumentation.

Improving Energy Efficiency. Applications that want to maximize energy efficiency can use ScaleClock to adaptively optimize the clock frequency at runtime. There is potential to dynamically save energy during execution as long as the CPU capacity is not fully utilized as noted in § 3.2. Tasks dominated by instructions that require access to CPU registers or random-access memory (RAM) scale well with frequency and are often most efficiently executed at the highest applicable frequency because minimizing execution time reduces static loss related to the active MCU [9]. Contrary, tasks slowed by I/O access or other asynchronous interactions execute more efficiently at a lower core frequency [10]. Losses due to dynamic switching of instructions without progressing a task are avoided in those cases, which quickly outweighs static losses. In practice, taking advantage of this unused potential requires either *a priori* knowledge about tasks or some assessment at runtime. ScaleClock follows the latter variant, the benefit of which we quantify in the following.

Impact of Dynamic Performance and Topology Control. Our analysis up to this point indicates that the energy saving

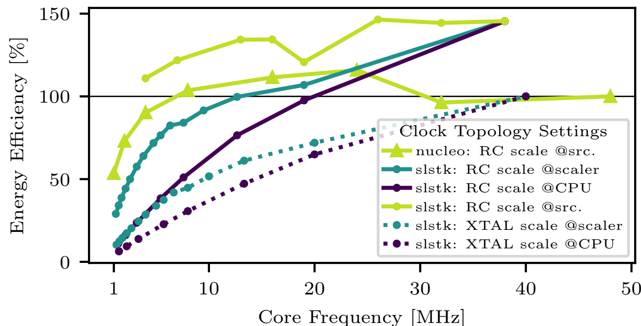


Figure 6: Energy efficiency while executing the same task at different topology and frequency configurations. Names refer to the clock source and the point in the topology where the frequency is scaled down. The 100 % baseline marks the highest frequency of the default topology.

potential depends on task characteristics. To capture other conditions which affect dynamic optimizations we measure how the energy efficiency of a given workload is affected by different topologies, core frequencies, and scaling methods.

The results in Figure 6 relate the converged average energy efficiency and core frequency for different topology and scaling variants. Displayed variants differ in type of clock source (RC vs XTAL) and the point within the topology path where the frequency adaptation happens (*i.e.,* as close as possible to either the source or CPU, or at an intermediate scaler). The workload includes all permutations of instructions and memory access variants used before ($\{$*add, mul, div*$\} \times \{$*reg, ram, flash*$\}$, see § 5.1). These operations are generally efficient at high frequencies as they are executed without busy waiting or asynchronous I/O. Therefore, the optimization potential for DVFS is low—aside from internal flash access that cannot keep up with the CPU speed.

We chose the 100 % baseline as the energy consumed at the highest frequency operation of (*i*) the *RC@src.* topology for the *nucleo-l476rg* or (*ii*) the *XTAL@scaler* topology for the *slstk3402a*. All configurations exhibit low efficiency at very low frequencies. The largest efficiency impact (more than 40 %) is obtained by selecting an RC source over a crystal oscillator (*XTAL* vs. *RC*). The noticeable drop at 20 MHz (*RC@src* on *slstk3402a*) follows from constraints of this particular system configuration which lead to an unfavorable ratio between flash speed and core frequency. Another notable effect shows the *nucleo-l476rg* device, at which the benefits of voltage scaling outweigh the efficiency penalty of lower frequencies between 8 and 24 MHz giving more than 15 % better energy efficiency. Comparing configurations with the same source and frequency (*e.g., XTAL@x* or *RC@x*) shows that energy efficiency improves with scaling a node that is closer to the clock source. We can conclude that already without dynamic optimizations ScaleClock can improve energy efficiency by about 15 % with scaling down voltage and frequency and over 40 % by switching the source topology.

Recalling the linear relation between frequency and current (see § 5.1), we now question how the potential to reduce power consumption can be translated into further energy savings for applications that do not require full CPU perfor-

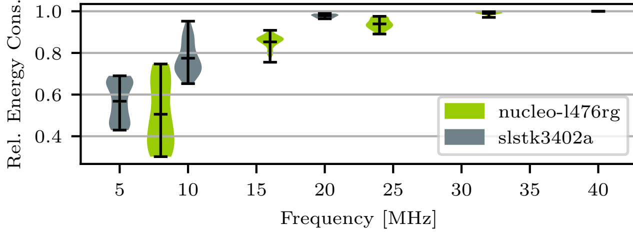


Figure 7: Energy distributions for task sets executed at their energy-optimal core frequency, relative to consumption at highest freq. (40 MHz). Ticks: extrema and mean values.

mance. We create a synthetic benchmark to explore the space of achievable energy savings. It consists of 100 tasks with varying portions of computational and time-dependent operations. We parameterize the task set from fully scalable computational workloads to purely time-dependent operations in order to cover the full range of possible workload characteristics between the extremes of scalability. We measure the energy consumption for each task at each frequency and the energy-optimal setting is identified per task. Regarding the use of synthetic benchmarks we note that the energy-related characteristics of a task are its execution time and its switching activity (*i.e.,* α as given in § 3). Varying those synthetically in a micro-benchmark reveals characteristics, but also transfers to other tasks as their timing is known (measured via PU assessment), α is task-specific, and the PU values are established per task. An exception is that flash wait-state adaptation may slightly affect α towards lower frequencies (by reducing the wasted cycles for bottle-necked flash access). Yet, this effect can be fully isolated by performing the PU-assessment without flash adaptation.

Figure 7 displays the distribution of energy consumptions for all task subsets at their energy-optimal frequencies. Energy savings of more than 60 % can be achieved if a task is most efficiently executed at a low core frequency (here 8 MHz). It is worth noting that only dynamic frequency scaling (DFS) is applied in this case, of which we previously saw a reduced efficiency if applied to computation-intensive tasks. Voltage scaling is expected to improve these further.

Assessing Performance Utilization. We now evaluate the ScaleClock mechanism for dynamically identifying the optimal frequency, *i.e.,* the PU metric for assessing the scaling potential of a task (see Equation 2). Figure 8 shows the distributions of PU values for the same task sets as a function of their optimal frequencies. Per platform we observe a strictly monotonic relation between the optimal frequency and the PU values. This leads us to the conclusion that it is viable to deduce energy-optimal frequencies for the tasks from their respective PU values. Hence, the metric justified its effectiveness for dynamically identifying the optimal frequency.

Stirring the DVFS functions from online assessments enables the system to automatically align performance with task demands. Figure 9 shows the current profile of the *nucleo-l476rg* MCU while executing two threads that perform workloads of different kind. Both threads start back-to-back and are triggered in an alternating pattern—a typical producer-consumer scenario. One benefits from a higher core frequency (*i.e.,* high PU, processing), while the other is most efficiently executed at a lower frequency (*i.e.,* low PU, acquisition). After the task is started, a stair pattern is visible related to the execution of the automatic PU-assessment that cycles through multiple available frequencies while both threads still execute. It is worth noting that the method introduced in § 3.2 is used with more than two frequencies here and the collected data of each frequency pair is used for the calculation of the PU value. Then the respective PU factors are derived for both threads and written out together with debug information via a serial connection. The duration of this step is governed by the serial communication while the overhead of the calculation is negligible. Thereafter, the DFS mechanism is enabled and reduces power consumption of the low PU thread by more than 70 %. Even though the lower frequency of the low PU thread also reduces the performance, the overall energy consumed for the same work is reduced by almost 40 %.

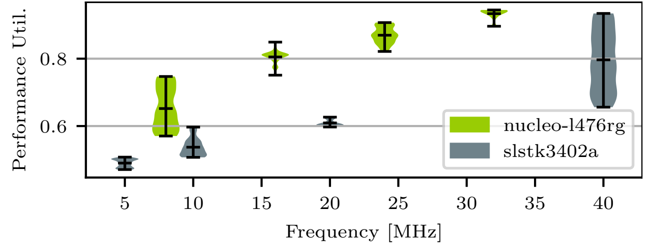


Figure 8: PU value distributions for the task sets executed at their energy-optimal core frequency. Ticks mark extrema and mean values.

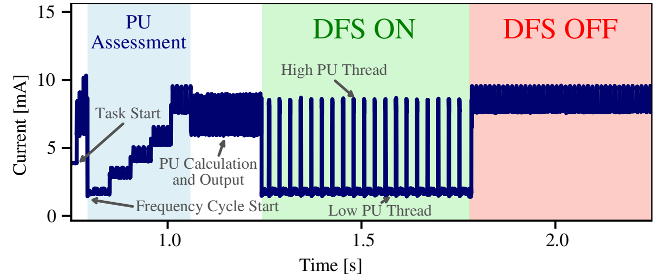


Figure 9: MCU current draw when performing online performance utilization assessment, comparing automatic DFS with static high frequency operation.

5.2 ScaleClock overhead

The overhead induced by ScaleClock is important for its utility. We measure overhead in terms of memory and runtime of the base operations. Base operations refer to initialization steps such as configuration exploration, as well as adaptation steps to change frequency and topology.

Memory. We differentiate between platform-agnostic parts and (static) data that encodes the platform-specific clock tree model. Memory of the generic parts such as the clock configurator or transition manager is dominated by instructions whereas the clock tree model mainly consists of static data. Table 2 lists the memory overhead of the different Scale-

Table 2: Memory used by the ScaleClock building blocks.

Component	ROM Size	RAM Size
Generic clock	24 bytes	-
Config register descriptor	32 bits	-
Shared register LUT entry	32 bits (single pointer)	-
Zero based mux option	32 bits (single pointer)	-
LUT mux option	64 bits (pointer + reg. value)	-
Clock configurator	5 kB	-
Clock manager	7.5 kB	172 bytes
Static clock tree model	2.5 kB	-
Task PU data (per thread)	-	32 bytes

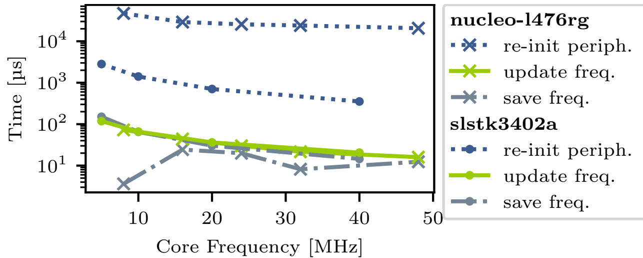


Figure 10: Timing overhead for simple frequency transitions performed at different clock frequencies.

Clock components. Overall, the generic part of the ScaleClock module uses ≈ 5 % of the total memory required by the test firmware used for this paper, whereas the platform specific clock tree model requires below 1 % of the memory.

Clock Tree Exploration. Proper clock tree configuration usually forces developers to carefully study hardware data sheets [17]. ScaleClock substantially eases this duty by providing a unified interface for exploring, configuring, and testing clock configurations interactively. On the `slstk3402a` platform all possible configurations for driving the core clock can be evaluated in less than three seconds. The faster `nucleo-l476rg` finishes this exploration in less than 500 ms. Nevertheless, the overhead of exploring configurations is considered non-critical because this rarely needs execution (*e.g.,* at initial boot) and the results can easily be cached. Albeit *a priori* exploration is preferable, ScaleClock is fully capable of performing this step on demand.

Time to Change Frequency. Switching between different frequencies is more critical because it is expected to run frequently, *e.g.,* when employing PU-based DVFS. Figure 10 shows the time overhead for clock transition steps as a function of the active frequency. The execution order matters as save and update are first run at the initial frequency while for re-init the target frequency is already in effect.

In detail, we find that the actual frequency change incurs significantly less overhead than post-processing (*i.e.,* re-init). This suggests that most impactful performance optimizations should focus on code for peripheral re-initialization. Driver init-code is also not likely to be optimized for execution time, since with static clock configuration it is only executed at boot. In many cases re-init steps can be avoided by using independent clock domains for peripherals or by ensuring that the same frequency is maintained for peripherals. It is worth noting that the save step is only needed for the current implementation to copy the previous state of configuration registers into the memory.

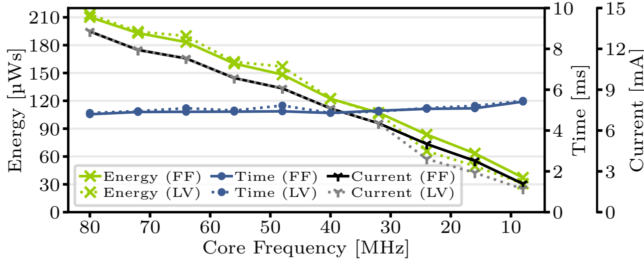
6 Case Study: Energy-Efficient Networking

Communication is central for IoT nodes. Whenever bulk data (*e.g.,* firmware updates) or live data (*e.g.,* health parameters) are transmitted, a significant share of the system energy is spent for communication, even if advanced network architectures such as edge processing are deployed. Systems with static clock configuration introduce a major mismatch between available CPU processing speed and (relatively low) throughput needed during transmissions. We now analyze performance benefits for senders and receivers while ScaleClock optimizes clock speeds for networking tasks.

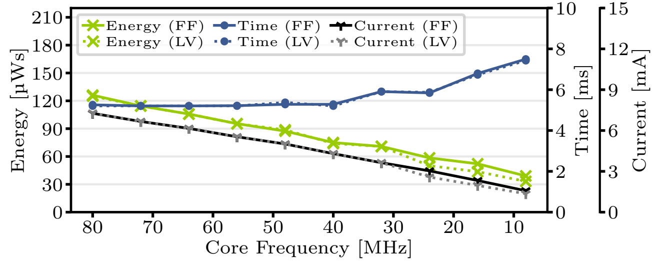
Basic setup. We implement a plain UDP sender-receiver scenario between two nodes. The sender transmits 64 packets of preconfigured payload to the receiver via a single link. We disable link-layer retransmissions (*i.e.,* ACK requests) and back-off mechanisms (*i.e.,* CSMA/CA) at the sender side to prevent blocking by the lower layer and isolate non-deterministic effects of the environment. This effectively maximizes the throughput towards the bandwidth-limited radio (*i.e.,* minimizes dynamic optimization potential) and therefore represents a pessimistic scenario. When measuring the impact at the receiver, we enable both mechanisms to maximize the incoming channel throughput at the receiver side. We conduct 64 runs of each of the following experiment settings and present (converged) averages.

We evaluate CPU power consumption and performance impact while applying DVFS via ScaleClock. As in our prior evaluations (see § 5), we conduct our experiments on the *nucleo-l476rg* evaluation board, extended by an AT86RF233 IEEE802.15.4 radio module that is connected via Serial Peripheral Interface (SPI). The RIOT firmware includes the default network stack *gnrc* on top of the *at86rf2xx* radio driver in its default configuration with an SPI target frequency of 5 MHz. Since lower CPU frequencies allow for the reduction of flash wait states and core voltage but both settings sometimes mutually exclude each other, we also investigate the impact of two policies that favor either fast flash access (FF) or low voltage (LV).

Impact of core clock frequency. In this experiment, we vary the core clock frequency. Figure 11(a) depicts a strong correlation between current and frequency but a weak correlation between frequency and transmission time (*i.e.,* throughput). This indicates significant energy savings at very low (temporal) performance penalty. When reducing the frequency from 80 MHz to 40 MHz, the consumed energy reduces by up to 42 % – whereas the time only increases by 1 %. At 1/10th of the frequency (8 MHz), energy is reduced by ≈ 82 % at a moderate timing penalty of ≈ 14 %. This result is line with our expectation because the radio imposes a bottleneck where the packet processing becomes negligible in relation to the time required for transmission. The LV and FF policies have no significant impact on transmission times, but the low voltage variant is able to further reduce energy consumption. The fact that network op-

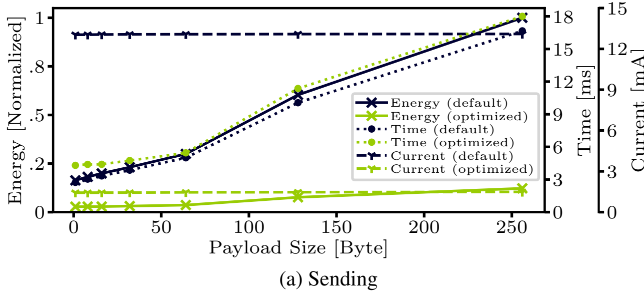


(a) Sending: Reducing core clock frequency significantly improves energy consumption at minor increase of transmission time.

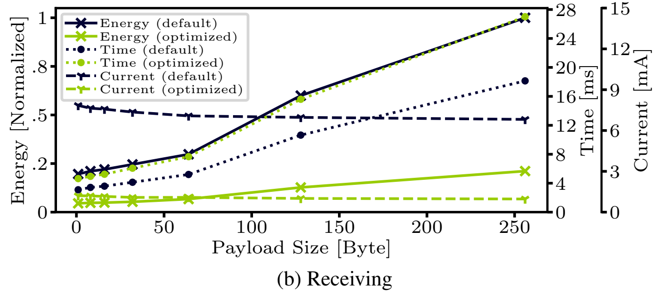


(b) Receiving: Reducing core clock frequency saves less energy and exposes higher performance penalty at low core frequencies.

Figure 11: Using different clock frequencies for sending and receiving 64 bytes payload based on UDP/6LoWPAN.



(a) Sending



(b) Receiving

Figure 12: Impact of different payload sizes on energy consumption and timing, comparing the default (high frequency) configuration and the most energy efficient configuration. Energy is normalized to highest consumption.

erations do not incur much flash access explains why lower voltage excels faster flash.

In contrast to the sender side, the receiver (Figure 11(b)) exhibits an overall lower consumption as it effectively requires fewer interactions with the radio. The CPU gets notified asynchronously by the radio once data is available, and can then read and process the packet. Setting the CPU frequency to half reduces the consumed energy by ≈ 40 % – again at a very small temporal performance penalty of 2 %. The relative energy savings for slowing down to 1/10th are slightly smaller (≈ 69 %), which is partially related to the significantly bigger performance impact of ≈ 44 % increase in time. Packet processing only starts after a complete reception, which—if decelerated—stretches the time to become ready for the next reception by freeing the frame buffer. This effectively causes more retransmissions. The sender runs at its default high core frequency and hence does not reduce stress towards the receiver. Coordinating clock management across nodes could potentially improve this further. In this paper we focus on the case of an unaltered environment, as in practice node performance is not assumed to be uniform and protocols must be able to cope with that.

These results clearly show that with ScaleClock both communication directions can leverage unused optimization potentials without sacrificing performance.

Impact of payload size. The potential for energy savings also depends on the packet size. Figure 12(a) shows the relation between energy consumption, transmission time, and average current for different payload sizes. All metrics of the default case are compared to the energy optimized configuration. Energy values are normalized to the highest energy value (*i.e.,* the unoptimized transmission of the biggest payload of 256 B). Notably across all packet sizes, the transmission times increase far less than the energy consumptions reduce. One reason for this is that even the smallest packet still induces a relevant amount of communication towards the radio module, which is also throughput limited because of the SPI bus. Additionally, even a single byte of payload comes with several bytes for involved communication protocols. The average current is flat for the default and the optimized variant, uncorrelated with the packet size. In the optimized case the relative time penalty grows towards the smallest payload (1 Byte) compared to higher payloads as the processing time (limited by the slower CPU speed) becomes a more dominant factor.

Three significant differences become apparent at the receiver-side (see Figure 12(b)): (*i*) the smaller average current consumption across the full set of measurements; (*ii*) the average current now noticeably increases for very small packets, which declines slower for bigger payloads; (*iii*) the temporal performance is overall more affected by the energy optimization. This is directly reflected by the level of energy savings in the optimized case, clearly visible when comparing the values of bigger payloads to the sender case.

Impact of clock topology. Last, we investigate the effect of adapting clock topology, *i.e.,* the different clock paths used to derive a specific frequency. Figure 13(a) compares the energy for transmission with different core clock topologies using either LV or FF policy. Here, all energy values are normalized to the energy consumption of transmitting the same

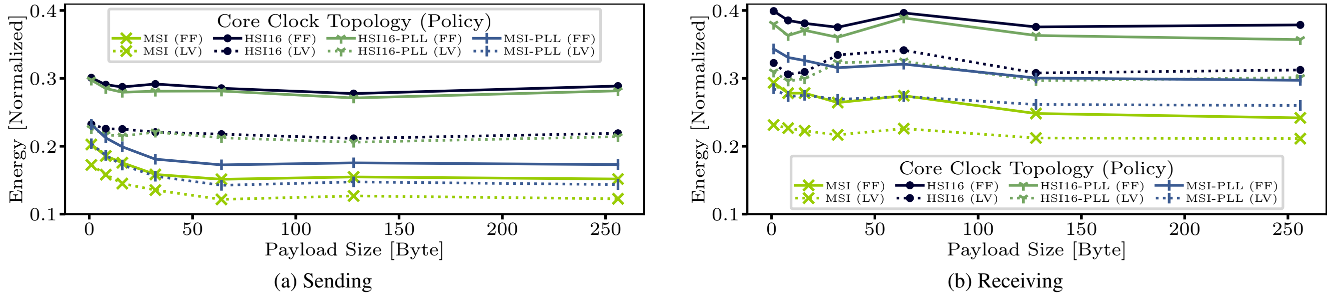


Figure 13: Energy impact of payload sizes, comparing the default (high frequency) and different topology configurations.

payload while operating at the default core clock topology (in this case `MSI-PLL` at static frequency of 80 MHz). In all cases, the energy efficiency improvements are slightly lower or very small payloads. It is possible to achieve significant energy savings of up to 70 % with FF and more than 77 % with LV policy for any topology setting. Yet, being able to switch the topology improves energy efficiency by at least another 6 % and up to 14 % when comparing to only adapting frequency via the top-most topology.

Figure 13(b) shows the equivalent results for the receiver side. The relative energy savings are slightly lower due to the generally lower current, as we already observed in Figure 11(b). This partially derives from a reduced bus communication with the radio module because of asynchronous triggering of processing steps. In contrast, the sender requires several steps that poll the radio for register state changes via the SPI bus. Different from the sender, there are also cases where different topology configurations outperform others. This can be seen for payloads below 50 B (`HSI16` vs `MSI-PLL`) and hints at certain dynamic application conditions that may change which topology is most preferable.

Overall, these results bolster our assumption that versatile dynamic clock reconfiguration is valuable for leveraging the full energy optimization potential of low-power hardware.

7 Related Work

Even after more than a decade of research on advanced dynamic energy management for constrained devices [4], the limit on power consumption keeps getting pushed lower [18]. More devices leave batteries behind and adopt intermittent power sources [19, 20]. Adaptability becomes a key feature for optimizing energy consumption under dynamic conditions [21] and development paradigms shift towards systems with more sophisticated abstractions [2]. Following this perspective, we briefly highlight related work that guides future directions which are not yet applicable to our domain. IoT related solutions are then discussed in more detail for the related areas of DVFS and energy efficient networking.

Simonović *et al.* [17] propose to encode clock trees using a template based formal language, which eventually could be provided by manufacturers. Such common representation of clock trees would mitigate the error-prone manual translation from data sheets. In a case study, they model a rather complex multiprocessor system-on-a-chip but do not share details on the application or quantitative performance results.

The Common Clock Framework (CCF) [22] of Linux handles clock configuration via its device tree [23]. CCF uses dynamic memory allocation, recursive operations, and large data structures, all of which are avoided on constrained MCUs. Furthermore, it can not explore configurations.

Liu *et al.* [24] propose EA-DVFS, an energy aware DVFS approach designed for energy harvesting systems that run tasks at an appropriate clock speed depending on energy availability. The authors evaluate their approach on a rather powerful embedded system with a processor running up to 1 GHz. They focus on real-time systems where the energy savings reduced deadline misses by more than 50 %.

DVFS continues to trickle down from data centers [25], personal computers and smartphones to more constrained target devices like wireless sensors [8, 26, 27]. Chiang *et al.* [10] propose a dynamic clock management system that aims for power reduction of tasks limited by IO operations and switches between distinct clocks for that purpose. Different to our work, they use an implicit mechanism that manages the active clock. We employ a proactive online assessment to provide fine-grained, energy-aware processing adaptations for tasks. Further, we consider an abstract clock model, advanced topology control via complex transitions, and voltage scaling in addition to frequency scaling.

Ahmed *et al.* [9] propose D^2VFS, a discrete DVFS variant for the intermittent device class. It scales down frequency and voltage as the supply capacitor empties during operation to increase clock cycles available for processing. In contrast to ScaleClock it limits possible frequencies to a small subset, instead of providing full control over the clock tree. Our proactive assessment method steers system performance towards given optimization goals. D^2VFS is reactive and does not aim for a generic clock configuration architecture.

Kulau *et al.* [8] propose IdealVolting, which leverages safety margins of the manufacturer voltage level specification by undervolting far below the specification limits, enabling energy savings of more than 40 %. In this work, we do not undervolt below recommended specification limits and employ hardware with integrated voltage scaling capability.

Antonio *et al.* [26] introduce a programmable power management on Wireless Sensor Network (WSN)-class devices that controls DVFS and power gating on the chip level. Their work is orthogonal to ours, as they focus on automatic power gating and DVFS implemented in hardware by a custom design on the register-transfer level. ScaleClock instead improves software control for devices that already pro-

vide those features. Since such devices are becoming more widely available [28], we investigate how these features can be uniformly exposed to upper software layers. We want to bridge the gap between specific hardware capabilities and energy-aware software which is platform-agnostic.

Improved energy efficiency of low power radio communication was approached on higher layers with topology control algorithms [29], and the physical layer via ultra low power wake up radios [30], or polymorphic radios that dynamically adapt between active and backscatter transmissions [31]. As our work exploits energy savings related to the common fundamental mismatch between CPU and radio throughput, we expect systems with such ultra low-power radios to also significantly benefit from our solution.

8 Conclusion and Outlook

In this paper, we proposed ScaleClock, an approach for generic online clock reconfiguration that suits constrained IoT devices. We reduced the complexity of managing clock dependencies by modeling clock trees as simple, reusable base components with a memory efficient way for encoding configuration parameters and constraints. Based on this lean model, we could enable dynamic exploration and reconfiguration of clock trees at runtime. We demonstrated the validity of our concept by implementing ScaleClock on two independent hardware platforms and evaluated its significant impact on energy savings when used for cross-platform DVFS.

With ScaleClock we have a tool at hand that allows for fine-grained control of core system parameters. This may help to master emerging challenges. Future directions of this research are threefold. First, additional control feedback mechanisms should be investigated across manifold application scenarios, including complex IoT networks and services. Second, our algorithms shall be studied in a supervised long-term deployment. Third, our generalized, hardware-agnostic access to the clock configuration shall give rise to new use cases and applications of timekeeping and signal generation.

Artifacts: All artifacts are available openly on GitHub `https://github.com/inetrg/RIOT/tree/ScaleClock`

Acknowledgements: Funding was provided by the Hamburg *sharing.city.college* of *ahoi.digital* and the BMBF project *PIVOT*.

9 References

[1] A. Dunkels *et al.*, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors." in *Proc. of IEEE LCN*. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 455–462.

[2] A. Levy *et al.*, "Multiprogramming a 64kB Computer Safely and Efficiently," in *Proc. of SOSP '17*. ACM, 2017, p. 234–251.

[3] E. Baccelli *et al.*, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, pp. 4428–4440, December 2018.

[4] X. Jiang *et al.*, "An Architecture for Energy Management in Wireless Sensor Networks," *SIGBED Review*, vol. 4, pp. 31–36, July 2007.

[5] M. Rottleuthner, *et al.*, "Sense Your Power: The ECO Approach to Energy Awareness for IoT Devices," *ACM TECS*, vol. 20, no. 3, pp. 24:1–24:25, March 2021.

[6] S. Eyerman and L. Eeckhout, "Fine-Grained DVFS Using on-Chip Regulators," *ACM TACO*, vol. 8, Feb. 2011.

[7] A. Castagnetti *et al.*, "Power Consumption Modeling for DVFS Exploitation," in *13th Euromicro Conf. on Digital Systems Design*. NJ, USA: IEEE, September 2010, pp. 579–586.

[8] U. Kulau *et al.*, "IdealVolting: Reliable Undervolting on Wireless Sensor Nodes," *ACM TOSN*, vol. 12, April 2016.

[9] S. Ahmed *et al.*, "Intermittent Computing with Dynamic Voltage and Frequency Scaling," in *Proc. of EWSN '20*. Feb. 2020, pp. 97–107.

[10] H. Chiang *et al.*, "Power Clocks: Dynamic Multi-Clock Management for Embedded Systems," in *Proc. of EWSN '21*. Feb. '21, p. 139–150.

[11] P. Levis *et al.*, "TinyOS: An Operating System for Sensor Networks," in *Ambient Intelligence*, Weber *et al.*, Springer, 2005, pp. 115–148.

[12] D. H. Kim *et al.*, "Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics," in *2015 IEEE 3rd Int. Conf. on CPS, Networks, and Applications*, 08 2015, pp. 78–85.

[13] Keithley, "Model DMM7510 7-1/2 Digit Graphical Sampling Multimeter Specifications," https://de.tek.com/sitewide-content/marketing-documents/m/o/d/model-dmm7510-7-1-2-digit-graphical-sampling-multimeter-specifications, October 2016.

[14] S. Sudevalayam and P. Kulkarni, "Energy Harvesting Sensor Nodes: Survey and Implications," *IEEE Communications Surveys & Tutorials*, vol. 13, pp. 443–461, March 2011.

[15] N. A. Bhatti *et al.*, "Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences," *ACM TOSN*, vol. 12, pp. 24:1–24:40, August 2016.

[16] A. Kansal *et al.*, "Power Management in Energy Harvesting Sensor Networks," *ACM TECS*, vol. 6, pp. 32–44, Sept. 2007.

[17] M. Simonović *et al.*, "An Approach to Modeling Clock Tree of a Complex System-on-Chip," in *2016 24th Telecommunications Forum (TELFOR)*. Piscataway, NJ, USA: IEEE, November 2016, pp. 1–4.

[18] G. Kazdaridis *et al.*, "Nano-Things: Pushing Sleep Current Consumption to the Limits in IoT Platforms," in *10th Int. Conf. on the Internet of Things*, ser. IoT '20. New York, NY, USA: ACM, 10 2020.

[19] D. Jagtap and P. Pannuto, "Reliable Energy Sources as a Foundation for Reliable Intermittent Systems," in *8th WS Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys)*. ACM, 11 2020, p. 22–28.

[20] A. Y. Majid *et al.*, "Continuous Sensing on Intermittent Power," in *2020 19th ACM/IEEE IPSN*, IEEE, April 2020, pp. 181–192.

[21] A. Bakar *et al.*, "REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 5, Sep. 2021.

[22] Mike Turquette. The Common Clk Framework. https://www.kernel.org/doc/Documentation/clk.txt.

[23] Embedded Linux Wiki. Device Tree Reference. https://elinux.org/Device_Tree_Reference.

[24] S. Liu *et al.*, "Energy Aware Dynamic Voltage and Frequency Selection for Real-Time Systems with Energy Harvesting," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, March 2008, pp. 236–241.

[25] S. Bhalachandra *et al.*, "Improving Energy Efficiency in Memory-Constrained Applications Using Core-Specific Power Control," in *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, ser. E2SC'17. New York, NY, USA: ACM, 2017.

[26] R. Antonio *et al.*, "Implementation of Dynamic Voltage Frequency Scaling on a Processor for Wireless Sensing Applications," in *TENCON 2017*. IEEE, Nov. 2017, pp. 2955–2960.

[27] T.-T. Zhu *et al.*, "Error-Resilient Integrated Clock Gate for Clock-Tree Power Optimization on a Wide Voltage IOT Processor," *IEEE Trans. Very Large Scale Int. (VLSI) Systems*, vol. 25, pp. 1681–1693, 2017.

[28] H.-S. Kim *et al.*, "System Architecture Directions for Post-SoC/32-bit Networked Sensors," in *Proc. of the 16th ACM SenSys*. New York, NY, USA: ACM, November 2018, pp. 264–277.

[29] J. Ma *et al.*, "Energy-Efficient Localized Topology Control Algorithms in IEEE 802.15.4-Based Sensor Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 711–720, 4 2007.

[30] R. Piyare *et al.*, "Ultra Low Power Wake-Up Radios: A Hardware and Networking Survey," *IEEE Communications Surveys Tutorials*, vol. 19, pp. 2117–2157, 7 2017.

[31] M. Rostami *et al.*, "Polymorphic Radios: A New Design Paradigm for Ultra-Low Power Communication," in *2018 ACM SIGCOMM*. New York, ACM, 2018, p. 446–460.