Seminar Paper: Dynamic Clock Control

Tobias Häberlein
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
tobias.haeberlein@fau.de

ABSTRACT

Modern embedded systems offer the ability to dynamically select and configure different clock sources at system runtime. *Dynamic Voltage and Frequency Scaling (DVFS)* systems can take advantage of these capabilities and select a certain clock frequency and core voltage depending on the current type of system load in order to save power. This seminar paper presents two approaches by Rottleuthner et al. and Chiang et al. that implement such DVFS mechanisms on modern microcontroller platforms. The selected approaches will then be compared and discussed in more detail.

1 INTRODUCTION

Today, the energy consumption of embedded systems plays an increasingly important role. *Microcontroller Units (MCUs)* are deployed in environments where either a reliable source of power cannot be guaranteed or only limited battery energy is available [8]. In order to consume as little energy as possible, many embedded applications therefore follow the *race-to-idle* strategy [5]: During active periods of the MCU, a high clock frequency is used to reach a low power sleep mode as quickly as possible. This approach is easy to implement since the clock frequency is determined before runtime and statically configured for the application. However, this causes typical IoT-applications to waste energy, especially if they consist of both compute-intensive operations and I/O-bound operations (e.g. reading values from an external sensor). This is because compute-intensive operations are most energy-efficient when a high frequency clock is used, while I/O-bound operations are most efficient when a low frequency clock is used [3]. Depending on the workload of the system, energy could be wasted if the application uses only one preconfigured clock frequency.

Modern MCU platforms offer the ability to change the currently used frequency by selecting between different clock sources dynamically during runtime. The available clock sources can differ not only in terms of clock frequency and power consumption, but also in terms of accuracy and temperature stability [12]. Adaptive software can take advantage of these configuration possibilities through *Dynamic Voltage and Frequency Scaling (DVFS)*, which is a mechanism that changes the core frequency and voltage during runtime depending on the current system load.

Changing the configuration of the active system clock is not always without difficulty: On the one hand, a frequency change may not always be possible at any point in time due to constraints of peripherals such as buses, sensors, and flash controllers [10]. As an example, changing the clock frequency during a data transmission via UART may lead to corrupt data packets if data is sent with the wrong baudrate after the frequency change. On the other hand, peripherals may place certain requirements on the clock used, like an *Analog-to-Digital-Converter (ADC)* that needs a specific minimum frequency to achieve a certain sample rate.

Another problem is the lack of software support for DVFS in common IoT operating systems [10]. Most systems only allow a statically preconfigured clock, only some of them support some kind of clock configuration change during runtime, albeit with many limitations. This is due to the fact that most MCU platforms come with complex *clock trees* consisting of components such as frequency multipliers, gates and multiplexers. Changing the clock frequency is therefore not a simple task, but can require the reconfiguration of many components in this tree. Mappings of platform-dependent clock trees at the hardware abstraction layer do not fully exist yet.

In this paper, two different proposed solutions for implementing DVFS in modern IoT operating systems are presented and discussed. The first approach by Rottleuthner et al. [10] focuses more on the abstraction of hardware-specific clock trees, while the approach of Chiang et al. [3] revolves around complying with constraints set by peripherals and finding the right time to change the clock configuration.

In the following, Section 2 discusses the advantages and challenges of DVFS on MCUs in more detail. Section 3 presents *Scale-Clock* [10], a system that introduces an abstract model of platform-specific clock trees to be used for clock configuration in the IoT operating system *RIOT*. Section 4 focuses on choosing the right clock configuration for DVFS at the right time, taking a closer look at the two solutions mentioned above [3, 10], which are then compared in Section 5. In Section 6, similar approaches are briefly discussed before concluding with a summary and outlook in Section 7.

2 DYNAMIC VOLTAGE AND FREQUENCY SCALING (DVFS)

DVFS is a technique used by modern operating systems to save energy when running applications by dynamically adjusting the frequency and voltage of the system at runtime. It takes advantage of the fact that power consumption is heavily dependent on the core clock frequency and voltage. This section describes the situations in which it is beneficial to adjust the clock frequency on modern MCUs.

2.1 Categorizing Power Consumption

The power consumption of CMOS chips in microcontrollers can be divided into a static and a dynamic part [2]. Static power consumption is caused by leakage currents through the transistors, even when they are not actively switching. Dynamic power consumption on the other hand is caused by switching activity in the system that causes capacitors to charge and discharge. Dynamic power consumption depends on the clock frequency, whereas static power consumption is independent of the frequency.

By dynamically adapting the clock frequency to the operations being performed, the system can save power. Compute-intensive operations should be executed at the highest possible clock frequency. This minimizes the static part of the power consumption, as the time spent per operation gets smaller at higher frequencies. I/O-bound operations, however, such as reading values from an external sensor, should be performed at the lowest possible clock frequency. In this case, the dynamic power consumption is minimized.

DVFS systems ensure that the appropriate clock frequency is used based on the current type of system load as described above. They then adjust the core voltage to the lowest possible value, which is a function of the selected frequency. For most MCUs however, the potential energy savings from changing the core voltage are rather limited. Instead, most savings are achieved by choosing an optimal clock frequency via *Dynamic Frequency Scaling (DFS)* [10].

During periods when the peripherals are inactive and the CPU is idle, the MCU can enter a low-power sleep mode to save energy. This is usually the responsibility of the operating system or the application developer and is not covered by DVFS. DVFS is only relevant during periods when the system is active.

2.2 Challenges in Embedded Systems

Existing DVFS systems, such as the one used in Linux, cannot be easily ported to microcontrollers, as they were often not designed with the memory and computational constraints of embedded systems in mind [10, 13]. In addition, unlike desktop platforms, modern MCU platforms offer a wide variety of different clock sources, each of which can have different properties. These clocks are configurable and dynamically switchable at system runtime [12]. New DVFS approaches must therefore not only take the constraints and limitations of microcontrollers into account, but also the wide variety of clock topologies of different MCU platforms.

3 MAPPING CLOCK TREES IN SOFTWARE

In order to save power with DVFS, it is necessary to have software support to be able to switch between different clock configurations of the underlying hardware platform. Rottleuthner et al. present an OS component called *ScaleClock* for this purpose that abstracts the hardware-specific clock trees by identifying some basic building blocks [10]. With ScaleClock, an application developer can simply select a desired (valid) frequency via an operating-system interface, whereupon a suitable clock-tree configuration is automatically selected and set. It is even possible to use ScaleClock to dynamically adjust the core clock frequency depending on the current type of system load. This is discussed in Section 4.2. The mapping of hardware-specific clock trees in ScaleClock is described in detail in the following sections.

3.1 Identifying Building Blocks

In principle, the clock trees of microcontroller platforms are very different. Some platforms offer very complex configuration options for various clocks (e.g. STM32 [12]), while others offer little or no customization options (e.g. nRF52 [7]). Despite these considerable differences, almost all clock trees can be described in software by using just a few basic building blocks. Figure 1 shows an example of what such a clock tree might look like.

A *source node* marks the start of each clock signal within the clock tree. This is usually an internal or external oscillator. The gate nodes (-○⁄ ○-) are the simplest elements of the clock tree. Depending on their configuration, they may or may not forward their input

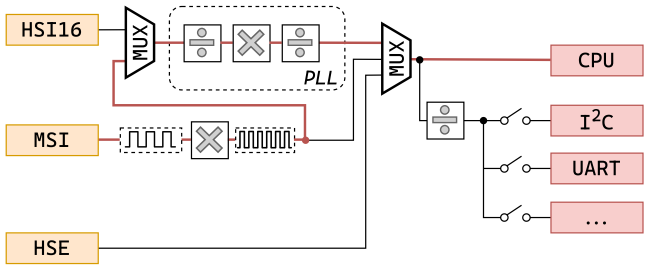

Figure 1: Simplified example of a clock tree, based on the STM32L47x platform [12]. The clock signal is generated by source nodes on the left and makes its way through various scalers, multiplexers and gates. It finally arrives at the consumer nodes on the right.

signal to their output port. Multiplexer nodes (`MUX`) have multiple clock signal inputs and forward one of them to their output. In the example in Figure 1, the CPU clock can be driven either by a *Phase-Locked Loop* (`PLL`), by the variable `MSI` oscillator or by an external high frequency oscillator (`HSE`). Scaler nodes are either multipliers (⊗) or dividers (÷). Multipliers multiply the incoming clock signal by a fixed or configurable factor, whereas dividers divide the signal by a factor. These nodes are important for peripherals such as I^{2}C or UART (*consumer nodes*), which expect a certain frequency that does not necessarily match the current system clock frequency.

According to the authors of ScaleClock, the presented basic components are generally sufficient to map all elements of a hardware-specific clock tree. More complex clock structures do exist, but they can usually be mapped by simply combining several of the basic components.

3.2 Switching between Clock Configurations

A path from a source node to an end node in the clock tree corresponds to a specific *topology*. In ScaleClock it is possible to switch between several of these topologies at runtime. A specific clock frequency at a consumer node can then be set by changing the scaling factors within the scaler nodes of the topology.

Depending on the currently active clock topology, switching to a new core frequency can be as simple as changing one or more scaling factors. In more complex cases, the system has to transition to a new topology by reconfiguring the output of a multiplexer node. Some platforms may however prohibit the configuration of active clock nodes, which can increase the complexity of transitioning from one to another topology. Furthermore, some topologies might have source clocks such as PLLs that require a certain amount of time to stabilize and deliver unstable frequencies in the meantime [12]. ScaleClock calls these cases *complex transitions* and handles them as follows: If it is not possible to switch directly between a source and a target clock configuration, an intermediate configuration is used temporarily. The system first switches to this intermediate configuration, and it only then applies the pending changes to the clock nodes of the target configuration. Afterwards, the system switches to the target configuration and disables nodes of the old configuration that are no longer in use.

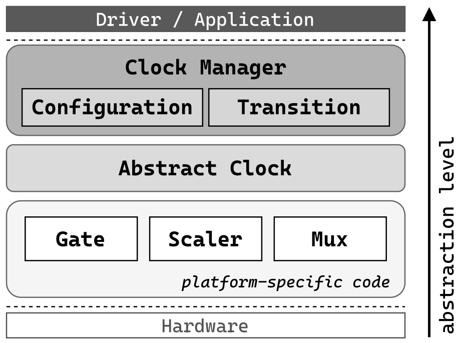

Figure 2: Relevant components of the ScaleClock architecture based on the publicly available source code [9].

3.3 ScaleClock Implementation

ScaleClock has been developed as a kernel module for the IoT operating system *RIOT* [9]. The most relevant parts of the architecture are shown in Figure 2, which is based on the publicly available source code of ScaleClock [9]. Different abstraction levels are provided to interact with the clock system. At the lowest level are the basic building blocks for mapping the clock tree in software, which were introduced in Section 3.1 (i.e., mux, scaler, gate, ...). These blocks are used in the platform-dependent part of the source code to reconstruct the hardware-specific clock tree according to datasheets of the manufacturer. It should be noted that even within a specific hardware platform, the exact structure of the clock tree may vary slightly from device to device. For example, the *NUCLEO-L476RG* development board which has been used by the authors of the paper can optionally be equipped with an external high-frequency oscillator [11]. ScaleClock uses predefined preprocessor macros that can be set by the application developer in order to distinguish among these different configuration options. In addition, the platform-specific part of the code also allows for the specification of possible restrictions with regard to maximum clock frequency and minimum voltage of clock nodes. Preferred topologies can also be specified, as well as the power consumption of each clock source, which plays a role in the selection of a specific clock configuration when using DVFS.

The *Abstract Clock* interface of ScaleClock allows platform-independent access to the clock tree. For example, the scaling factors of scaler nodes or the output of multiplexer nodes can be changed via the interface. At this level of abstraction, however, no additional restrictions are taken into account. This means that if a clock node has certain limitations regarding its maximum clock frequency, a higher clock frequency could still be set by increasing the scaling factors of its ancestor node(s).

The *Clock Manager* is located at the highest abstraction layer and provides methods that take a desired target frequency and then automatically select and configure a suitable clock topology. All restrictions within the clock tree are taken into account. Peripheral drivers may use the Clock Manager to temporarily block clock configuration changes, for example during data transfers. Additional driver constraints such as minimum frequency limitations are currently not handled by ScaleClock.

As the available memory space on MCUs is severely limited, special attention has been paid to a memory-efficient implementation of ScaleClock. For example, lookup tables are used wherever possible to reduce redundancy. Since most clock nodes can be configured using only a handful of memory-mapped registers, ScaleClock saves the memory address of these registers in one central lookup table and stores only a tiny index in the clock nodes. Further memory saving techniques have been employed which are not discussed in more detail here. All in all, the authors claim that with ScaleClock, the total memory required by the MCU firmware is increased by only about 5 %.

4 CHOOSING THE RIGHT CLOCK

In order to maximize power savings with DVFS, the clock configuration of the MCU should be adjusted to the current type of system load at the right time. As described in Section 2.1, when the current application needs to wait for peripherals, a low core clock frequency should be selected. Conversely, when the application is performing computationally intensive operations, it is best to use the highest possible frequency. In the following, two kernel-based dynamic clock-management systems are presented, each of which handles the distinction between these two cases differently.

4.1 Power Clocks

Chiang et al. present *Power Clocks* [3], a kernel-based DFS system that utilizes the key insight that compute-intensive operations are efficiently executed at fast clock frequencies, while I/O-heavy operations are most efficiently executed at lower clock frequencies. With Power Clocks, the operating system kernel dynamically selects the most energy-efficient clock based on the current number of active and requested peripherals.

4.1.1 Peripheral Constraints. In Power Clocks, the peripheral drivers must request a clock from the kernel before they are allowed to begin with their operation. In their request, they can specify certain requirements that the clock must meet. For example, they can require a certain minimum frequency, or only allow clocks that have a certain accuracy. Peripheral drivers can also specify whether they can cope with sudden clock changes during operation by setting a `no_jitter`-flag. Each peripheral has its own set of requirements: For example, drivers for ADC or UART are usually highly dependent on the current clock frequency and cannot tolerate abrupt clock changes. Drivers for SPI and I^{2}C, on the other hand, may be able to handle them, since the data transfer in these protocols is controlled by a dedicated clock line which is independent of the system-clock frequency.

4.1.2 Architecture. Power Clocks has been implemented for the IoT operating system *Tock*. It consists of two main components: The *ClockManager* is the central part of the DFS system. It takes care of selecting the most efficient clock during system runtime taking into account the constraints of the peripheral drivers that act as *ClockClients*. Each ClockClient must register with the ClockManager and inform it of its constraints before it starts operating.

4.1.3 Clock Change Algorithm. When an I/O operation is to be performed, the peripheral driver (ClockClient) must send a clock request to the central ClockManager. The ClockManager will only grant a request immediately under certain circumstances: To ensure that pending clock requests are not further delayed, the request must not have the `no_jitter` flag set. In addition, the client's requested minimum clock frequency must be compatible with the minimum clock frequencies of all other pending ClockClients.

If one of the above conditions is not met when a clock request is sent, the request is placed in a central FIFO queue. In this case, the ClockClient must wait at least until the next time the ClockManager selects a new clock frequency. A new clock frequency is chosen after the current I/O operation is finished and **all** threads of the system are yielded. This avoids situations where the ClockManager selects a new frequency, e.g. 1 MHz, but shortly after that a new clock request with a higher minimum frequency arrives, e.g. 4 MHz. By waiting until all threads in the system have yielded their operation, the ClockManager can make an optimal clock selection based on a global view of all requested I/O operations.

When selecting a new clock frequency, the ClockManager iterates over the queue from front to back and gradually forms the intersection of all clock frequencies of the pending requests. It starts with the supported clock frequencies of the first request, then forms the intersection of all supported clock frequencies of the second request, and so on. If at one point the intersection returns an empty result, the ClockManager skips the current request. The next time the clock selection takes place, this request will be the first one in the queue. From the final intersection, the ClockManager chooses the lowest possible clock frequency and configures the system clock accordingly. It then calls the callback function of each ClockClient whose clock request was granted, indicating to them that they can now start with their I/O operations.

If no ClockClient is currently running and there are no pending clock requests, Power Clocks assumes that the system is performing compute-intensive operations. In this case, it wants to switch to the highest possible clock frequency as quickly as possible in order to remain energy-efficient. However, an immediate clock change can lead to the problem of *thrashing*. This occurs when the system constantly switches between fast and slow clock speeds between I/O operations that are interrupted by short periods of driver code execution. Since clocks have to stabilize, and the clock change itself takes some time and consumes power, the ClockManager waits until the time quantum of the current thread has expired before performing a clock change. In Tock, the time quantum of threads is set to 10 ms, which the authors of Power Clocks believe is a good heuristic for estimating when the MCU is performing compute-intensive tasks. If no new clock requests have arrived after this time, the ClockManager automatically changes the clock frequency to the highest possible setting.

4.1.4 *Clock Configuration.*

Clock configuration in Power Clocks is an implicit mechanism that the authors do not describe in detail in the paper. All relevant clock configurations have been predetermined for each platform and are statically defined in the code. The system therefore only has to choose between one of the few predefined clock configurations at runtime. However, as explained in Section 3, configuring the core clock of microcontrollers can be a challenging task due to their intricate clock trees. Power Clocks lacks a more sophisticated approach for clock configuration, such as the one presented by Rottleuthner et al.

4.2 ScaleClock

ScaleClock [10] chooses a fundamentally different approach to determine the time of the clock change. It introduces a new metric that is used by the system to decide per thread if it should be executed at a high or low frequency.

4.2.1 *Performance Utilization.*

ScaleClock introduces a new metric called *Performance Utilization (PU)* to decide whether to run a thread at a high or low clock frequency. The metric is a measure of how well the execution time of the thread decreases as the clock frequency increases. Equation 1 shows the calculation rule for the PU metric. The active execution time t_{busy} of the thread at a frequency F_1 is set in relation to the execution time at a higher frequency F_2.

$$PU = \frac{t_{busy}(F_1)}{t_{busy}(F_2)} \cdot \frac{F_1}{F_2}, \quad F_1 < F_2 \tag{1}$$

Threads with a high PU value near 1 have a perfect scalability. At twice the clock speed, they run twice as fast. These threads are executed at the fastest possible clock frequency, so they are as energy efficient as possible. Threads with a relatively low PU value near 0, on the other hand, have very poor scalability. If the clock frequency is doubled, their execution time is only minimally reduced. This is typical for threads that need to wait for I/O operations to complete. Such threads are executed most efficiently at the lowest possible clock frequency.

4.2.2 *DVFS.*

To dynamically scale the clock frequency at system runtime, ScaleClock needs to know the PU value for each thread. The metric is determined at system startup for all threads by opportunistically changing the clock frequency during operation. The system then collects data for each thread at each frequency, such as the number of context switches and the busy and idle times. To do this, ScaleClock interacts with the operating system's scheduler. Afterwards, it is able to calculate the PU value in order to find an optimal target frequency for each thread. The optimum frequency is then set by the system's ClockManager before a thread is scheduled, using the interface described in Section 3.3.

ScaleClock is also able to minimize the core voltage of the system if desired. All constraints on the minimum voltage per selected clock frequency are thus hard-coded into the platform-specific code of ScaleClock [9]. Adjusting the voltage, however, is not always desirable, especially when using flash memory. In this case, lowering the core voltage may lead to an increase in read latency, as additional wait states have to be introduced when the CPU accesses the flash memory. ScaleClock therefore gives the application developer the option to choose between *Fast Flash* or *Low Voltage* in such cases.

5 EVALUATION

Both Power Clocks and ScaleClock promise significant power savings with their approaches over statically configured clock frequencies. Both systems offer the ability to use DFS to automatically adapt the system's clock frequency to the type of operations being performed without requiring the application developer to write any additional code. In the following, the two approaches are evaluated, and their limitations are shown.

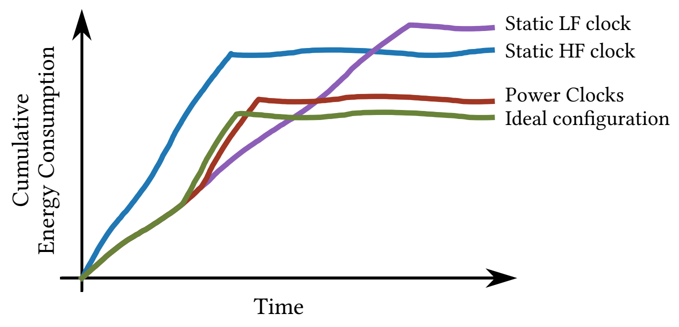

Figure 3: Sketch of the cumulative energy consumption for a typical IoT application. Static clock frequencies are compared to Power Clocks and to a hand-tuned, ideal clock configuration.

5.1 Power Clocks

The way the Power Clocks algorithm works is quite simple: If at least one I/O operation is currently being performed or at least one I/O operation is pending, Power Clocks selects the lowest possible clock frequency that satisfies the constraints of all clock requests. If no I/O operations are being performed, the fastest possible clock frequency is used.

Figure 3 is a sketch showing the cumulative energy consumption for a typical IoT application where Power Clocks can save a lot of energy. The application first collects data from an external sensor and then performs calculations on the read data. Several configurations of the application are compared with each other. In the case of the statically selected high-frequency (HF) clock, the system consumes the most energy. This is because a lot of energy is wasted waiting for the sensor to be read. The statically selected low frequency (LF) clock is more energy efficient during these periods, but is inefficient when the MCU is performing CPU-intensive calculations. With DFS, Power Clocks is able to combine the advantages of both static approaches and thus comes very close to the energy consumption of an ideal hand-tuned configuration. Overall, the authors claim that Power Clocks can achieve more than 25 % in energy savings compared to using a static clock frequency.

Power Clocks can save energy in many typical IoT use cases, but it also has some limitations, so that it may not always be the optimal solution. If an application consists only of CPU-intensive operations, then the selection of a static high-frequency clock makes the most sense. In such cases, the use of Power Clocks would add unnecessary overhead. If, on the other hand, an application constantly uses peripherals for which the `no_jitter`-flag is set, other peripherals could potentially starve to death if they expect a different clock frequency. This could be the case, for example, if an ADC is constantly sampling an input signal while another thread is simultaneously trying to read from an external sensor via a protocol that requires a different clock configuration.

In contrast to ScaleClock, Power Clocks is not able to dynamically adjust the system's core voltage. Also, the selection of the clock is an implicit mechanism that is not described further by the authors. An abstraction of hardware-specific clock trees as in ScaleClock does not exist.

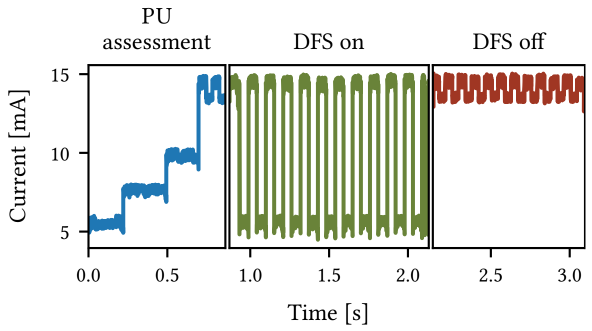

Figure 4: Power consumption of a sample application that consists of one CPU-bound thread and one I/O-bound thread while using ScaleClock. This test was performed on a *Nucleo-L476RG* from STMicroelectronics [11]. Power measurements were taken using a *PPK2* from Nordic Semiconductor [6].

5.2 ScaleClock

ScaleClock offers an abstraction of hardware-specific clock trees in the operating system and thus provides a platform-independent interface for clock configuration. The authors of the paper implement the system for two hardware platforms, and explain that support for other platforms is possible with a few new lines of code.

Figure 4 shows the power consumption of an example application that is using the ScaleClock DFS mechanism. The application consists of two threads that follow a producer-consumer pattern. Thread 1 reads data from a sensor via SPI and sends it to thread 2, which then performs calculations on this data. The first second of the system runtime consists of the PU assessment phase. Both threads are executed at different clock speeds in order to obtain as many data points as possible to calculate the PU metric. After the assessment phase, a PU value has been determined for both threads: Thread 1 has a low PU value and will therefore be executed at a low clock frequency later on. Conversely, thread 2 has a high PU value and will be executed at a high clock frequency. ScaleClock takes care of changing the clock frequency before each thread is scheduled. This can be clearly seen in the figure as the power consumption alternates between high and low values. In the last phase, DFS was disabled and the clock was set to a static frequency of 80 MHz. As expected, the power consumption is significantly higher in this case, even though the thread execution time is only slightly lower than in phase 2. Overall, the authors claim that ScaleClock could reduce the energy consumption of the device by up to 40 %, depending on the use case.

Unlike Power Clocks, ScaleClock determines the time of a clock change not by keeping track of the number of **globally** active / pending I/O operations, but via the PU metric on a **per-thread basis**. This has two major drawbacks:

(1) Determining the PU value per thread is time and, therefore, energy consuming, as different clock frequencies have to be tried out during system startup. However, it should be noted that it is not necessary to run this clock-configuration test at every startup, instead the application developer could run the PU assessment once and then hardcode the calculated values.

(2) The PU value only makes sense for threads that have exactly one task. As soon as a thread performs both CPU-intensive and I/O-intensive operations, energy is wasted because ScaleClock only selects one clock frequency per thread. The application developer must be aware of this limitation.

Another disadvantage of ScaleClock is that it ignores driver constraints. In its current form, peripheral drivers have no way of communicating their minimum and maximum clock frequencies to the system. The authors state that in such cases the relevant peripheral devices can be operated within their own clock domain. However, this would mean that some benefits of using a single clock source for the whole system would be lost, as using more clock sources also consumes more power.

Finally, it should be noted that switching between different clock configurations is a time-consuming task. Not only the actual configuration change takes time, but also the reinitialization of code for peripherals and timers. The application developer must therefore be careful to create threads that are active long enough to recoup these time and energy costs.

5.3 Combining Both Approaches

The two approaches presented for dynamically adjusting the clock frequency have different advantages and disadvantages. Compared to ScaleClock's PU metric, the Power Clocks algorithm for selecting the correct system clock incurs a lower time overhead and is independent of the individual thread's implementation. It also considers constraints of peripheral drivers. ScaleClock, on the other hand, provides a powerful and abstract interface that can interact with and configure the underlying hardware-specific clock trees. In the future, systems that use a combination of both approaches are conceivable. Parts of the Power Clocks algorithm could be integrated into the already well-structured code base of ScaleClock, combining the best aspects of both approaches.

6 SIMILAR APPROACHES

Dynamic power management has been an active field of research for many years, not only for ultra-low-power microcontrollers, but also for handheld platforms and devices that use energy harvesting. In the following, similar approaches to the two presented DVFS systems are briefly introduced.

6.1 Using Hardware Event Counters

Weissel and Bellosa propose an energy-aware scheduling policy for non-realtime systems called *Process Cruise Control* [14]. Very similar to ScaleClock and Power Clocks, they have found out that when the CPU has to wait, it is more energy efficient to run it at a lower frequency. The authors use hardware event counters to determine on a per-thread basis whether the system should use a low or a high clock frequency. As a primary indicator, they use the *memory requests per clock cycle* event to find out if a thread could benefit from a reduction in clock speed. This approach is similar to the one used in ScaleClock, where the PU metric correlates with the most efficient clock frequency of a thread. In contrast, however, Process Cruise Control is able to dynamically update its metrics while the thread is running, whereas the PU value of a thread in ScaleClock is only determined once.

6.2 Intermittent Devices

Ahmed et al. propose a DVFS approach for intermittently-computing devices called *D^2VFS* [1]. It takes into account the characteristics of the capacitors used in such devices, whose voltage drops much faster than in conventional batteries, as the devices extract energy from them. The authors explain that using static high-frequency clocks is generally more energy efficient, however, these clocks can only operate over a limited range of high voltages. As a result, if the supply voltage drops below a certain threshold, the device will shut down. Static low-frequency clocks can operate over a much wider supply voltage range, but using them all the time wastes energy, as discussed in the sections above. D^2VFS is a runtime technique that solves these issues by dynamically reconfiguring the clock frequency according to the current supply voltage. It uses voltage detectors in hardware that fire an interrupt as soon as a certain threshold is crossed. The main difference of this approach compared to ScaleClock and Power Clocks is that it is reactive and does not consider the current application workload.

6.3 Timing and Energy Requirements

One aspect that has been neglected so far is the determination of *Worst Case Execution Times (WCET)* and *Worst Case Energy Consumption (WCEC)* in order to meet timing and energy requirements of applications. Dengler et al. present a new approach called *FusionClock* that can handle time-triggered schedules and generate code to dynamically reconfigure the system's clock tree [4]. FusionClock uses a clock-tree-reconfiguration graph that pays special attention to the latency and energy consumption of transitioning between different clock configurations. Using minimal-flow analysis through this graph, FusionClock is able to find the worst-case-optimal energy demand of the application while still meeting its timing-related deadlines. Unlike ScaleClock and Power Clocks, the approach of Dengler et al. is able to give static runtime guarantees and can find resource-optimal clock-tree configurations. This approach, however, requires a periodic task model, which is not suitable for all IoT applications.

7 CONCLUSION AND OUTLOOK

Modern MCU platforms offer the ability to dynamically change the clock configuration at system runtime. The two presented DVFS systems ScaleClock and Power Clocks make use of this functionality to select the most energy-efficient clock frequency depending on the current system load. The authors of each approach therefore focused on different aspects. ScaleClock deals with the mapping of platform-dependent clock trees in the operating system kernel. PowerClock instead focuses on choosing the right clock at the right time, taking into account all peripheral constraints. The authors of the paper show that their algorithms can achieve energy savings of up to 40 % compared to statically chosen clock frequencies. As both DVFS approaches have been developed at operating system level and remain largely hidden from the application developer, it is quite conceivable that they could be integrated into popular IoT operating systems in the future. A combination of both approaches is also imaginable, using ScaleClock as the interface to modify the clock configuration and Power Clocks to select the right clock at the right time.

REFERENCES

[1] Saad Ahmed, Qurat ul Ain, Junaid Haroon Siddiqui, Luca Mottola, and Muhammad Hamad Alizai. 2020. Intermittent Computing with Dynamic Voltage and Frequency Scaling. In *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks*. 97–107.

[2] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. 2016. Energy-Aware Scheduling for Real-Time Systems: A Survey. *ACM Trans. Embed. Comput. Syst.* (2016). https://doi.org/10.1145/2808231

[3] Holly Chiang, Hudson Ayers, Daniel Giffin, Amit Levy, and Philip Levis. 2021. Power Clocks: Dynamic Multi-Clock Management for Embedded Systems. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks*. 139–150.

[4] Eva Dengler, Phillip Raffeck, Simon Schuster, and Peter Wägemann. 2023. FusionClock: Energy-Optimal Clock-Tree Reconfigurations for Energy-Constrained Real-Time Systems. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. 6:1–6:23.

[5] David H.K. Kim, Connor Imes, and Henry Hoffmann. 2015. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. 78–85.

[6] Nordic Semiconductor [n. d.]. *Power Profiler Kit II*. Nordic Semiconductor. https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2

[7] Nordic Semiconductor 2018. *nRF52840 Product Specification*. Nordic Semiconductor. https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.0.pdf v1.0.

[8] Tifenn Rault, Abdelmadjid Bouabdallah, and Yacine Challal. 2014. Energy efficiency in wireless sensor networks: A top-down survey. *Computer Networks* 67 (2014), 104–122.

[9] Michel Rottleuthner. 2023. *A ScaleClock Implementation for RIOT*. https://github.com/inetrg/RIOT/tree/ScaleClock

[10] Michel Rottleuthner, Thomas C Schmidt, and Matthias Wahlisch. 2023. Dynamic Clock Reconfiguration for the Constrained IoT and Its Application to Energy-Efficient Networking. In *Proceedings of the 2022 International Conference on Embedded Wireless Systems and Networks*. 168–179.

[11] STMicroelectronics 2020. *User Manual: STM32 Nucleo-64 boards (MB1136)*. STMicroelectronics. https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf 14.0.

[12] STMicroelectronics 2021. *STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs*. STMicroelectronics. https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf Rev. 9.

[13] Mike Turquette. [n. d.]. *The Common Clk Framework*. https://www.kernel.org/doc/Documentation/clk.txt

[14] Andreas Weissel and Frank Bellosa. 2002. Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 238–246.