Poster Abstract: Dynamic Multi-Clock Management for Embedded Systems

Holly Chiang
Stanford University
hchiang1@stanford.edu

Daniel Giffin
Stanford University
dbg@scs.stanford.edu

Amit Levy
Stanford University
levya@cs.stanford.edu

Philip Levis
Stanford University
pal@cs.stanford.edu

ABSTRACT

Modern microcontrollers come with a selection of clock sources that have widely differing frequencies and power consumptions. For applications whose workloads vary over time, dynamically changing the clock can provide significant energy savings. The varying constraints of embedded hardware environments and the complex interactions of multiprogrammed systems makes this approach burdensome to do in application logic. Power Clocks orchestrates energy optimizing clock management in the kernel, obviating the need for application involvement while still achieving acceptable performance for typical workloads. This poster describes Power Clocks's design and presents preliminary results.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**;

KEYWORDS

Clock management, Power optimization

ACM Reference Format:
Holly Chiang, Daniel Giffin, Amit Levy, and Philip Levis. 2018. Poster Abstract: Dynamic Multi-Clock Management for Embedded Systems. In *The 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18), November 4–7, 2018, Shenzhen, China.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3274783.3275176

SenSys '18, November 4–7, 2018, Shenzhen, China

ACM ISBN 978-1-4503-5952-8/18/11…$15.00
https://doi.org/10.1145/3274783.3275176

1 INTRODUCTION

Advances in microcontroller hardware and in embedded operating systems have opened up a new avenue for investigation. Newer microcontrollers provide multiple clock sources that vary in frequency, precision and power consumption; higher frequencies usually require more power, but may use less energy per cycle. The difference in power consumption may be orders of magnitude: the microcontroller used for evaluation here can draw anywhere between 12μA to almost 2mA, depending on the clock source.

To minimize energy consumption, a developer could select the lowest-power clock source that meets the needs of a particular application. However, an application's workload tends to be dynamic. For instance, an application could only require a fast clock 10% of the time, but by not changing the clock, it would end up using that clock 100% of the time. For applications with functional needs that vary significantly over time, clock selection should occur dynamically.

To effect this dynamism, the application could explicitly request particular clocks as necessary. However, this approach requires the developer to have detailed hardware knowledge in order to know which clocks are compatible with each possible peripheral state. This approach can also be bug-prone and would have to be reimplemented for every new application as well as for every chip the application runs on. Furthermore, explicit clock selection by applications becomes infeasible in light of recent advances in embedded operating systems, which allow multiple applications to run concurrently on a single chip[1].

Therefore, efficient clock selection requires a programming abstraction that is simple enough to extend to new applications, and is flexible enough to deal with concurrency. To address these requirements, we present Power Clocks, a dynamic clock-management system that provides energy savings without requiring any changes on the part of the application.

2 SYSTEM DESIGN

The aim of Power Clocks is to dynamically select a clock that minimizes overall energy usage despite operating with dynamic workloads. Some peripheral operations are clock dependent, for instance CPU instructions complete faster when a faster clock is applied. Other operations such as data transfers over SPI or UART, or ADC sampling are clock independent and operate at a preset rate, where using a clock frequency faster than the threshold needed to ensure correct operation has no effect on their runtimes. Power Clocks should select the appropriate clock per circumstance, such as using a clock with the lowest power per frequency ratio for clock dependent operations, and using the lowest power clock that allows correct operation for clock independent operations.

Power Clocks is implemented through ClockManager and ClockClient interfaces. The central clock controller implements ClockManager and peripherals implement ClockClient. During initialization, peripherals call `register`, so that the ClockManager has a reference to each peripheral. Modern microcontrollers can have dozens of peripherals, so requiring the controller to keep track of the state of each can be resource-intensive. Instead, each peripheral keeps track of its own state and notifies the manager when changes occur. Before beginning a operation that needs the clock, a peripheral calls `need_change` to check if the current clock is compatible with its clock requirements. If the current clock is not compatible, the peripheral calls `clock_change`. This signals the ClockManager to then query all registered peripherals for their clock requirements. Rather than having the controller figure out what clocks can be used for a given peripheral state, each peripheral directly reports a list of clocks compatible with its current state. The ClockManager then chooses the lowest power clock that meets all peripheral requirements and informs each registered ClockClient that the system clock has been changed by calling the `clock_updated` function on each ClockClient. Power Clocks assumes there will always be a clock that is compatible with any system state, as is the case when a static clock choice is used.

If a ClockClient has a clock dependent operation, it calls `lock` before it begins its operation. This prevents the ClockManager from changing the clock while the peripheral's operation is ongoing, ensuring operation correctness. `lock` returns a boolean indicating whether or not the peripheral obtained a lock. It will always return true unless a clock change is pending, in which case it returns false and the peripheral must wait until the lock is released. This forces all peripheral operations that require specific clocks to be implemented asynchronously. Once a peripheral's operation finishes, the peripheral calls `unlock`. If this results in the lock count dropping to zero, a `clock_change` is triggered.

3 RESULTS

To evaluate Power Clocks, we use an example application intended to mimic sensor operation. Once per second the application takes a millisecond worth of ultrasonic ADC samples, converts all the samples to millivolt values, and then writes the converted values to flash. This application has clock-dependent operations in the form of CPU calculations, and clock independent operations in the form of ADC sampling and flash erases and writes.

ADC sample rate	Static DFLL (μJ)	Static RC-FAST (μJ)	Hand Code (μJ)	Power Clocks (μJ)
125 ksps	719.4	574.2	491.7	528.0
250 ksps	907.5	739.2	633.6	650.1
300 ksps	973.5	801.9	693.0	679.8

Table 1: Comparision of the energy usage of the sensor application under different clock selection policies

The application was evaluated on a SAM4L Cortex M4 microcontroller and the Tock operating system. Table 1 compares the energy the application requires for its operations based on different clock selection policies. If an application developer is unfamiliar with the hardware, they are likely to go with the fastest clock, in this case DFLL, as it is most likely to work in all scenarios. If the developer is more familiar with the hardware, they may choose the lowest power clock that works for their application, in this case RCFAST, and see 18-20% reduction in energy usage. If a developer then wants to have further energy savings, they may specify clock changes within the application code for 29-32% energy savings. Finally, if the application developer uses Power Clocks, they can see 27-30% energy savings, close to that of an explicitly-managed, application-aware approach, without requiring any hardware specific knowledge.

4 ACKNOWLEDGMENT

We'd like to thank Petar Penkov for his help designing the interface. This work is supported by Intel/NSF CPS Security grant #1505684, the Secure Internet of Things Project, the Stanford Data Science Initiative, and gifts from Google, VMware, Analog Devices, and Qualcomm.

REFERENCES

[1] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786