Dynamic Multi-Clock Management for Embedded Systems

Holly Chiang, Hudson Ayers, Daniel Giffin, Amit Levy, Philip Levis
Deployments are limited by battery lifetime
Sensor energy profile

- Sensor deployment can be a very costly operation in terms of man hours or getting authorization
  - Maximize deployment -> extend battery life -> reduce energy usage

- Where do embedded applications spend their energy?
  - Most time spent in deep sleep
  - Most energy spent on brief active periods of I/O and computation

How can we reduce the energy used during active periods?
Clock sources

To support energy efficient applications, modern microcontrollers have multiple clock sources

- Faster clocks use more power but tend to be more energy efficient
- Up to two orders of magnitude difference in power draw depending on clock choice

<table>
<thead>
<tr>
<th>Clock</th>
<th>Frequency</th>
<th>Current</th>
<th>Startup</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCSYS</td>
<td>113600 Hz</td>
<td>12 µA</td>
<td>38 µs</td>
</tr>
<tr>
<td>RC1M</td>
<td>1 MHz</td>
<td>35 µA</td>
<td>-</td>
</tr>
<tr>
<td>RCFAST4M</td>
<td>4.3 MHz</td>
<td>90 µA</td>
<td>0.31 µs</td>
</tr>
<tr>
<td>RCFAST8M</td>
<td>8.2 MHz</td>
<td>130 µA</td>
<td>0.31 µs</td>
</tr>
<tr>
<td>RCFAST12M</td>
<td>12 MHz</td>
<td>180 µA</td>
<td>0.31 µs</td>
</tr>
<tr>
<td>OSC0</td>
<td>16 MHz</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RC80M</td>
<td>80 MHz</td>
<td>300 µA</td>
<td>1.72 µs</td>
</tr>
<tr>
<td>PLL</td>
<td>48-240 MHz</td>
<td>120-500 µA</td>
<td>30 µs</td>
</tr>
<tr>
<td>DPLL</td>
<td>20-150 MHz</td>
<td>122-1919 µA</td>
<td>100 µs</td>
</tr>
</tbody>
</table>
Motivation

● Most applications choose a static clock
  ○ Lowest power clock that meets application requirements
  ○ Energy efficient clock
  ○ Default clock

● Hand coding clock changes is difficult
  ○ Requires developer to have hardware specific peripheral knowledge
  ○ Bug prone
  ○ Not portable
  ○ Doesn’t work for multiple apps

Power Clocks: dynamic clock management in the kernel
Challenges

- **What** clock to change to
  - Peripherals have wide range of hardware-specific clock requirements
  - I/O vs compute bound peripherals

- **When** to change the clock
  - Ensure most efficient clock is always being used

- **How** to ensure correctness
  - Synchronous vs asynchronous peripherals

<table>
<thead>
<tr>
<th></th>
<th>Read (μJ)</th>
<th>Write (μJ)</th>
<th>Read (ms)</th>
<th>Write (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCSYS</td>
<td>22176</td>
<td>165</td>
<td>1235.3</td>
<td>5.2</td>
</tr>
<tr>
<td>RC1M</td>
<td>2046</td>
<td>185</td>
<td>147.6</td>
<td>5.6</td>
</tr>
<tr>
<td>RCFAST4M</td>
<td>581</td>
<td>198</td>
<td>34.5</td>
<td>5.6</td>
</tr>
<tr>
<td>RCFAST8M</td>
<td>393</td>
<td>215</td>
<td>19.0</td>
<td>5.6</td>
</tr>
<tr>
<td>RCFAST12M</td>
<td>327</td>
<td>215</td>
<td>13.2</td>
<td>5.4</td>
</tr>
<tr>
<td>OSC0</td>
<td>267</td>
<td>234</td>
<td>9.8</td>
<td>5.4</td>
</tr>
<tr>
<td>RC80M</td>
<td>221</td>
<td>320</td>
<td>4.7</td>
<td>5.3</td>
</tr>
<tr>
<td>PLL</td>
<td>215</td>
<td>320</td>
<td>4.2</td>
<td>5.4</td>
</tr>
<tr>
<td>DPLL</td>
<td>205</td>
<td>320</td>
<td>3.7</td>
<td>5.4</td>
</tr>
</tbody>
</table>
API

ClockManager
  set_max_frequency
  set_min_frequency
  set_clocklist
  set_need_lock
  enable_clock
  disable_clock

ClockClient
  clock_enabled

  sample(freq):
    //setup ADC
    //start sampling

  stop_sampling():
    //disable ADC

  sample(freq):
    CM.set_min_freq(freq*32)
    CM.set_need_lock()
    CM.enable_clock()

  clock_enabled():
    //setup ADC
    //start sampling

  stop_sampling():
    //disable ADC
    CM.disable_clock()
Compatible clock requirements
Incompatible clock requirements
Request buffer

Current Clock: A
lock_count: 1

Request 1:  

Request 2:  

Request 3:  

Current Clock: A
lock_count: 1

Request 3:  

Request 1:  

Request 2:  

Request 1:  

Request 2:  

Request 3:  
Request buffer

Current Clock: A
lock_count: 1

Request 1:  
A
B
C

Request 2:  
A
B
C

Request 3:  
A
B
C

Current Clock: A
lock_count: 1

Request 3:  
A
B
C

Request 4:  
A
B
C

...  

Request 1:  
A
B
C

Request 2:  
A
B
C

Will extend the starvation of a proceeding request
Request buffer

Current Clock: A
lock_count: 1

Request 1:  
Request 2:  
Request 3:  
Request 4:  

If head of request queue is blocked, a new request will run only if:
1. Request does not require a lock
2. Must exist a clock that satisfies both the requesting client and all proceeding clients in the queue
Limitations

● Only works if peripherals have limited runtime
  ○ Energy-limited applications keep peripheral operations short
● Timing delay caused by locking
  ○ Peripheral operations are already asynchronous
● Does not perform well on applications that work best with static clock
  ○ Adds inefficiency due to clock change overhead
Implementation

- **Tock - secure embedded OS written in Rust**
  - Allows multiple concurrent applications to run on a single microcontroller

- **imix development board**
  - SAM4L Cortex M4 (64kB RAM, 512 kB flash)
  - BLE and 802.15.4 radios, variety of sensors and I/O buses
  - Jumpers to measure power for each subsystem
ADC-Radio application

(a) Static RCFAST (4 MHz)

(b) Static DFLL (48 MHz)

(c) Hand Tuned

(d) Power Clocks
### Code overhead

<table>
<thead>
<tr>
<th></th>
<th>ROM</th>
<th>diff</th>
<th>RAM</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tock</td>
<td>131352</td>
<td>-</td>
<td>57344</td>
<td>-</td>
</tr>
<tr>
<td>+ClockManager</td>
<td>131404</td>
<td>52</td>
<td>57344</td>
<td>0</td>
</tr>
<tr>
<td>+ADC</td>
<td>134076</td>
<td>2672</td>
<td>57344</td>
<td>0</td>
</tr>
<tr>
<td>+Flash</td>
<td>134540</td>
<td>464</td>
<td>57348</td>
<td>4</td>
</tr>
<tr>
<td>+I2C</td>
<td>134736</td>
<td>196</td>
<td>57348</td>
<td>0</td>
</tr>
<tr>
<td>+SPI</td>
<td>134956</td>
<td>220</td>
<td>57348</td>
<td>0</td>
</tr>
<tr>
<td>+USART</td>
<td>135208</td>
<td>252</td>
<td>57348</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>3856</td>
<td></td>
<td>57348</td>
<td>4</td>
</tr>
</tbody>
</table>
## CPU cycle overhead

<table>
<thead>
<tr>
<th>Function</th>
<th>Changes clock</th>
<th>Lock</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable_clock</td>
<td>Yes</td>
<td>Yes</td>
<td>113-189</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td></td>
<td>110-217</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>722</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td></td>
<td>717</td>
</tr>
<tr>
<td>disable_clock</td>
<td></td>
<td>Yes</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>749</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td></td>
<td>587</td>
</tr>
</tbody>
</table>
Conclusion

- Embedded applications can significantly extend their deployment lifetimes by dynamically changing the clock in response to application workloads.
- Changing the clock manually places the engineering burden on the developer:
  - Requires hardware-specific knowledge, bug prone, not portable, doesn’t work when running multiple apps.
- Power clocks automatically manages clock changes in the kernel:
  - No application changes necessary.
  - 31% less energy than an optimal static clock.
  - Minimal code and CPU cycle overheads.