Goals for lecture

- Lab four
- Example scheduling algorithm design problem
  - Will initially focus on static scheduling
- Real-time operating systems
- Comparison of on-line and off-line scheduling code

Example problem: Static scheduling

- What is an FPGA?
- Why should real-time systems designers care about them?
- Multiprocessor static scheduling
- No preemption
- No overhead for subsequent execution of tasks of same type
- High cost to change task type
- Scheduling algorithm?

Rate monotonic scheduling

Main idea

- 1973, Liu and Layland derived optimal scheduling algorithm(s) for this problem
- Schedule the job with the smallest period (period = deadline) first
- Analyzed worst-case behavior on any task set of size n
- Found utilization bound: \( U(n) = n \cdot (2^{1/n} - 1) \)
- 0.828 at \( n = 2 \)
- As \( n \to \infty \), \( U(n) \to \log 2 = 0.693 \)
- Result: For any problem instance, if a valid schedule is possible, the processor need never spend more than 71% of its time idle

Homework index

1. Reading assignment . . . . . . . . . . . . . . 29
2. Lab six . . . . . . . . . . . . . . . . . . . . . 32

Lab four

- Talk with Promi SD101
- Sample sound at 3 kHz
- Multihop

Problem: Uniprocessor independent task scheduling

- Problem
  - Independent tasks
  - Each has a period = hard deadline
  - Zero-cost preemption
- How to solve?

Optimality and utilization for limited case

- Simply periodic: All task periods are integer multiples of all lesser task periods
- In this case, RMS/DMS optimal with utilization 1
- However, this case rare in practice
- Remains feasible, with decreased utilization bound, for in-phase tasks with arbitrary periods
Rate monotonic scheduling

- Constrained problem definition
- Over-allocation often results
- However, in practice utilization of 85\%-90\% common
  - Lose guarantee
- If phases known, can prove by generating instance

Critical instants

Main idea:
A job’s critical instant a time at which all possible concurrent
higher-priority jobs are also simultaneously released

Useful because it implies latest finish time

Proof sketch for RMS utilization bound

- Consider case in which no period exceeds twice the shortest period
- Find a pathological case
  - Utilization of 1 for some duration
  - Any decrease in period/deadline of longest-period task will cause deadline violations
  - Any increase in execution time will cause deadline violations

RMS worst-case utilization

- In-phase
  - $\forall k \text{ s.t. } 1 \leq k < n-1 : e_k = p_{k+1} - p_k$
  - $e_n = p_n - 2 \sum_{k=1}^{n-1} e_k$

Proof sketch for RMS utilization bound

- See if there is a way to increase utilization while meeting all deadlines
- Increase execution time of high-priority task
  - $e_i' = p_{i+1} - p_i + \epsilon = e_i + \epsilon$
  - Must compensate by decreasing another execution time
- This always results in decreased utilization
  - $e_k' = c_k - \epsilon$
  - $U' - U = e_i' + e_k' = \frac{e_i}{p_i} + \frac{e_k}{p_k} = \frac{p_i}{p_k} - \frac{p_k}{p_i} = \frac{p_i^2 - p_k^2}{p_ip_k}$
  - Note that $p_i < p_k \rightarrow U' > U$

Proof sketch for RMS utilization bound

- Get utilization as a function of adjacent task ratios
- Substitute execution times into $\sum_{k=1}^{n} \frac{e_k}{p_k}$
- Find minimum
- Extend to cases in which $p_n > 2 \cdot p_k$

Notes on RMS

- Other abbreviations exist (RMA)
- DMS better than or equal RMA when deadline $\neq$ period
- Why not use slack-based?
- What happens if resources are under-allocated and a deadline is missed?
Essential features of RTOSs

- Provides real-time scheduling algorithms or primitives
- Bounded execution time for OS services
  - Usually implies preemptive kernel
  - E.g., Linux can spend milliseconds handling interrupts, especially disk access

Threads

- Threads vs. processes: Shared vs. unshared resources
- OS impact: Windows vs. Linux
- Hardware impact: MMU

Threads vs. processes

- Threads: Low context switch overhead
- Threads: Sometimes the only real option, depending on hardware
- Processes: Safer, when hardware provides support
- Processes: Can have better performance when IPC limited

Software implementation of schedulers

- TinyOS
  - Light-weight threading executive
- \(\mu\)C/OS-II
  - Linux
  - Static list scheduler

TinyOS

- Most behavior event-driven
- High rate \(\rightarrow\) Livelock
- Research schedulers exist

\(\mu\)C/OS-II

- Similar to BD threads
- More flexible
- Bigger footprint

BD threads

- Brian Dean: Microcontroller hacker
- Simple priority-based thread scheduling executive
- Tiny footprint (fine for AVR)
- Low overhead
- No MMU requirements

Old linux scheduler

- Single run queue
- \(O(n)\) scheduling operation
- Allows dynamic goodness function
(1) scheduler in Linux 2.6

- Written by Ingo Molnar
- Splits run queue into two queues prioritized by goodness
- Requires static goodness function
  - No reliance on running process
- Compatible with preemptible kernel

Real-time linux

- Run linux as process under real-time executive
- Complicated programming model
- RTAI (Real-Time Application Interface) attempts to simplify
  - Colleagues still have problems at > 18 kHz control period

Real-time operating systems

- Embedded vs. real-time
- Dynamic memory allocation
- Schedulers: General-purpose vs. real-time
- Timers and clocks: Relationship with HW

Summary

- Static scheduling
- Example of utilization bound proof
- Introduction to real-time operating systems

Reading assignment

- Read Chapter 12 in J. W. S. Liu, Real-Time Systems.

Goals for lecture

- Lab four?
- Lab six
- Simulation of real-time operating systems
- Impact of modern architectural features

Lab four

- Please email or hand in the write-up for lab assignment four
- Problems? See me.
  - Will need everything from lab four working for lab six

Lab six

- Develop priority-based cooperative scheduler for TinyOS that keeps track of the percentage of idle time.
- Develop a tree routing algorithm for the sensor network.
- Send noise, light, and temperature data to a PPC, via the network root.
- Have motes respond to send audio samples and buzz commands.
- Play back or display this data on PPCs to verify the that the system functions.
Outline

• Introduction
• Role of real-time OS in embedded system
• Related work and contributions
• Examples of energy optimization
• Simulation infrastructure
• Results
• Conclusions

Introduction

• Real-Time Operating Systems are often used in embedded systems.
  • They simplify use of hardware, ease management of multiple tasks, and adhere to real-time constraints.
  • Power is important in many embedded systems with RTOSs.
  • RTOSs can consume significant amount of power.
  • They are re-used in many embedded systems.
  • They impact power consumed by application software.
  • RTOS power effects influence system-level design.

Role of RTOS in embedded system

Related work and contributions

• Instruction level power analysis
• System-level power simulation
  Y. Li and J. Henkel, Design Automation Conf., 1998
• Our work
  • First step towards detailed power analysis of RTOS
  • Applications: low-power RTOS, energy-efficient software architecture, incorporate RTOS effects in system design

Single task network interface

Checksum computation and output

TCP example

Simulated embedded system

• Easy to add new devices
• Cycle-accurate model
• Fujitsu board support library used in model
• μC/OS-II RTOS used
Multi-tasking network interface

RTOS power analysis used for process re-organization to reduce energy
21% reduction in energy consumption. Similar power consumption.

ABS example

Straight-forward ABS implementation

Periodically triggered ABS

Selectively triggered ABS

Selectively triggered ABS timing

63% reduction in energy and power consumption
Experimental results

Agent example

Experimental results

Optimization effects

TCP example:
• 20.5% energy reduction
• 0.2% power reduction
• RTOS directly accounted for 1% of system energy

ABS example:
• 63% energy reduction
• 63% power reduction
• RTOS directly accounted for 50% of system energy

Mailbox example: RTOS directly accounted for 99% of system energy

Semaphore example: RTOS directly accounted for 98.7% of system energy
Partial semaphore hierarchical results

<table>
<thead>
<tr>
<th>Function</th>
<th>Minimum Energy (µJ)</th>
<th>Maximum Energy (µJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>realstart</td>
<td>0.41</td>
<td>0.00</td>
</tr>
<tr>
<td>init</td>
<td>6.41 mJ total</td>
<td></td>
</tr>
<tr>
<td>timer</td>
<td>1.31</td>
<td>0.00</td>
</tr>
<tr>
<td>liteled</td>
<td>2.02 %</td>
<td>5.51 mJ total</td>
</tr>
<tr>
<td>startup</td>
<td>1.74 %</td>
<td></td>
</tr>
<tr>
<td>do</td>
<td>887.44</td>
<td>0.28</td>
</tr>
<tr>
<td>main</td>
<td>0.90 mJ total</td>
<td></td>
</tr>
<tr>
<td>save</td>
<td>1.56</td>
<td>0.00</td>
</tr>
<tr>
<td>data</td>
<td>0.28 %</td>
<td>0.31 mJ total</td>
</tr>
<tr>
<td>init</td>
<td>1.31</td>
<td>0.00</td>
</tr>
<tr>
<td>data</td>
<td>0.88</td>
<td>0.00</td>
</tr>
<tr>
<td>bss</td>
<td>2.72</td>
<td>0.01</td>
</tr>
<tr>
<td>cache</td>
<td>1.90</td>
<td>1.20</td>
</tr>
<tr>
<td>unf</td>
<td>155.18 mJ total</td>
<td>9.90 mJ total</td>
</tr>
<tr>
<td>trap</td>
<td>0.75</td>
<td>0.00</td>
</tr>
<tr>
<td>OSSemPend</td>
<td>2.48</td>
<td>0.78</td>
</tr>
<tr>
<td>win</td>
<td>1.20</td>
<td>6.33</td>
</tr>
<tr>
<td>unf</td>
<td>31.18 mJ total</td>
<td>9.82 %</td>
</tr>
<tr>
<td>trap</td>
<td>0.29</td>
<td>2.90 mJ total</td>
</tr>
<tr>
<td>OSSemPost</td>
<td>5.82</td>
<td>0.84</td>
</tr>
<tr>
<td>OSDisableInt</td>
<td>27.91</td>
<td>1.62</td>
</tr>
<tr>
<td>OSSched</td>
<td>3.76</td>
<td>1.18</td>
</tr>
<tr>
<td>OSSchedPend</td>
<td>19.07</td>
<td>6.00</td>
</tr>
<tr>
<td>OSSchedList</td>
<td>6.96</td>
<td>5.24</td>
</tr>
<tr>
<td>OSSemPend</td>
<td>6.54</td>
<td>29.04</td>
</tr>
<tr>
<td>etc.</td>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>

Energy per invocation for µC/OS-II services

<table>
<thead>
<tr>
<th>Service</th>
<th>Minimum Energy (µJ)</th>
<th>Maximum Energy (µJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSSemPend</td>
<td>6.54</td>
<td>73.64</td>
</tr>
<tr>
<td>OSMemPut</td>
<td>5.38</td>
<td>7.91</td>
</tr>
<tr>
<td>OSSemInit</td>
<td>6.41</td>
<td>8.22</td>
</tr>
<tr>
<td>OSMemGet</td>
<td>5.82</td>
<td>84.55</td>
</tr>
<tr>
<td>OSSched</td>
<td>27.91</td>
<td>1.62</td>
</tr>
<tr>
<td>OSSchedList</td>
<td>20.10</td>
<td>20.93</td>
</tr>
<tr>
<td>OSSemCreate</td>
<td>19.40</td>
<td>19.75</td>
</tr>
<tr>
<td>OSSemInit</td>
<td>1727.70</td>
<td>1823.26</td>
</tr>
<tr>
<td>OSSemPend</td>
<td>27.41</td>
<td>27.47</td>
</tr>
<tr>
<td>OSMboxCreate</td>
<td>7.07</td>
<td>82.91</td>
</tr>
<tr>
<td>OSMboxPut</td>
<td>8.22</td>
<td>84.55</td>
</tr>
<tr>
<td>OSMboxPend</td>
<td>6.38</td>
<td>7.91</td>
</tr>
<tr>
<td>OSSQInit</td>
<td>6.96</td>
<td>52.34</td>
</tr>
<tr>
<td>OSSched</td>
<td>108.89</td>
<td>258.53</td>
</tr>
<tr>
<td>OSSchedList</td>
<td>27.87</td>
<td>29.04</td>
</tr>
<tr>
<td>OSSemPend</td>
<td>0.09</td>
<td>0.00</td>
</tr>
<tr>
<td>exceptionHandler</td>
<td>0.09</td>
<td>0.00</td>
</tr>
<tr>
<td>printf</td>
<td>0.09</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Conclusions

- RTOS can significantly impact power
- RTOS power analysis can improve application software design
- Applications
  - Low-power RTOS design
  - Energy-efficient software architecture
  - Consider RTOS effects during system design

Summary

- Labs
- Simulation of real-time operating systems
- Impact of modern architectural features

Impact of modern architectural features

- Memory hierarchy
- Bus protocols ISA vs. PCI
- Pipelining
- Superscalar execution
- SIMD
- VLIW