Harmonized Scheduling + Performance Monitoring & Optimization

Raj Rajkumar

Lecture #12

Outline

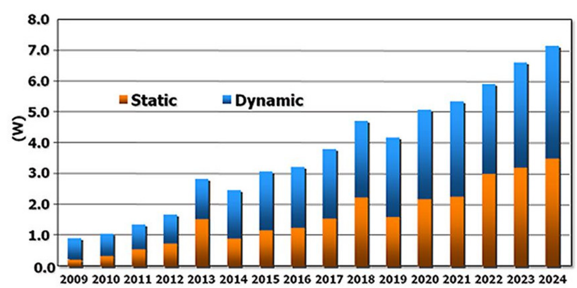
- Power and time constraints of microcontrollers
- Rate-Harmonized Scheduling
- Energy-Saving Rate-Harmonized Scheduling+
- Energy-Saving Rate-Monotonic Scheduling

CMOS Power Dissipation

- $P_{total} = P_{dynamic} + P_{static}$
- **Dynamic Switching Power**
  - $P_{dynamic} = K * C_L * V_{dd}^2 * f$
  - Due to charging and discharging of output capacitances
  - Can be reduced using Voltage and Frequency Scaling (VFS)
- **Static Leakage Power**
  - $P_{static} = V_{dd} * I_{leakage}$
  - Reduced using low-power **sleep states**
    - power gating and/or clock gating

Slow Down or Sleep ?

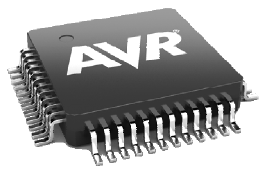
- At fabrication technologies smaller than 65 nm, static leakage power dominates the total power consumption of CMOS-based VLSI circuits.



SoC Consumer Portable Power Trend [Source: ITRS, 2010 Update]

Slow Down or Sleep ?

- Many low-power microcontrollers do not support VFS but support sleep states.



CPU Power Consumption Parameters

Processor	Frequency (MHz)	Active Power (mW)	Idle Power (mW)	Sleep Power (uW)	Sleep to Idle (ms)	Idle to Active (us)
Atmega 1281 *AVR*	8	23	6.6	16	12	6
NXP K22 *ARM Cortex-M4*	120	81.09	31.89	54.3	0.140	5.7

Power Modes of Microcontrollers

Power state	Power (mW)	Upward Transition Time
Active	30 mW	n/a
Idle	6 mW	6 us
Sleep	5 uW	10 ms

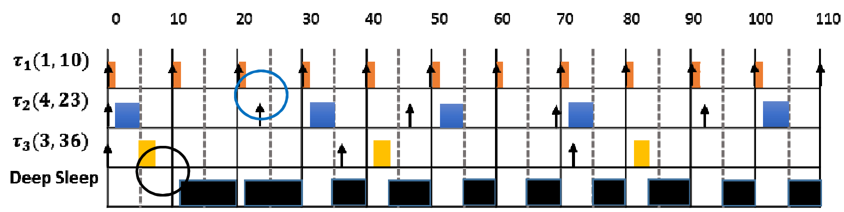
- Power Management: maximize the Sleep-time of processors
  - given {Sleep, Idle, Active} modes of operation

System Model

- **Periodic Independent Tasks** $\tau_i: (C_i, T_i, D_i)$
  - C_i: Worst-case execution time (WCET) of any job of task τ_i
  - T_i: Period
  - D_i: Relative deadline
  - U_i: Task Processor Utilization (C_i/T_i)
- Fixed priorities assigned to tasks following the Rate-Monotonic Scheduling policy

Rate-Harmonized Scheduling (RHS)*

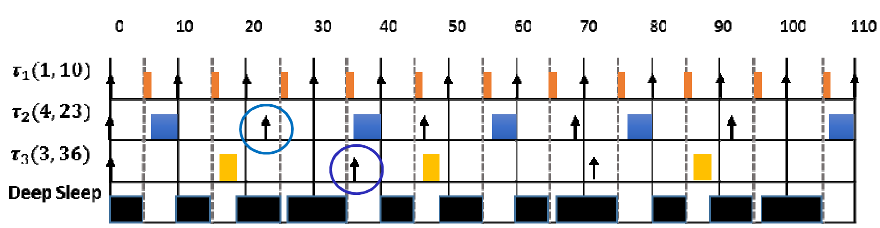
- Pick a ***harmonizing period***
  - ≤ the shortest period in task set
  - same phasing as the highest priority task
- **Harmonization**: tasks when released become eligible to execute ***only*** at the ***next*** harmonizing period boundary



Rate-Harmonized Scheduling ($T_{Harmonizing} = 10$)

Energy-Saving Rate-Harmonized Scheduling (ES-RHS)*

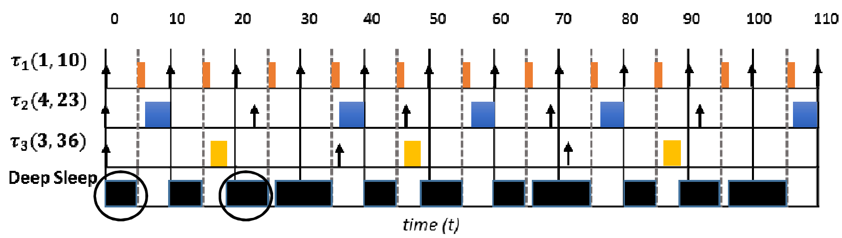
- **Create an Energy Saver (forced sleep) task with**
  - period = T_{sleep}, harmonizing period => *highest system priority*
  - budget = $C_{Sleep} \geq C_{SleepMin}$



Energy-Saving RHS ($C_{sleep} = 5, T_{sleep} = 10$)

ES-RHS Properties

- Every idle duration in the ES-RHS schedule will ***precede*** and be ***contiguous*** with a forced sleep duration.
- ***Every idle duration*** can be utilized to put the processor into deep sleep.
  - ***Optimal*** sleep utilization (energy savings)



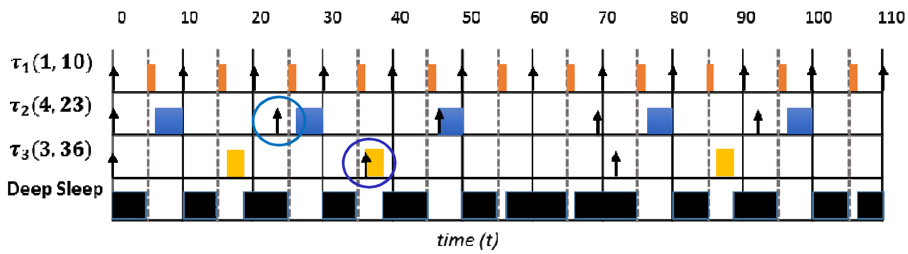
Energy-Saving RHS ($C_{sleep} = 5, T_{sleep} = 10$)

Extended Definition of Harmonization

- **A task is eligible to execute when**:
  - *the processor is busy (executing a regular task),* OR
  - *a Harmonizing Period boundary has been reached*

Energy-Saving Rate-Harmonized Scheduling + (ES-RHS +)

- Uses the ***re-defined*** notion of harmonization
- Tasks can become eligible to execute ***earlier*** than in ES-RHS



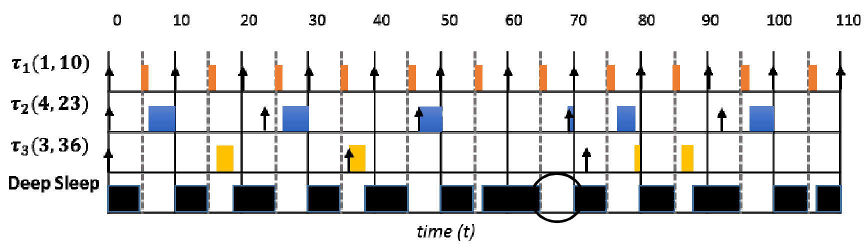
Energy-Saving RHS+ ($C_{sleep} = 5, T_{sleep} = 10$)

ES-RHS+ Properties

- ***Preserves*** the ***optimality*** of ES-RHS
  - Clustering of all idle durations with the energy saver task
- ***Enhances*** schedulability
  - Response Time recurrence relation for task τ_i in ES-RHS+
    - $W_0 = C_i + T_{sleep} - C_{sleep}$
    - $W_{k+1} = C_i + T_{sleep} - C_{sleep} + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j$
  - Response Time recurrence relation for task τ_i in ES-RHS
    - $W_0 = C_i + T_{sleep}$
    - $W_{k+1} = C_i + T_{sleep} + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j$
- Worst-case blocking ***reduces*** from T_{sleep} to $T_{sleep} - C_{sleep}$
  - Better schedulability and task response time
- ***Less restrictive*** and ***easier*** to implement than ES-RHS.

Energy-Saving Rate-Monotonic Scheduling (ES-RMS)

- ***Practical*** extension to RMS
- **Create an Energy Saver task with**
  - period = $T_{sleep} \leq T_1 \rightarrow$ the *highest priority*
  - budget = $C_{Sleep} \geq C_{SleepMin}$



Energy-Saving RMS ($C_{sleep}=5, T_{sleep}=10$)

ES-RMS Properties

- ES-RMS has ***better schedulability*** than ES-RHS+
  - Response Time recurrence relation for task τ_i in ES-RHS+
    - $W_0 = C_i + T_{sleep} - C_{sleep}$
    - $W_{k+1} = C_i + T_{sleep} - C_{sleep} + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j$
  - Response Time recurrence relation for task τ_i in ES-RMS
    - $W_0 = C_i$
    - $W_{k+1} = C_i + \left\lceil \frac{W_k}{T_{sleep}} \right\rceil C_{sleep} + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j$
- Worst-case blocking factor is ***eliminated***
  - Better schedulability
  - Lesser ***total deep sleep***, but greater ***forced sleep*** utilization

Comparative Evaluation

- **Methodology**
  - Simulations using randomly generated task sets
- ES-RHS/ES-RHS+ better than RMS
  - up to 33% better energy-savings on Firefly motes
- In most cases, ES-RMS yields ***deep sleep utilization*** very ***close*** to the optimal (on average 1.9% - 6.7% difference)
  - practical for use in many operating systems
- ES-RMS can provide greater ***forced sleep*** utilization
  - Up to 18% greater forced sleep duration than ES-RHS+
  - Useful for energy-savings on multi-core scheduling
- Estimated Energy Savings
  - NXP K22 → *ES-RHS+:* 40.57 mW, *ES-RMS:* 41.65 mW → (2.59%)
  - AT1281 → *ES-RHS+:* 11.50 mW, *ES-RMS:* 11.73 mW → (1.96%)

RHS Conclusions

- Power and time constraints of Microcontrollers
- Rate-Harmonized Scheduling
- Energy-Saving Rate-Harmonized Scheduling (+)
- Energy-Saving Rate-Monotonic Scheduling

Tools

- Optimization in Traditional Enviroments
- Monitoring and Optimization Requirements in Real-Time Environments
- Performance Monitoring Tools
  - ARM (Advanced Real-time Monitor)
  - WindView from Wind River
  - TimeTrace from TimeSys
  - Linux Trace Toolkit

Traditional Monitoring and Optimization Tools

- Symbolic debuggers
  - Single-step through logic
  - Modify registers, memory and variables
  - Set breakpoints
- Lots of **`printf`** statements
- Optimizing compilers
  - Intra-procedural optimization
    - Optimize code within a procedure
  - Inter-procedural optimization
    - Optimize code across procedures
  - Optimizing compilers....
    - `gcc -On` *// as n gets larger, more optimizations applied*

Intra-Procedural Optimization

- Unwinding of loops

```
for (i = 0; i < 4; i++) {
    x += 4y;
}
```

```
x += 4y;
x += 4y;
x += 4y;
x += 4y;
```

```
z = 4y;
x += z;
x += z;
x += z;
x += z;
```

- Dead-code elimination

```
x = 2;
if (x > 3) {
    …
}
```

```
x = 2;
```

- Smart arithmetic

```
x = y*17;
```

```
x = (y << 4) + y;
```

```
x = y*28;
```

```
x = (y << 5) – (y << 2);
```


Inter-Procedural Optimization

```
integer b;          // variable "global" to the procedure Silly.

int Silly(int a, int x) {
   if (x < 0)   a = x + b;
   else a = -6;
}

main()
{
  int a, x;        // These variables are visible to Silly only as params.

  x = 7; b = 5;
  Silly(a, x); printf("%d", x);
  Silly(x, a); printf("%d", x);
  Silly(b, b); printf("%d", b);
}
```

If arguments passed by value, no effect!

If arguments are passed by reference,

```
x = 7; b = 5;
if (x < 0) a = x + b; else a = -6; print x;   // a is changed.
if (a < 0) x = a + b; else x = -6; print x;   // params are swapped
if (b < 0) then b = b + b else b = -6; print b;
```

- The compiler can follow the constants along the logic and find that the predicates of the *if* statements are constant and so...

```
x = 7; b = 5;
a = -6; printf("7");       // b is not referenced
x = -1; printf("-1");      // b is referenced...
b = -6; printf("-6");      // b is modified as parameter
```

- Since `a, b` and `x` deliver nothing to the outside world, there is no point in this code either, and so the result is

```
print 7;
print -1;
print -6;
```


Real-Time Systems

- Many traditional techniques can be applied…
  … carefully

- Must be very careful about side effects

```
foo() {
   …
   x = 5;  // may seem like dead code to a traditional
     compiler
   // could actually be writing to an I/O device
}
```

- Such variables are usually tagged with the keyword "**`volatile`**"

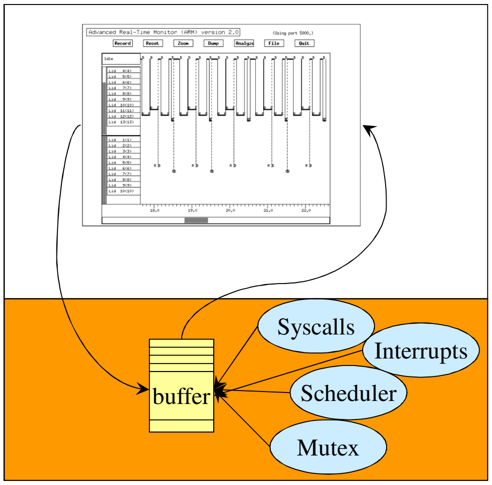
Other Requirements

- Inserting breakpoints that stop program will change timing behavior
- Adding `printf` statements can change timing behavior
- Recursive calls can necessitate big stacks consuming potentially valuable memory.
  - Want to perform memory *vs.* CPU tradeoffs
- Want to know what is happening when
  - What states tasks are in at different times
  - Which task is communicating with whom
  - What interrupts are happening and when
  - What the execution times are for different tasks
  - What the periods of different tasks are
  - What the budget is for reservations (or aperiodic servers)
  - What the system overheads are
  - Study input jitter and output jitter

Advanced Real-Time Monitor (ARM)

- A performance monitoring tool
- Designed to analyze and visualize the runtime behavior of target nodes in real time.
  - Allows user to reach into a remote target and view the scheduling events
  - Extracted using event taps in RT-Mach.

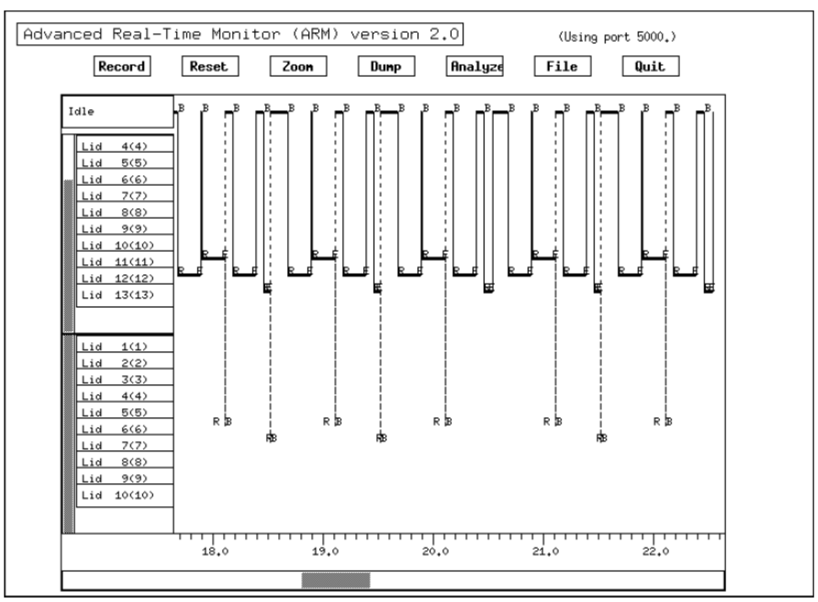
ARM Architecture



1. OS registers “events of interest”
   - System calls and arguments
   - Scheduler actions
   - Task states
   - Suspensions and Activations

   each with a high-resolution timestamp
2. Store in kernel buffer
3. Application makes system call periodically to obtain event buffer
4. Application plots data and interacts with user.

ARM Screen-Shot



ARM Messages

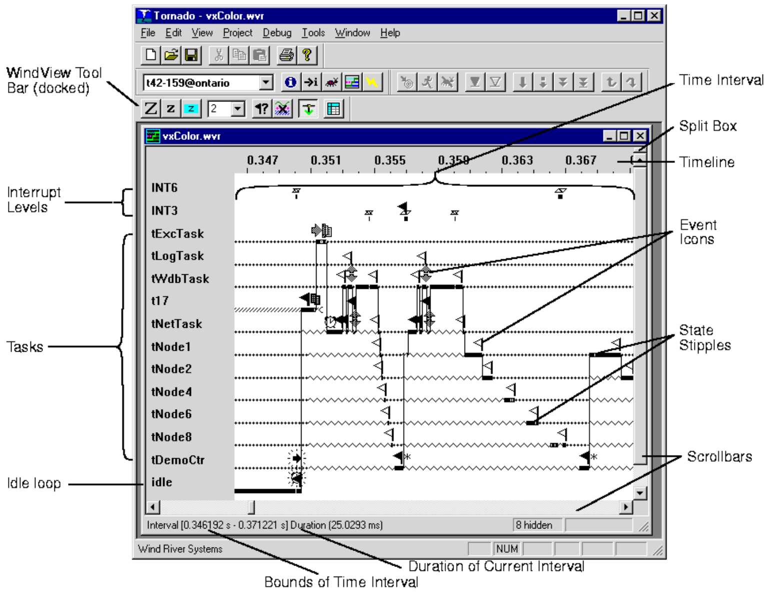
```
Time: 18850 ms.
Periodic tasks  : 1
Aperiodic tasks : 10
Total CPU utilization: 0.533 (cyclic tasks (0.444))  (acyclic tasks (0.089))
Meet deadline : 64
Missed deadline: 0
  aborted: 0
Events: 1088 events  57.719 per second
```


Cautionary Notes

- The execution time of one instance of (say a periodic) task depends upon
  - # of instructions executed
  - Branches taken or not taken in code (data dependence)
  - # of loop iterations (more data dependence)
  - Overheads of context switching
  - Caching behavior
    - In both instruction and data caches
  - Demand-paging behavior: usually disabled for real-time tasks
    - Must block paging of code, data, stack *and* heap segments
  - Interference on system bus
  - Conflicts on I/O devices
  - Cycle-stealing on main memory
  - Speculative execution within processor
  - …

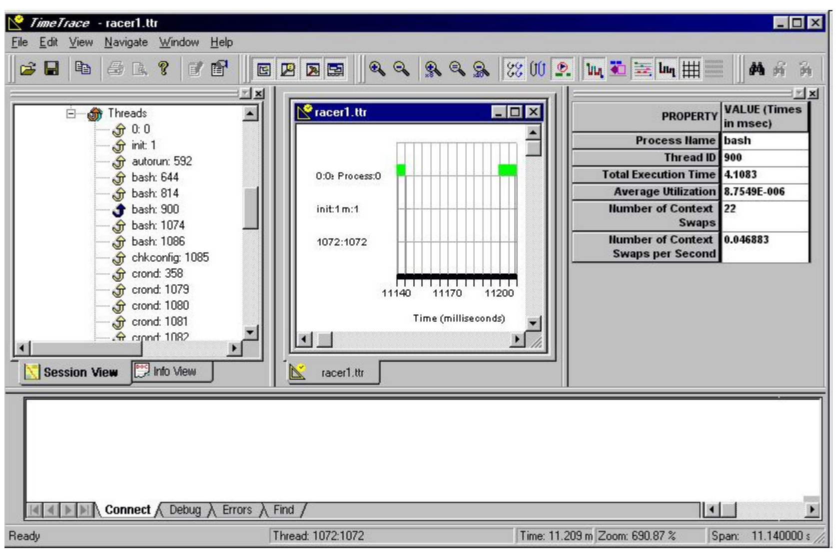
$\left.\begin{array}{l}\text{Overheads of context switching} \\ \dots \\ \text{…}\end{array}\right\}$ System-dependent

WindView from Wind River



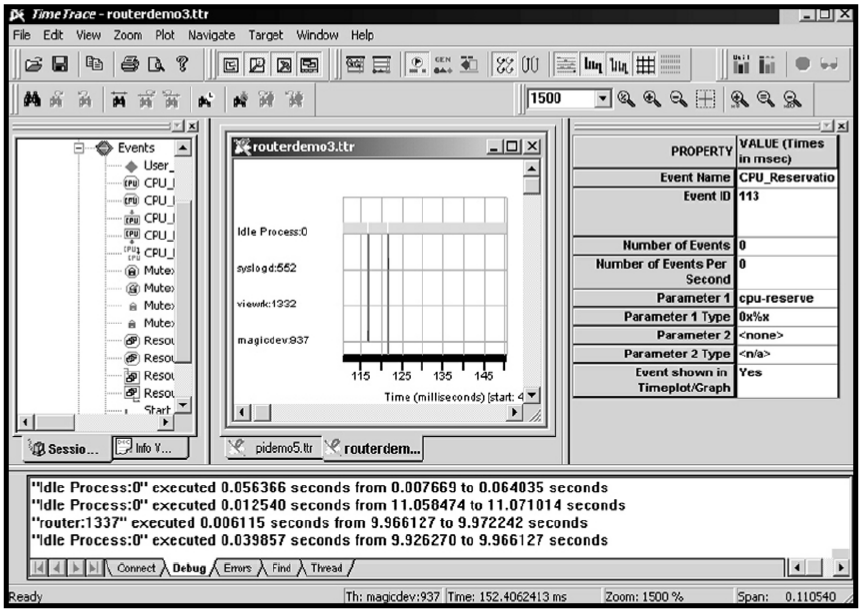
Electrical & Computer ENGINEERING | 18-648: Embedded Real-Time Systems | Carnegie Mellon

TimeTrace from TimeSys



Electrical & Computer ENGINEERING | 18-648: Embedded Real-Time Systems | Carnegie Mellon

Real-Time Profiling Using TimeTrace

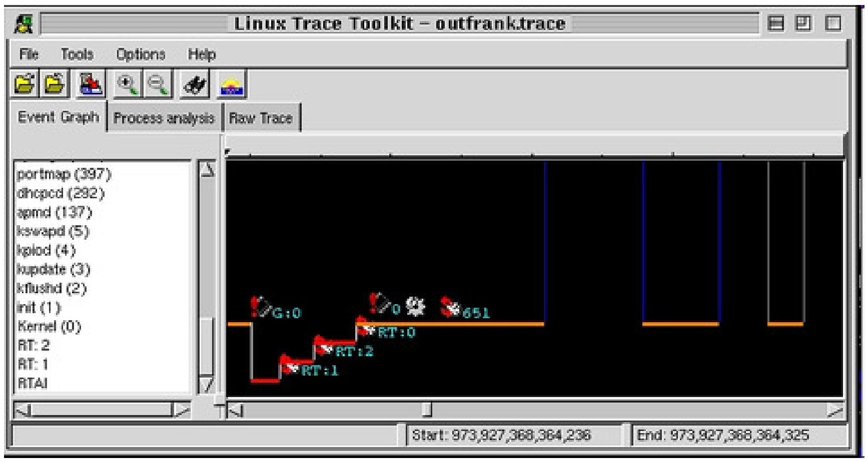


Some Features of TimeTrace

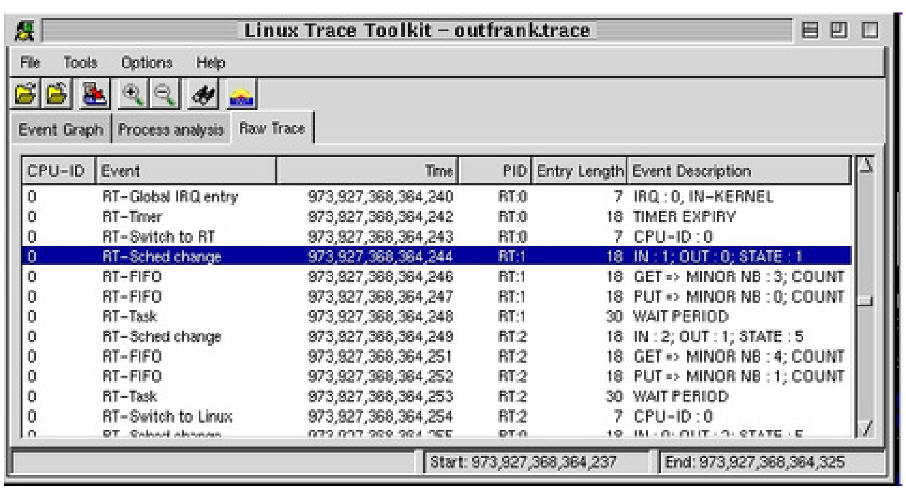
- Understand periodicity of tasks
  - Track period values
- Real-time data collection and plotting

Task Interactions with Linux Trace Toolkit

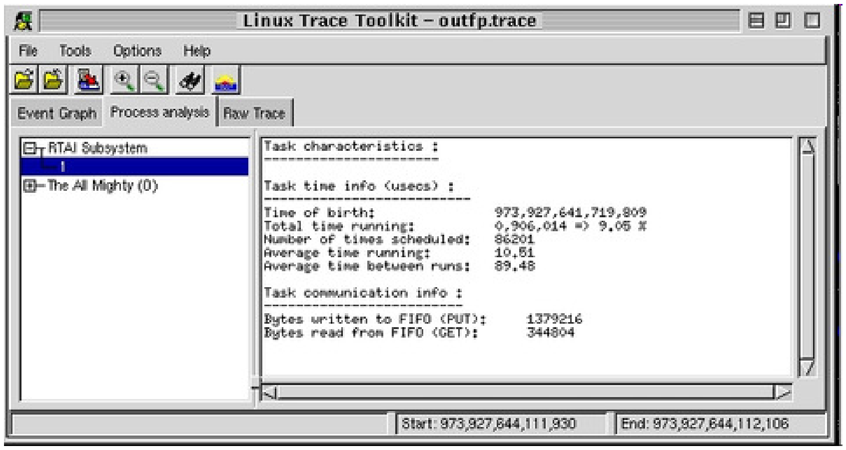
- Open source version available for Linux



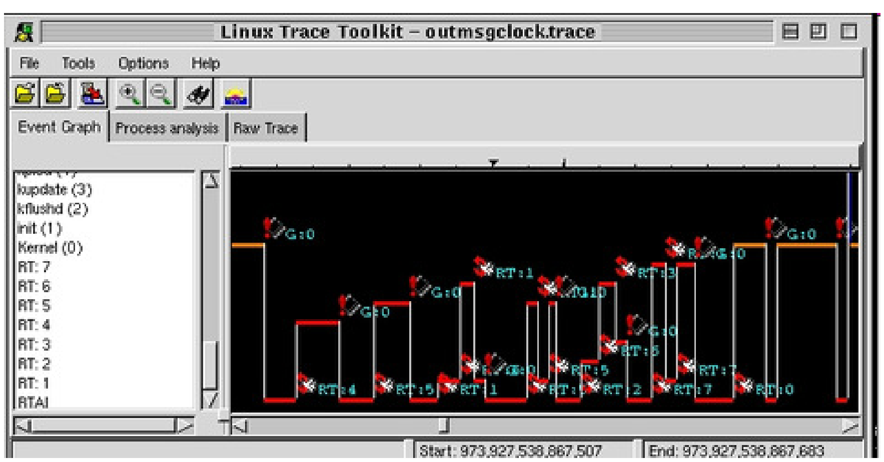
Raw Data from Linux Trace Toolkit



Execution Statistics Linux Trace Toolkit



More Interactions in Linux Trace Toolkit



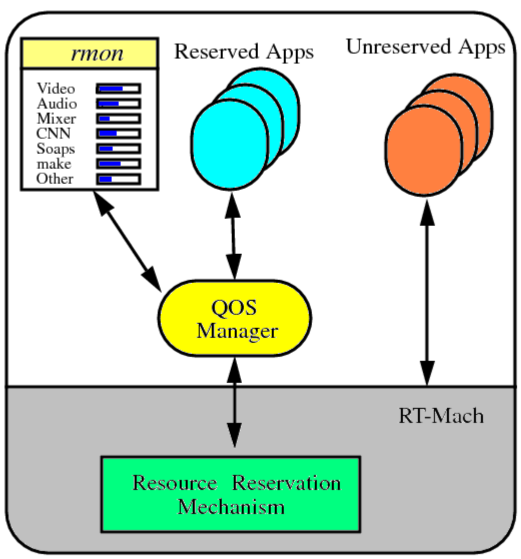
Working With Reservations

18-648: Embedded Real-Time Systems

Electrical & Computer ENGINEERING

Carnegie Mellon

Reservation Monitoring and Adaptation



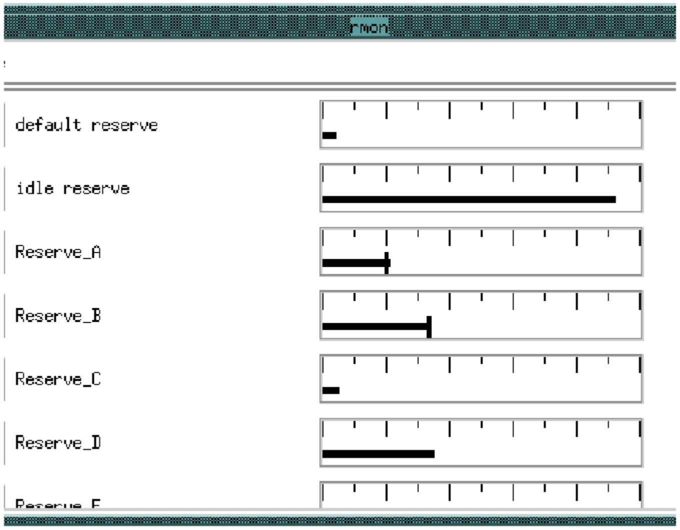
18-648: Embedded Real-Time Systems

Electrical & Computer ENGINEERING

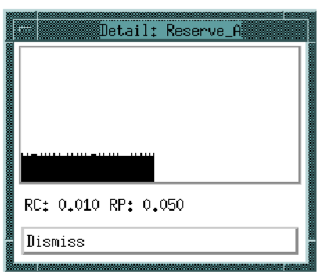
Carnegie Mellon

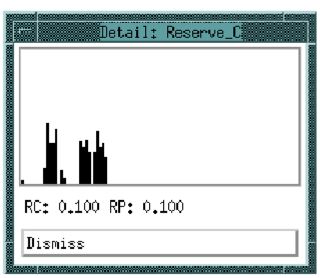
Reservation Monitoring

- **rmon** from Carnegie Mellon



More `rmon` Features

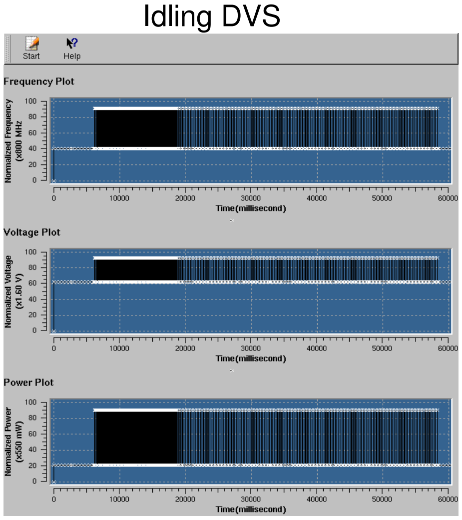




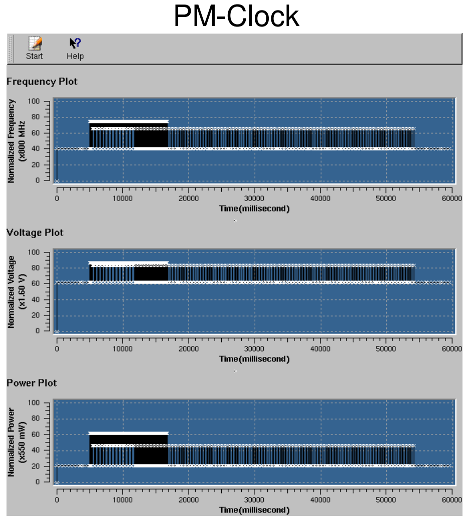
Working with DVS and DFS

PowerMon to monitor Energy & Clock Frequency

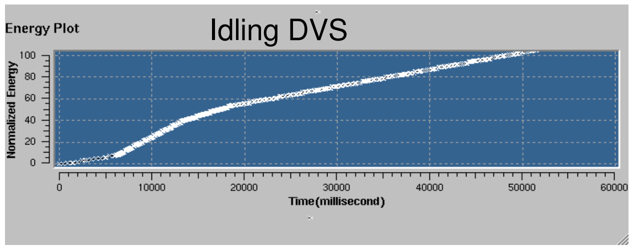
Idling DVS

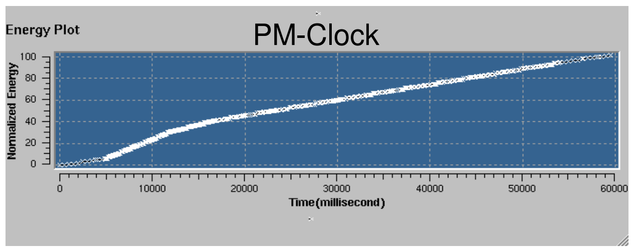


PM-Clock



Energy Monitoring with PowerMon





Conclusions

- Optimizing compilers try to squeeze more performance out
  - Often, there is a tradeoff between optimizing memory space and execution time
- Real-time systems can have very different needs
- A host of performance monitoring tools is available