The Curiosity Rover Landing

http://www.youtube.com/watch?v=a4YqNoLkmxE
Landing a Spacecraft on Mars

- The control software onboard the spacecraft consists of about 3 MLOC. Mostly in C, with a small portion (mostly for surface navigation) in C++.
- The code runs on a radiation hardened CPU. The CPU is a version of an IBM PowerPC 750, called RAD750. It has 4 GB of flash memory, 128 MB of RAM, and runs at 133 MHz.
- ~75% of the code is autogenerated from formalisms, e.g., state-machine descriptions and XML files. The remainder was handwritten, in many cases building on code from earlier Mars missions.

FIGURE A. The amount of flight code that is flown to land spacecraft on Mars has grown exponentially in the last 36 years. Its Compound Annual Growth Rate comes out at roughly 1.20—close to the median value of 1.16 from previous columns.

Midterm Demo

- 3/26, in class.
- 20 min/team (including 3 min for discussion).
- Must show something real!
- Test and set up your demo in advance.
- Email Rahav a summary of your demo and progress
  - Clearly state the contribution of each team member.
Midterm Exam

- Open book, note, papers.
- Bring your calculator!
- Lecture slides override textbook when inconsistent.
Scope

- Introduction
- Power management
- Program optimization
- TinyOS
Embedded Systems

- Non-functional constraints
  - Real-time
  - Power
  - Energy
  - Memory
  - Cost
  - Size

- Designed to tight deadlines by small teams
Alternative Technologies

- Application-Specific Integrated Circuits (ASIC)
- Microprocessor
- Field-Programmable Gate Arrays (FPGA)
ASIC

✓ Performance
✓ **Power**: Fewer logic elements $\rightarrow$ low power

✗ **Development cost**: Very high
  - 2 million $ for starting production of a new ASIC
  - Needs a long time and a large team

✗ **Reprogrammability**: None!
  - Difficult to upgrade systems
  - Single-purpose devices
Microprocessor

- Performance
  - Programmable architecture is fundamentally slow!
    - Fetch, decode instructions
  - Highly optimized architecture and manufacturing process
    - Pipeline; cache; clock frequency; circuit density; multi-core…

- Power
  - Processors perform poorly in terms of performance/watt!
  - Power management can alleviate the power problem.

- Flexibility, development cost and time
  - Let software do the work!
State of the Practice

- Microprocessor is the dominant player
  - Flexibility + low development cost >> low performance/watt
  - Power management is crucial.

- Microprocessor + ASIC is common
  - Ex: cell phone

- FPGA is expected to improve
Power vs. Energy

- Power = energy consumption per unit time

- Power → Heat

- Energy → Battery life
Hardware Support

- Clock gating
- Shut down power supply
- Dynamic Voltage Scaling
Requirements

- Minimize power under performance constraints
  - Real-time applications

- Optimize performance under power constraints
  - Battery lifetime constraint

- Different tradeoff points in design space
Factors in Dynamic Power Management

- **Device:** Power State Machine (PSM)
- **Workload:** distribution of active and idle intervals
SA-1100 Power State Machine

- **P_{ON} = 400 mW**
- **P_{OFF} = 50 mW**
- **P_{OFF} = 0.16 mW**
- **P_{TR} = P_{ON}**

States:
- **run**
- **idle**
- **sleep**

Transitions:
- From **idle** to **run**: 10 μs
- From **run** to **sleep**: 90 μs
- From **sleep** to **idle**: 90 μs
- From **run** to **run**: 160 ms
Analysis

- Inherent exploitability
  - No performance penalty
  - Assume full knowledge of workload in advance

- Actual energy saving and performance penalty under a practical policy
Break-Even Time $T_{BE}$

- Enter an inactive state is beneficial only if the idle time is longer than the break-even time

  - $P_{TR} \leq P_{ON}: T_{BE} = T_{TR} = T_{ON,OFF} + T_{OFF,ON}$
  - $P_{TR} > P_{ON}: \text{Larger } T_{BE} \text{ to compensate for energy cost}$
Inherent Exploitability

Chenyang Lu
Metrics

- **(Performance) Safety**: $\text{Prob}(p|o)$
  - If an observed event happens $\rightarrow$ the probability of $T_{\text{idle}} > T_{\text{BE}}$
  - Overprediction $\rightarrow$ lower safety $\rightarrow$ higher performance penalty

- **(Energy) Efficiency**: $\text{Prob}(o|p)$
  - If $T_{\text{idle}} > T_{\text{BE}}$ $\rightarrow$ the probability of successfully predicting it.
  - Underprediction $\rightarrow$ lower efficiency $\rightarrow$ waste more energy.
Fixed Timeout Policy

- Enter inactive state when the system remains idle for $T_{TO}$.
- Wake up in response to activity.
- Premise: If a system has been idle for $T_{TO} \Rightarrow$ remain idle for $>T_{BE}$.
Impact of Timeout Threshold

Chenyang Lu
Possible Improvement

- **Predictive shutdown**: shut down immediately when the processor becomes idle.
  - Avoid wasting energy before reaching timeout threshold
  - More efficient, less safe

- **Predictive wakeup**: wake up before activity occurs.
  - Reduce performance penalty for wake up
  - Less efficient, safer
Open standard for power management.
ACPI System Power States

Used as *contract* between hardware and OS vendors
Power Consumers

- Instruction execution (CPU)
- Cache (instruction, data)
- Main memory
- Storage
- Display
- Network interface
- I/O devices
Energy Efficiency of Memory Operations

Relative energy per operation: register < cache < memory

- memory transfer: 33
- external I/O: 10
- SRAM write: 9
- SRAM read: 4.4
- multiply: 3.6
- add: 1
Power Optimizations

- Reduce memory footprint
  - Reduce code and data size
  - Analyze footprint to find right size

- Find correct cache size
  - Analyze cache behavior (size of work set)

- Minimize memory and cache access
  - Use registers efficiently → fewer cache access
  - Eliminate cache conflicts → fewer memory access

- Shorter execution time → more idle time
Program Optimization and Analysis

- Performance
- Memory footprint
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        z[i][j] = b[i][j];

zptr = z; bptr = b;
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
    {
        zind = i*M+j;
        bind = i*M+j;
        *(zptr+zind) = *(bptr+bind);
    }

zptr = z; bptr = b;
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
    {
        zbind = i*M+j;
        *(zptr+zbind) = *(bptr+zbind);
    }

zptr = z; bptr = b; zbind = 0;
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
    {
        zbind++;
        *(zptr+zbind) = *(bptr+zbind);
    }
Array Conflicts in Cache

```c
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        a[i][j] = a[i][j] + b[i][j];
```
More

- Function Inlining
  - Cost of function calls
  - Code size vs. performance

- Register allocation
int main() { f1(x); }
void f1(int a) { f2(a); }

; f1 is called by main()
LDR r0, [r13] ; load para. into r0 from stack
STR r14, [r13] ; store f1’s return addr.

; f1 calls f2()
STR r0, [r13, #4]! ; push para. for f2 to stack
BL f2 ; branch and link to f2

; f1 receives return from f2()
SUB r13, #4 ; pop f2’s para. off stack

; f1 returns to main()
LDR r15, [r13] ; restore register and return
Execution Time Analysis

- Execution time is affected by both program path and instruction timing
  - Program path depends on input data values.
  - Instruction timing depends on pipelining, cache behavior…

- Accurate execution time is unknown *a priori*

- Compile-time analysis vs. measurement
Reducing Code Size

- Function inlining?
- Avoid loop unrolling.
- Use processors with dense instruction sets.
- Use compact instruction set.
- Hardware support for code compression.
TinyOS Two-level Scheduling

- **Tasks do intensive computations**
  - Non-preemptive FIFO scheduling
  - Bounded number of pending tasks
- **Events handle interrupts**
  - Interrupts trigger lowest level events
  - Events can signal events, call commands, or post tasks
- **Two priorities**
  - Event/command
  - Tasks
Good luck! 😊