
arm
Performance Evaluation and OS-Aware Debugging
Operating System Lab-in-a-Box

Previous

- Sharing Data in RTX
- Lab

After you code…

- Now you have built your MCU programs based on RTOS, what next?
- Assume it is functionally sound (no functional bugs)
  - Optimize your program
  - Based on its performance
  - Tweak your program/configuration of the OS until optimal or you are satisfied
- Debugging terminology
  - Functional debugging
  - Performance debugging
  - Modular debugging
  - Tracing
  - Profiling
  - Terms are sometimes used interchangeably and mostly overlapping
- Intrusiveness – notice that almost all debugging techniques will affect performance
- Watchdog – commonly used recovery mechanism in embedded systems

Functional Debugging

- Focus on the functional property of your program
- Does the result match your expectation?

- Stepping through your program
- Tracing
- Breakpoints
- Print statements
- LED heartbeat debugging

- Beware of some of the concurrency bugs – hard to identify and recover from:
  - Deadlock
  - Livelock
  - Not releasing mutexes

Performance Evaluation

- Or performance debugging
- What are the performance metrics for your application?
  - Throughput: number of tasks completed in a time unit
  - Latency/response time: delay between request and the response to the request
  - Turnaround time: delay between request and completion of the request
  - CPU utilization
  - Variation: does your program fluctuate a lot?
  - Deadline
  - Priority
  - Memory: may not be obvious for this course as the board has plenty of ROM/RAM space and RTX is relatively lightweight
  - Power: OS will induce power consumption overhead
- Really varies and affected by architecture, compilers, boards and applications.
- Trade-off: good design demands good compromises
- Tune your OS configuration

Performance Evaluation

- We will mainly focus on the timing behaviour of our system
- Worst case execution time (WCET) analysis
- Simulation
  - Difficult but sometimes necessary
- Static timing analysis
  - Examine the source code
  - State space exploration and modelling
- Measure directly
  - Experimental and accurate
  - Can be time consuming

*Wilhelm, Reinhard, et al. "The worst-case execution-time problem—overview of methods and survey of tools." *ACM Transactions on Embedded Computing Systems* (2008).

Prediction of Execution Time

- What aspects of execution time do we care about?
  - Average – what's the typical performance?
  - Worst-case – must meet deadlines
  - Best-case – too fast might cause race conditions or other problems
  - Distribution and variability

- *"Prediction is extremely difficult. Especially about the future."*
  – Niels Bohr

- Prediction Techniques
  - Manual estimate based on databook – shortcomings are obvious
  - Use a processor simulator
  - Measure a real system

Finding the WCET

- Need WCET for all tasks and ISRs to perform scheduling analysis

- Sources of execution time variations
  - Software
    - Different input data may trigger different control flow behaviour
  - Hardware
    - Some instruction execution times may depend on input data
    - Pipelines may stall
    - Caches may miss
    - Branch target buffers may miss
    - DMA activity may slow or delay task execution

Examining Object Code

- C and other high-level languages hide implementation details and allow programming at a higher level

- This abstraction means that it isn't always obvious what assembly code will be generated for a C statement, and (for example) how long it will take to execute
  - *a = b*c[i+j]* depends on:
    - data types of a, b, c, i, j: int? float? double? long?
    - Instruction Set Architecture (ISA) and MCU implementation
      - Multiply instruction for the data type
      - Advanced addressing modes suitable for array indexing

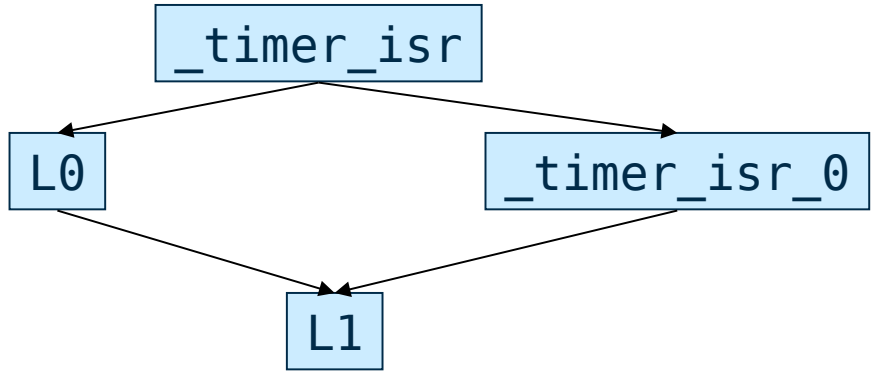
- Examining the object code generated by the compiler is the only way to get an accurate picture of what will happen
  - It gets ugly very quickly ("does not scale well").

- We need representations which reflect the program's structure in order to simplify the analysis

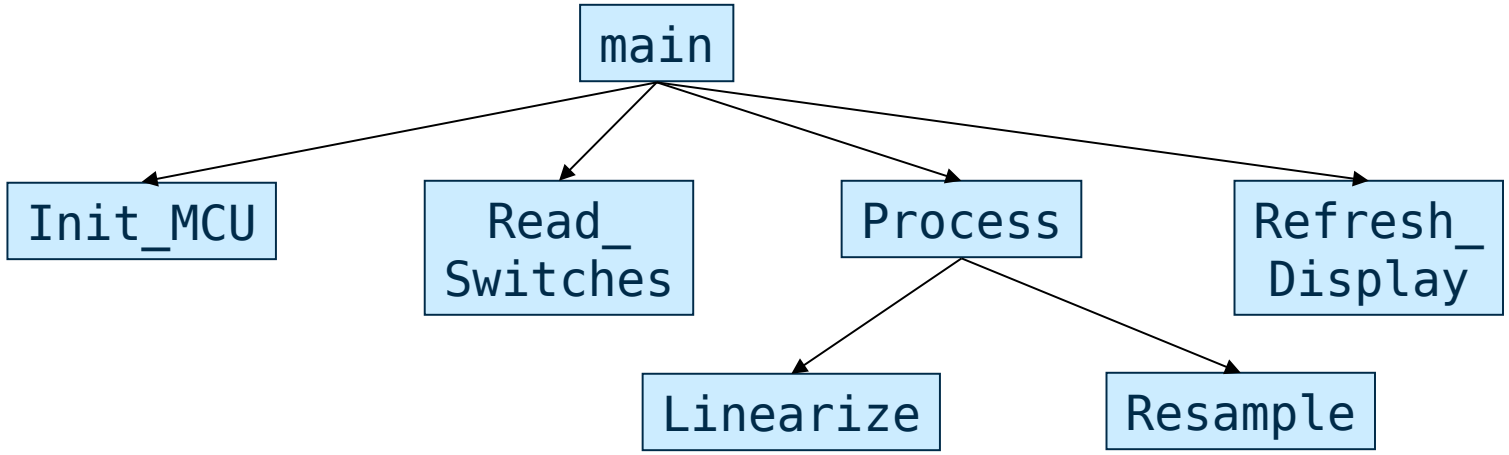
- Also, compilers and other tools use these representations to analyze and optimize the code automatically

Control Flow Graphs and Call Graphs

- Control flow graph (CFG)
  - A flow chart which shows the execution sequence of the program
  - Each node is a basic block (sequence of instructions, potentially with conditional jump at end)
  - Create one CFG per **subroutine** or **interrupt service routine**



- Call graph
  - A hierarchical (tree) form which shows the nesting of subroutine calls
  - Each node is a subroutine
    - Going down an arrow indicates calling a subroutine
    - Going up an arrow (backwards) indicates returning from that subroutine
  - Create one call graph per **program**



arm

CFG Formation Rules

- A CFG consists of **basic blocks (BB)** joined by edges
- Basic block: a sequence of consecutive instructions such that each instruction is executed **exactly once** if the basic block is executed.
- This implies
  - the flow of control begins at the entry and leaves at the exit
  - there is no conditional branching except potentially at the end.
    - no instructions can be skipped within a basic block
  - conditional branch (+skip) instruction effectively ends the basic block
  - a jump/branch into a BB will split it into two
  - a subroutine call ends the basic block
- Relationships with other BBs
  - Predecessors: all basic blocks which can execute immediately before the given basic block
  - Successors: all basic blocks which can execute immediately after the given basic block.
- For our purposes, a new label starts a new basic block (except in the case of consecutive labels, in which case the basic block is assigned the first label)

Call Graph Details

- Each subroutine is represented by a node
- Each potential call from subroutine A to B is represented by a directed edge from A to B
- Each ISR has a node, but it is not called by any other code (except if software interrupts are supported)
- Operations not supported by ISA (e.g. modulo (%)) may be implemented with subroutine linked in from a C library, leading to a deeper call graph than expected

Static Timing Analysis Procedure

- Compile source code
- Examine assembly code
- Form basic blocks
- Form control flow graph from basic blocks
- Determine duration per basic block by adding instruction durations
- Evaluate paths through function
  - Best and worst-case times for function
  - Deal with control-flow complexity
    - For code in conditional region (if-then-else),
      - If control-flow path is known, calculate exact time for path
      - If control-flow path is unknown, *bound* the time: choose the larger time for WCET, the smaller for BCET
    - For code in loop, use the exact number of iterations (if known) or else try to derive a bound (minimum and maximum)

How Long Does An Instruction Take?

- As a guideline for Cortex-M cores:
- Most instructions
  - 1 cycle
- Loads and stores
  - Usually 2 cycles: to AHB interface or SCS
  - 1+N cycles: load multiple, store multiple, push, pop (N registers)
- Any instruction writing to PC
  - 2 Cycles
- Conditional branch
  - Not taken: 1 cycle
  - Taken: 2 cycles
- Other branches
  - Unconditional, exchange, link & exchange: 2 cycles
  - Link: 3 cycles

- See the Technical Reference Manual of your core for more details

Measure a Real System

- If simple enough, can count instruction cycles. But not scalable as nowadays program flow and program inputs can easily spin out of control.
- In most cases, directly measure the timing behaviours of your system is more intuitive and direct

- Use external analysis tools
  - Logic analyzer looks for start and end addresses of routine in question
  - Oscilloscope looks for special events (e.g. on debug pins)
    - Set up GPIO port to set output bit upon entering routine, clear it upon exiting

- Use a high-resolution timer in the circuit
  - Configure as cycle counter
    - Most MCUs have counter peripherals
    - Can select prescaled clock source if needed to increase time range
    - Make it increment a counter variable when it overflows

Example – Measuring Execution Time

```
void clear_ticks(void) {
  ta0s = 0; // stop counting
  overflow_count = 0;
  ta0 = 0xffff;
  ta0s = 1; // restart counting
}

void main (void) {
  unsigned long t;
  mcu_init();
  init();
  clear_ticks();
  // do processing to be timed
  t = get_ticks();
  while (1);
}
```

Timing Data Analysis

- Statistics are informative
  - Minimum, maximum
  - Mean: sum total time and divide by number of measurements

```
#define MIN(a,b)    (((a)<(b))? (a):(b))
#define MAX(a,b)    (((a)>(b))? (a):(b))
#define NUM_TESTS   (300)
  unsigned long t=0, min, max;
  float sum=0.0;
  min = 0xffffffffl;
  max = 0;
  for (i=0; i<NUM_TESTS; i++) {
    Clear_Ticks();
    f = tan(i/100.0);
    t = Get_Ticks();
    min = MIN(t, min);
    max = MAX(t, max);
    sum += t;
  }
```

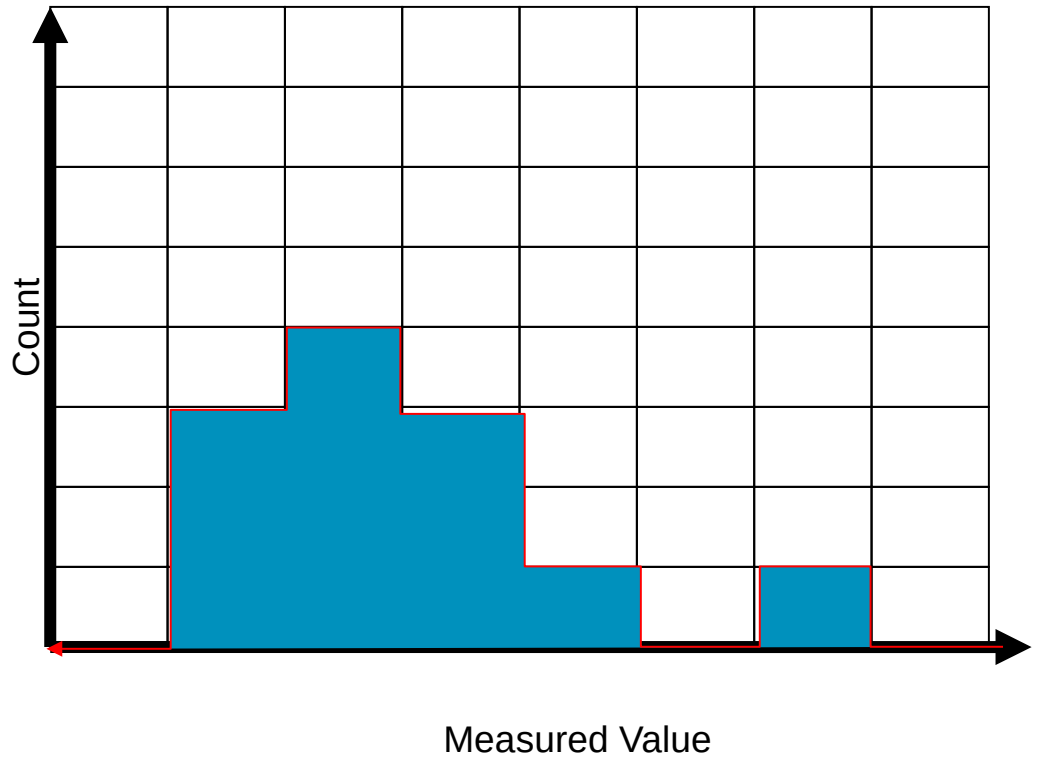
Repeatability

- Does the same input lead to the same output, or are other factors in the system affecting the computation?

```
min = 0xffffffffl;
max = 0;
for (i=0; i<3000; i++) {
    Clear_Ticks();
    f = sqrt(37/100.0);
    t = Get_Ticks();
    min = MIN(t, min);
    max = MAX(t, max);
}
```

Histogram Showing Distribution of Execution Times

```
#define HIST_SIZE 10
int hist[HIST_SIZE];
for (i=0; i<HIST_SIZE; i++)
    hist[i] = 0;
for (i=0; i<300; i++) {
    Clear_Ticks();
    f = sqrt(i/100.0);
    t = Get_Ticks();
    min = MIN(t, min);
    max = MAX(t, max);
    n = (unsigned) (t/250);
    hist[min(n, HIST_SIZE-1)]++;
}
```



- Horizontal axis: range of values of measured variable (bins)
- Vertical axis: number of times (frequency) variable had that value

Evaluating Responsiveness

- Three important types of critical path
  - From interrupt request to ISR running
  - From interrupt request to user task (signalled by ISR) running
  - From one user task to another user task (signalled by first task)

- First two most critical for RT embedded systems

- How to measure?
  - Use embedded instruction trace capability – many Cortex M MCUs have ETM or MTB
  - Instrument program to send out trace information (e.g. debug signals) to view with oscilloscope or logic analyzer

OS Aware Debugging

- Performance evaluation for OS can be slightly different:
  - “Useful” tasks do not really terminate
  - Either sleep for a while or wait for some events
- Preemption and interrupts
- How can you detect deadlock
- You actually need to know more than just the timing
  - Task states
  - Priorities
  - Visit times (profiler’s job)
  - OS overhead
- The lab session will introduce some OS aware debugging features of Arm MDK

Optimizations

- Minimize interrupt lock-out time
  - Application
  - OS
- If response time is more important than CPU utilization, then adjust task priorities accordingly
- Which part of the code uses the most time?
  - Profiler – sampling
  - That’s the best place to start optimization for speed

Next

- Lab