System Performance Analysis

Introduction

- Performance
  - Means many things to many people
  - Important in any design
  - Critical in real time systems
    - 1 ns can mean the difference between system
      - Doing job expected
      - Failure
- Performance Measures
  - Simply put performance means meeting the (time) specification
    - To meet specification one must have specification
  - Task becomes one of
    - Identifying level at which performance is to be measured
    - Identifying meaningful parameters at that level
    - Selecting reasonable and proper values
    - As designers this is our job
  - Measures
    - *Response time*
      - Interval between occurrence of event and completion of associated action
        - Event can be
          - Internal
          - External
            - Timer
            - State change

      - Also referred to as
        - *Execution time*
        - *Throughput*
        - These really mean something different
    - *Time Loading*
      - Percentage of time CPU doing useful work
    - *Memory Loading*
      - Percentage of usable memory being used

  - In examining performance
    - Interested in several things
      - Exact times if computable
      - Bounded times if exact not computable
        - Can be measured

Deterministic

Let's now look at each of these

Response Time

Interval between event and completion of associated action

Command to A/D to make reading

Event from A/D signifying completion

Driven by type of system involved

Let's examine different control flow segments

Examine response time of each

Polled Loops

These are the simplest and best understood

Response time comprised of 3 components

1. Hardware delays in external device to set signaling event
2. Time to test the flag
3. Time needed to respond to and process event associated with flag

External Hardware Device

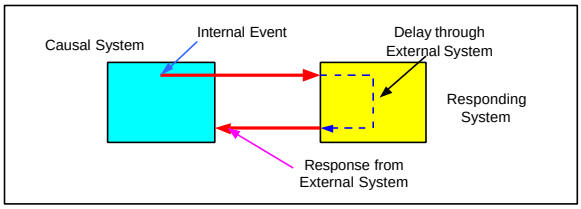
Must consider 2 cases

1. Response through external system to internal event prior

2. Asynchronous external event

Case 1

Let's look at a graphical depiction of problem



In analyzing the behaviour must consider

Time to get to polling loop

From internal causal event

Delay through the external device

Time to generate response

Can be complicated to analyze

Particularly if triggering event

Takes several different paths through external device

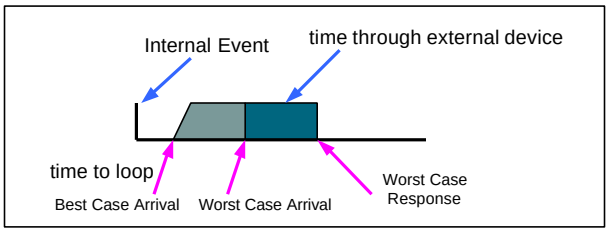
May not be possible to calculate

Alternately

Place upper bound

Set minimum limit on hard real time behaviour

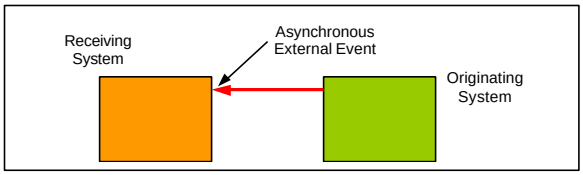
In time picture looks like the following



Call this time τ_{ed1}

Case 2

Our problem now appears as



In such case

Can't determine when event will occur

OK

Flag Time

Determined from execution time of machine's bit test instruction

Call this time τ_f

Processing Time

Time to perform task associated with triggering event

Triggering event may be

Internal

External

This time must include
Time to
Reach flag from current instruction
We're in the polling loop
Not at test instruction
Must consider
Best, worst, and average times
Reset flag
Execute task
Call this time τ_p
Can consider anticipating event
If must meet very high speed requirement
With slow causative process or device
Can arm system
So that when event occurs
Everything set up to go
Could use hardware as implementing mechanism
Consider small window to respond to event

In computing time
Must consider 2 cases
First event
n^{th} event
First event
Loop unburdened
Becomes min time to execute
n^{th} event
Loop may permit events to queue
In this case
Must add time to complete some subset of
n-1 previous events
Can be bounded by
n-1 times time to process single event

Coroutine

In non-interrupt environment
May be computed directly
More often bounded
Computed as worst case path

Through each component

Interrupt Driven Environment

- Most complex of calculations
- Asynchronous events
- Can be non-deterministic
  - Events probably will not occur
    - Same sequence each time program executed
- We set a bound on the complexity
  - Assume only one such event
- Affecting factors
  - Interrupt latency
  - Context switch
    - To interrupt handler
    - To acknowledge interrupt
    - To processing routine
    - To original context
  - Schedule
    - Non-preemptive
    - Preemptive
  - Task execution
- *Preemptive*
  - Preemptive with fixed rate scheduling easiest to compute
  - Let's look at the pieces
    - **Context Switch**
      - Can be computed by directly
        - Identify instructions
        - Count time each takes
    - **Task Execution**
      - Usually three values considered
        - Min average max
      - Computed by
        - Counting instructions
        - Measuring length

    - **Interrupt Latency**
      - Must consider two cases
        1. Highest priority device
        2. Lower priority device

Case 1:
- Three factors
  - Time from leading edge in external device until internal recognition
  - Complete current instruction if interrupts enabled
    - Most processors complete current instruction
      - Before switching context
      - Bounded by longest instruction
  - Complete current task if interrupts disabled
    - Bounded by task size
  - Time bounded by longer of latter two factors

Case 2:
- Lower priority device
- Must consider two cases
  - Interrupt occurs and processed
    - Computed as above
  - Interrupt occurs and is interrupted
    - Unless interrupts disabled
    - Non-deterministic situation
    - In critical cases may have to
      - Change priority
      - Place limits on number of preemptions

Non-Preemptive
- Since preemption not allowed
- Computed as in highest priority case above

Time Loading
- Percentage of time CPU doing useful work
- By useful work we mean
  - Execution of those tasks for which program designed
- Entails understanding executing times
  - Constituent modules
- Compute by
  - Finding time spent in
    - Primary tasks

- Support tasks
- Compute ratio of
  - Primary / (Primary + Secondary)

Three primary methods
- Instruction counting
- Simulator
- Measurement instruments

Instruction Counting

- Requires that code actually be written
- At the end of the day
  - Best method to determine time loading
    - Due to code execution time

- Begins with identifying instructions involved in routine of interest
- From processor vendor's manuals
  - Determine time for each instruction
  - Can vary with
    - Addressing mode of instruction
    - From which piece of memory
      - Instruction or data must be fetched
        - Immediate
        - Register
        - Primary
        - Secondary
  - May wish to use
    - Min max average
- Determine path through code
  - May wish to use
    - Min max average
  - Select time based upon objective
  - For critical analysis
    - Pick longest path

- For *periodic* system
  - Total task execution time divided by time for individual module
    - Time loading for that task
- For *sporadic* systems
  - We use maximum task execution rates

Combined percentages over all tasks

Yields total time loading

If

Total time loading is T

T_i is cycle time for i^{th} task

A_i is execution time for i^{th} task

For n tasks

$$T = \sum_{i=1}^{n} \frac{A_i}{T_i}$$

Simulation

Instruction counting limited utility

As noted above many other factors in computing times

Memory accesses

Addressing schemes

Loop iterations

Simulation begins with completely understanding system

Then developing accurate model

Model can include

Hardware

Software

Both

We can use tools like VHDL or Verilog

Model hardware

Variety of software modeling tools available as well

Modeling can be done at variety of levels

Based upon kind of information being sought

Models

Two major categories of models

Behavioural or Conceptual

Usually based upon symbols to represent qualitative aspects

Structural or Analytic

Use mathematical or logical relations to represent physical behavior

Models of physical parts

We develop and apply models
- Variety of levels and reasons

Most common given as

System Level Model
- Described by a hierarchical and structural model
  - Representing a set of communicating functions or processes.
    - Functions are specified by a behavioral model

Functional Model
- Collection of functions
  - Hierarchical and graphical,
- Describes a system by
  - Set of interacting functional elements
  - Behaviour of each element
    - Software

Physical Model
- Describes architectural structure
  - Based upon real components and their interactions.

Structural Model
- Describes organization of system
  - Based upon components in system and interconnections among them
- Includes functional and physical level elements
  - Mapping between them
    - Binds the functional to physical
- Can be used at any level of abstraction

Behavioral Model
- Wide variety of models in this category
  - Behavior frequently expressed as function of time

Data Models
- Entity - Relation models
  - Represent world in terms of
    - Entities
      - Their Attributes
      - Relations between / among them

Instrumentation

- There are numerous instruments
  - Logic analyzers
  - Code analyzers
- Permit system
  - To be instrumented
  - Performance measured qualitatively
- Such instruments
  - Can measure max and min times
  - Time loops
  - Identify non-executed code
  - Rates of execution
    - Most frequently used code
    - Permits later optimization
- Major caveat
  - Any such measurements
    - Only as good as input to system
      - If not executing both
        - Typical
        - Extreme applications
      - Quality of measurement suspect
  - Further
    - Measurements are not predictive
      - Sense that one cannot say
        - Numbers guarantee performance of system
        - Under all circumstances
- Use any such instruments with caution
  - Can provide useful information
  - Keep in perspective

Performance Optimization

- Let’s now look at a few ways we can begin to
  - Improve system performance
- We’ll assume by performance
  - Refer to response time and time loading
    - Response time
      - Time from submittal to completion
    - Time loading
      - Percent of time CPU or memory
        - Doing work for us

Considerations

When optimizing important to think about

- What is being optimized
- Why is it being optimized
- What will be affect on overall program
  If software or hardware being optimized
  Eliminated from program
  Zero execution time
  What will affect be
- Is optimization appropriate to operating context
  Don't floating point performance
  If only working with ints

Common Mistakes

1. Expect improvement in one aspect to
   Improve performance proportional to improvement
   See comment above
   A 100% improvement in aspect affect 1% of performance
   Minimal

2. Hardware independent metrics predict performance
   Example code size
   Remember the difference between macros and subroutines

3. Comparing performance based upon couple of metrics
   Example clock rate, clock cycles per instruction, instruction count
   Higher clock rate or more instructions
   Does not guarantee better performance

4. Using peak performance as measure

5. Using synthetic benchmarks
   Code can be optimized to excel on benchmark
   Not encountered in real world

Reducing Response Times and Time-Loading

1. Perform measurements and computations
   At rate and significance consistent with
   Rate of change of data

- Values of data
  - Type of arithmetic
  - Number of significant digits calculated
- Often interacting with external world
  - Temperature typically very slowly changing entity
    - Measuring change at sampling interval greater than 1 second
      - Wasting CPU cycles

2. Use lookup tables or Combinational logic
   - Look up is faster than computing
   - Make a measurement, scale, convert
   - Arithmetic and shifting operations
     - Can be logical rather than arithmetic
   - Scaling a value by a constant
     - Logical operation
   - Multiplying two int
     - Combinational logic problem

Learn from the compiler guys
- Many tricks commonly used by compiler writers
  - Reduce code size
  - Improve speed performance

Be careful also
- Optimizing can cause problems
- Example
  - Put value in register
  - Assume several instructions value will be
    - There and unchanged
    - No need to reload
  - Value may have been modified by some other routine
    - Shared variables
    - C++ has
      - Volatile and const volatile qualifiers

3. Loop invariant optimization
   - Precalculate any values that will not change within
     - Block of repeated code
   - Some good compilers do this for you
   - Use precomputed value rather than recomputing each time
   - Can be particularly significant if

Operand require indirect memory access for example
Working with several arrays
Indices differ by integer value

4. Flow of control optimization
In branches or switches
Avoid repeated
Jumps or tests

```
    je $2
$1:   ...
$2: jmp $3 ...

    je $3
$1:   ...
$2: jmp $3 ...
```

C code fragment

```
switch (y)
{
    case 0: x = x+1;
    case 1: x = x+2;
    case 2: x = x+1;
}

switch (y)
{
    case 0:
    case 2: x = x+1;
    case 1: x = x+2;
}
```

May also be able to set x to value before switch
Change if necessary

```
while(1)        //  recall the if is a single expression
    if (light ==ON)
        light = OFF;
    else
        light = ON;
```

```
while(1)
    light = ~light;
```

5. Use registers and caches
    - Languages like C and C++
        - Support register type variables
    - Usually advantageous to utilize such types
    - Register operations faster than memory operations
    - When working with C or C++
        - Register qualification on variable declaration
            - Requests compiler put variable into register
        - No guarantee compiler will comply
    - Can force by writing code in assembler

    - Some processors support caching
        - Use caching to store frequently used variables
            - More rapid access than general purpose memory

6. Unroll loops
    - Consider the following simple code fragment

```
for(j = 0; j < 4; j++)
    a[j] = a[j]*8;
```

    - We can unroll this several ways
    - Case 1

```
a[0] = a[0]*8;
a[1] = a[1]*8;
a[2] = a[2]*8;
a[3] = a[3]*8;
```

Case 2

```
for(j = 0; j < 2; j++)
{
    a[j] = a[j]*8;
    a[j+1] = a[j+1]*8;
}
```

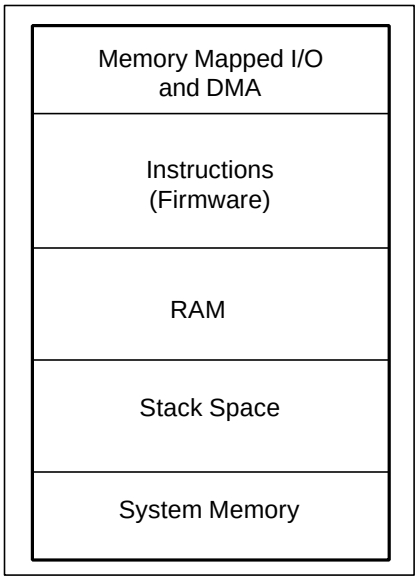
7. Use only necessary values
    - The X Windows mouse
    - Dragging a graphic wire frame
    - Slewing
8. Optimize common path or frequently used code block
    - Most frequently used path or highly used code segment
        - Should be most highly optimized

Memory Loading

- Today in may applications
    - Memory almost free
- Many cases when it's not
    - Weight
        - Aircraft
        - Spacecraft
    - Cost
        - Very high volume consumer products
            - Automotive
            - Televisions
    - Power consumption
        - Portable applications
- In such cases
    - Must optimize use of memory
        - Here use refers to amount
- As we saw earlier
    - Memory comprised of several different colors
        - Code space
        - Data space
        - System space
- Recall once again we optimize to focus most of resources
    - On getting target task completed

We defined memory loading as

Percentage of usable memory being used for task



Memory Map

Developing memory map

Useful for understanding allocation and use

We see an example in the accompanying figure

Lists addresses in memory allocated to each portion

Note this is primary physical memory

At top level

Comprised of two basic types

RAM and ROM

ROM used to hold words

Not expected to change at runtime

RAM used to hold words

May change at runtime

We build variety of data structures in RAM

If memory mapped I/O being used

All of physical memory will not be available

Holding data or code

Note it is possible for required code and data space

To exceed total available primary memory

Techniques called

Virtual memory and Overlays

Support such situation

When dealing with such situations

We must spend time bring these into primary memory

Total memory loading will be sum of individual loading

Instructions

Stack

Ram

Loading given by

$M_T = M_I*P_I + M_R*P_R + M_S*P_S$

The values M_i reflect memory loading

For each portion of memory

The values P_i represent the percentage of total memory

Allocated for program

M_T will be expressed as a percent

Example

Let system be implemented as follows

M_I – 15 megabytes

M_R – 100 kilobytes

M_S – 150 kilobytes

P_I – 55%

P_R – 33%

P_S – 10%

We get a value for M_T of

$$M_T = 0.55 \bullet \frac{15}{15.25} + 0.33 \bullet \frac{0.1}{15.25} + 0.1 \bullet \frac{0.15}{15.25}$$

M_T = 54%

Observe

We do not include memory mapped I/O / DMA space in calculation

Fixed by hardware design

Considerations

Allocate minimum amount necessary

Be sure to allow room for future growth

Leave remaining amount of RAM

For application program(s)

Instruction / Firmware Area

This portion of memory space

Generally ROM of one form or another

Reason called firmware

Contains instructions to implement application

Memory loading computed by

Dividing number of user locations

By maximum allowable

We get

$$M_I = \frac{U_I}{T_I}$$

RAM Area

This portion of memory space
- Generally for storing
  - Program data of one form or another
  - Global variables
- Occasionally used for storing
  - Instructions
  - Done to improve fetch speed
  - Support modifiable instructions
    - Generally we avoid such thing
  - Reason called firmware

Size determined at design time

Can only determine loading
- After design completed

Memory loading computed by
- Dividing number of user locations
  - By maximum allowable RAM area

We get

$$M_R = \frac{U_R}{T_R}$$

Stack Area

This portion of memory space
- Used to store
  - Context information
  - Auto variables

Depending upon design
- Can have multiple stacks
  - In this area of memory

From point of view of memory loading calculations
- Model as one single stack

Capacity generally determined at design time
- Size based upon use at runtime

Can establish a bound
- We assume a maximum number of tasks
  - Call that number t_{max}
- We next assume maximum allocation for each task
  - Call allocation s_{max}

From these we compute maximum stack size

$U_S = s_{max} * t_{max}$

Memory loading computed by

Dividing U_S

By maximum allocated stack area

We get

$$M_S = \frac{U_S}{T_S}$$

Memory allocation

Occam's razor applies

Never allocate more space that necessary

Most operating systems give control over stack size

Define size that is large enough without being too much

Be certain to understand problem

Stack overflow can be dangerous

If developing multithreaded to multitasking applications

Without purchased kernel

Carefully manage how stack allocated and deallocated

Trade-offs

Often times performance is optimization issue

Involves trading several contradictory requirements

Speed

Memory size

Cost

Weight

Power

Time must be spent up front to thoroughly understand

Application

Constraints