System Performance Analysis and Optimization

Introduction
Is performance important?
Yes! Faster, smaller, cheaper, lower power, is better
Assuming we meet all the specified constraints
Very important in embedded systems
Often we have to trade-off speed and memory size
To be able to do so, we need concrete numbers
Such techniques become more important with larger systems
We use a different approach when dealing with small systems
Wait – what is performance?
As we progress we’ll see

Must keep things in perspective

If we let our first thoughts focus on software
Remember for any algorithm or method
Has basic housekeeping
Constituent steps
Choice of language or processor can have ±100% affect
Therefore at first cut not really significant
These are details
Can usually wait for faster machine and problem goes away
Choice of fundamental algorithm
Can make vital difference

Other considerations
• Development time
• Ease of maintenance
• Extensibility

Approaches to improving software performance
• Tighter code
• 'Better' algorithms
• 'Better' data structures

There’s much more

An embedded system
Comprises collection of hardware and software components
Such a set intended to provide desired
   Behaviour
   Service

(Embedded) system design
Process of implementing desired behaviour or services
With such an implementation we want to
   Optimize certain aspects of behaviour or services
   Subject to set of constraints

Optimization and performance of a system
Mean many things to many people
While certainly can say inherently good
   What do they mean?
   What are we optimizing?
   What kind of performance are we examining?
   What is performance?

First steps at understanding
With any design
   Many different measures of performance
   Many different ways of analyzing performance
   Many aspects which we can optimize

Performance or Efficiency Measures
Overview
Performance or efficiency
   Usually means "time" (to run) or "space" (memory used)
   Simply put performance means meeting the 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
How to measure efficiency
- Run the program
- See how long it takes
- See how much memory it uses
- Tools are available

Lots of variability when running the program
- What input data?
- What hardware platform?
- What compiler?
- What compiler options?

Just because one program is faster than another right now
Will it always be faster?

Let’s begin to quantify
Several major areas can begin to focus on
- Complexity
- Time
- Power consumption
- Memory management and memory size
- Cost
- Weight
We will examine several of these

For each such measure
Must consider
- Best or min case
  When referring to time important in many scheduling algorithms
  When referring to cost
    Becomes value below which cannot remove any more parts
  Similar argument for power
- Average case
  Gives typical measure
  Often sufficient
- Worst case
  Largest or longest value of a particular measure
  When referring to time
    Sets an upper bound on a schedule
Typical embedded system
Comprises root hardware platform
May include number of peripheral devices as well
Goal of hardware is to implement specified behaviour or service(s)

From embedded point of view
✓ We consider hardware to comprise
   Computational elements
   Communication subsystem
   Memory
   Power comes in as adjunct to all

✓ We consider software (firmware) to be
   Algorithms and Data Structures
   Control and Scheduling

To optimize performance of combined system
Must consider performance of each hardware and software constituent
At the end of the day
Simply put performance means meeting the specification
To meet specification one must have specification
When we optimize one aspect of the system or another
Again we are optimizing against a specification

Based upon our 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

Limitations
We want to look at each of these aspects of performance in turn
Certainly our goal is to ‘improve’ performance

Before we attack the problem of performance improvement
Let’s step back and quantify limits of what can be done
From Amdahl’s Law we write

\[
\frac{T_{\text{total}}}{T_{\text{improved}}} = \frac{T_{\text{total}}}{(T_{\text{total}} - T_{\text{component}})} + \frac{T_{\text{component}}}{n}
\]

- \(T_{\text{total}}\) - System metric prior to improvement
- \(T_{\text{improved}}\) - System metric after improvement
- \(T_{\text{component}}\) - Contribution of the component to be improved to the system metric
- \(n\) - the amount of the improvement

**Example**

Consider a system with the following characteristics
- Task to improve executes in 100 time units
- Desire 80 time units
- Algorithm to be improved used 40 time units

\[
\frac{100}{80} = \frac{100}{(100 - 40) + \frac{40}{n}}
\]

Simplifying gives a value of 2 for \(n\)
- The algorithm speed will have to be improved to 20 time units

**Example**

Consider a system with the following characteristics
- Task to improve executes in 100 time units
- Desire 50 time units
- Algorithm to be improved used 40 time units

\[
\frac{100}{50} = \frac{100}{(100 - 40) + \frac{40}{n}}
\]

Simplifying gives a value of -4 for \(n\)
- The algorithm speed will have to run in negative time to meet the new spec
- Clearly impossible

With such restrictions in mind
- Let's now look at the various performance measures
- We'll begin on the software side and at a high level
- Let's look at assessing the complexity of our
  - Algorithms
  - Programs
Performance Analysis

Time

Time is one of more critical parameters in embedded systems
Consequently analyzing and optimizing time performance
Central component in optimizing overall system performance

Many embedded applications must perform in real time
In real time systems
1 ns can mean the difference between system
Doing job expected
Failure

What does time mean?
How do we begin to measure time?
What times do we measure?

To answer such questions we’ll begin by
Introducing and defining some of the terminology
Then taking a high-level view of
✔ System
✔ Jobs we must do
Finally move to the details

Time Based and Reactive Systems

In world of embedded systems
Can class systems into two broad categories
Time based
Reactive

Time Based Systems
Systems whose behaviour controlled by time
Can be
Absolute
Relative
Following an interval
Absolute time
Real world time
Interval
Distinct from duration
Interval marked by
Specific start and end times
Equal intervals have same start and stop

Duration
Relative time measure
Non-equal intervals
Can have same duration

Reactive Systems
Comprise tasks
Initiated by some event
Internal or external to system

Internal event
May be elapsed time
Bound on data exceeded

External event
Recognition of keystroke or switch activated
External response to internally generated command

Representing Time
When considering either time-based or reactive systems
Time is important element

✓ Time based
When does something occur
How tightly can time intervals be held or met

✓ Reactive
How quickly can event be recognized
How quickly and repeatedly can event be responded to

Issues of time important
When trying to schedule tasks and threads
Which means meeting the schedule

Tasks or threads that are initiated
With repeating duration between invocations
Called periodic
Such duration called period
Time to complete called execution time
Variation in evoking event called jitter
Must examine each context to determine
Significance of jitter with respect to time constraints

Let's see how we can express this.

Consider basic heart pace maker.

Figure illustrates Edmark wave of heartbeat.

Normal operation:
- Ventricular sense
  - Hearting filling with oxygenated blood
- Pump
  - Heart muscle contracts to pump blood
- Refractory period
  - Muscle fiber relaxes
  - Until can fill and contract

We can express changes in state in our systems in a variety of different ways:
- Timing diagram
- State chart
- Activity diagram

Simplest method probably a timing diagram.

Timing diagram in this context:
- Different from what may have encountered.
- Here we express behaviour of system moving between states.
- In basic diagram we express:
  - States along vertical axis
  - Time along horizontal axis

We elaborate by annotating:
- Durations
- Events
- Jitter
- State transitions
- Here we illustrate a periodic system.

The sloped lines indicate transition between states. Such transition potentially may be significant.
Most of time small compared to other times
Observe how leading and trailing jitter represented

Can show same thing for aperiodic sequence

Note we specify min and max times

Invocation of aperiodic tasks varies
Duration between such tasks called inter-arrival time
Such time is critical when determining how to schedule
Real time tasks

Under such circumstances
Must identify lower bound on inter-arrival time
May also need to consider such things as
Maximum number of events within given time interval

Thinking Schedule – A High-Level View in Time
As we now move to analyzing system temporal behaviour
Begin with high-level view

That high-level view means getting the job done
Getting the job done means meeting
Schedule
Time constraints

We begin with the schedule

When working with scheduling system
Must address priority of task
Priority based upon different criteria

Used to resolve which task to execute
When more than one task
Waiting and ready to execute

Tasks with higher priority
Execute preferentially over those with lower priority
Real time system one in which correctness implies timeliness
Most such systems carefully manage resources
To ensure maintaining predictability
Of the timeliness constraints
Predictability gives measure of accuracy
With which one can state in advance
When and how an action will occur
Thus schedule a real-time system

Task which must start or finish by specified time
Defined as hard or said to have a hard deadline
Missed deadline considered to be partial or total failure
✓ System is defined as hard real-time
  If contains one or more such tasks
  Such system may have other not or non hard real-time tasks
  Major focus however on hard deadlines

✓ System with relaxed constraints defined as soft real-time
  Such systems may meet deadline on average
  Soft real-time systems may be soft in several ways
  • Relaxation of constraint that missing deadline constitutes system failure
    Such system may tolerate missing specific deadline
    Provided some other deadline or timeliness constraint met
    Average throughput for example
  • May evaluate correctness of timeliness as
    Gradation of values rather that pass or fail
    How bad did we miss deadline

✓ System with tasks having some constraints (but relaxed) as well as hard deadline
  Defined as firm real-time

Task that can be determined to always meet timeliness constraint
  Said to be schedulable
Task that can be guaranteed to always meet all deadlines
  Said to be deterministically schedulable
  Occurs when event’s worst case response time
  Less than or equal to task’s deadline
When all tasks can be scheduled
Overall system can be scheduled
Does it matter
Following table captures timeliness constraints
With respect to whether task is soft or hard real-time

<table>
<thead>
<tr>
<th>Property</th>
<th>Non Real-time</th>
<th>Soft Real-time</th>
<th>Hard Real-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>No</td>
<td>Possibly</td>
<td>Yes</td>
</tr>
<tr>
<td>Predictable</td>
<td>No</td>
<td>Possibly</td>
<td>Yes</td>
</tr>
<tr>
<td>Consequences of late computation</td>
<td>No effect</td>
<td>Degraded</td>
<td>Failure</td>
</tr>
<tr>
<td>Critical reliability</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Response dictated by external Events</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Timing analysis possible</td>
<td>No</td>
<td>Analytic</td>
<td>Analytic, stochastic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sometimes), stochastic</td>
<td>simulation</td>
</tr>
</tbody>
</table>

A Look at Scheduling

Introduction
Distinguishing characteristic of real time systems
Have specified time constraints that must be met
Hard or soft constraints
To be able to meet hard real time deadlines
Must be able to determine prior to deployment
Whether or not deadlines can be met
Certainly one approach
Consider and schedule according to worst-case behaviour
While such an approach
Ensures that constraints can be met
Yields relatively poor performance

Have seen several general classes of schedule
- Static
  Schedule determined and set before deployment
  Typically implemented as units of
  Least common multiple of task durations
  Major problem once again inefficiency
  Typically does not need an OS
  Can easily implement with simple task queue
  Most effective when number of tasks fixed and
Subject to very little modification
To be effective
Knowledge of worst-case times
Essential

- Priority
  Tasks assigned a priority
  Although not required
  Such systems generally use RTOS
  Assumed that RTOS supports pre-emption
  RTOS ensures
  Highest priority task running at any time
Has potential of being more efficient than static scheme

Difficulty
Worst-case response times not immediately obvious
  Individual tasks
  Interrupts and associated handler
Without such knowledge
  Becomes very difficult to try to use priority scheme
Can develop such understanding
  Using approach called
  \textit{Deadline Monotonic Analysis}
  Scheme provides means to
  Analyze how tasks in priority-based schedule interact
  Ascertain worst-case response times
  Tasks
  Interrupts and interrupt handler

Deadline Monotonic Analysis
To work through deadline monotonic analysis

\textit{Assumptions}
  We establish some simplifying assumptions to begin with
  Will relax several as we develop the analysis

  1. We have a fixed set of tasks with known priorities
  2. Any tasks can be ready to run at any time
     Known minimum interval before can be ready again
  3. Execution duration is bounded
4. Task cannot suspend itself
   Cannot wait or block on event
5. Task deadline must be less than or equal to its period
   Related to point 2 above
6. Duplicate priorities not allowed
7. Tasks are preemptable and cannot be delayed by lower priority tasks
   Under such conditions tasks cannot
   Disable interrupts
   Share data via a semaphore
8. Time to schedule and execute context switch known and bounded

As we proceed
Will examine and modify last two assumptions

Vocabulary
Let’s now introduce some the necessary vocabulary

- $T_i$ - Period of task i
  Time between successive instances of task ready to run
- $C_i$ - Execution time on system processor for task i
  This is strictly task execution time of the $i^{th}$ task
  Does not include time for any nontask processing
  Other tasks or interrupts
- $R_i$ - Response time of task i
  Time between task
  Becoming ready to run and
  Completing worst case execution time
  Note this time will include time for any nontask processing
  Other tasks or interrupts
- $I_i$ - Time for other tasks and interrupts during task i
- $D_i$ - Deadline for task i
  Time when task must be completed
  Clearly task will always meet deadline if
  $D_i \geq R_i$

Analysis
From above definitions
Following relationship is clear
1. $R_i = C_i + I_i$
Presuming that we know task execution time $C_i$, problem reduces to finding $I_i$.

Given a second task $j$, number of times, $N$, it can preempt executing task $i$ is given by

Ceiling of ratio of response time for task $i$ and period of task $j$

Thus

2. $N = \left\lceil \frac{R_i}{T_j} \right\rceil$

For given $x$, ceiling function returns smallest integer $\geq x$

From which we compute the total time consumed by second task

3. $time = \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$

From eq 3 can easily compute time consumed for all tasks

Simply sum value over all higher priority tasks, $T_{hp}$

4. $I_i = \sum_{k \in T_{hp}} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_k$

Substituting back into 1 above we have following

5. $R_i = C_i + \sum_{k \in T_{hp}} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_k$

Which leads us to the recurrence relationship

6. $R_i^{n+1} = C_i + \sum_{k \in T_{hp}} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_k$

-14-
Now extend eq 5 to relax restriction of no blocking
To accommodate case when task can be blocked
By lower priority task
Denoted *priority inversion*

7. \( R_i = B_i + C_i + \sum_{k \in T_w} \left( \frac{R_i}{T_j} \right) \cdot C_k \)

Term \( B_i \) expresses time when such blocking occurs
Includes time for all blocking tasks

Computing value of \( B_i \) can be difficult
Placing restriction on such blocking
Makes problem tractable

**Restriction**
Higher priority task blocked at most once
By all lower priority tasks
Such restriction called
*Priority Ceiling Protocol*

**Priority Ceiling Protocol**
Stipulates that each semaphore has ceiling priority
Priority of highest priority task that can lock semaphore

Task can simultaneously hold multiple semaphores
Must be locked and unlocked in nested pattern
- Lock 1
- Lock 2
- Lock 3
- Unlock 3
- Unlock 2
- Unlock 1

**Using instant inheritance** algorithm
When task
- Locks semaphore
  Priority of task raised to ceiling priority of semaphore
- Unlocks semaphore
  Priority restored to original value
Consequence lock always succeeds
No other task can have semaphore locked
If so
Such task would have ceiling priority and be running
Current task could not be running

Based upon such knowledge
Can now compute blocking time for task i
First we define $T_{lp}$
   Set of lower priority tasks than task i

Examine set of semaphores that can be locked by each task $j$ in $T_{lp}$

Select subset of all such semaphores
   Have ceiling priority higher than that of task i
   Define such a subset as $S_j$

Define $T_{ime_{j,s}}$ as time task $j$ holds semaphore $s_k$ in $S_j$

We now have
8. $B_i = \max_{\forall j \in T_{lp}, \forall s_k \in S_j} \left( T_{ime_{j,s}} \right)$

Let’s now look at including overhead for
Scheduler and context switch

We recognize scheduler running continuously
Thus we have two components to time burden
1. Time to execute basic scheduler when no tasks need to be handled
2. Time to manage actual context switch

For a given duration and task with period $T_k$
Scheduler can be summoned number of times given by
9. $\text{invocations} = \frac{t}{T_k}$

Time burden for single task then given by
10. $\text{time} = \left\lfloor \frac{t}{T_k} \right\rfloor \cdot C_{\text{task}K}$

That for all tasks follows simply
11. \[ \text{time}_{\text{total}} = \sum_{\forall \text{tasks} \in \text{totalSchedtasks}} \left\lfloor \frac{t}{T_k} \right\rfloor \cdot C_{\text{task}K} \]

Time to execute basic scheduler for same duration

12. \[ \text{time}_{\text{scheduleBase}} = \frac{t}{T_{\text{schedule}}} \cdot C_{\text{schedule}} \]

In this case we have
- Period of the scheduler - \( T_{\text{schedule}} \)
- Worst-case execution time of scheduler - \( C_{\text{schedule}} \)

Combining we now have

13. \[ \text{time}_{\text{schedule}} = S_i = \frac{t}{T_{\text{schedule}}} \cdot C_{\text{schedule}} + \sum_{\forall \text{tasks} \in \text{totalSchedtasks}} \left\lfloor \frac{t}{T_k} \right\rfloor \cdot C_{\text{task}K} \]

Adding the time burden for scheduler overhead to our response time for task \( i \) we have

14. \[ R_i = S_i + B_i + C_i + \sum_{k \in T_{kp}} \left\lfloor \frac{R_i}{T_j} \right\rfloor \cdot C_k \]

The final piece of the equation includes
- Time burden of context switch, \( C_{\text{switch}} \)

Burden comprises several components
1. Time to suspend and save task \( i \)
2. Time to restore and resume task \( i \)
3. Time to activate preempting tasks
4. Time to suspend and save preempting tasks

First two items
- Included once

Latter two items must be included for each possible preempting task

Adding this final component we now have

15. \[ R_i = S_i + B_i + C_i + C_{\text{switch}} + \sum_{k \in T_{kp}} \left\lfloor \frac{R_i}{T_j} \right\rfloor \cdot (C_k + C_{\text{switch}}) \]
In the equation above
First instance of $C_{\text{switch}}$
Covers items 1 and 2
Second instance
Covers items 3 and 4

**Time – A More Detailed Look**

Let’s now go inside the individual hardware components and software tasks
See how time affects their performance

When looking at the time performance in embedded systems
Must consider both hardware and software timing

**Hardware**
Must consider
- Internal delays of hardware components
- Delays through external elements or systems as appropriate

**Software**
Affected by both
- Path through program
- Timing of individual instructions

**Measures**

*Response time*
Interval between occurrence of event and completion of associated action
Event can be
- Internal
- External
- Timer
- State change

Also sometimes 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
Although not time per se
Included here because can have indirect affect on timing

In examining time
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

**Polling 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 $\tau_{ed1}$

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 $\tau_f$

**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 $\tau_p$

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
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)

Several primary methods
  Complexity analysis
  Instruction counting
  Simulator or emulator
  Timer connected to various busses in system
  Measurement instruments

Complexity Analysis
  Complexity analysis gives us a high-level view

For our analysis begin with
  Abstract machine that uses steps of time and units of memory

Complexity analysis
  Working at macroscopic level
  Gives tool to evaluate
    Different algorithms
    Large chunks of code
With instruction counting
  Will work at more microscopic level
  Deal with specific machine and limited blocks of code
  Goal to optimize specific piece of code

For the abstract machine
  Instead of seconds or bytes
    Each elementary operation takes 1 step
Each elementary instance occupies 1 unit of memory
Need to quantify what comprises an elementary operation

Measure time and space in terms of the size of the input
Rather than details of the specific input
Allows us to focus on big issues

Example

// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

```c
int Sum(int A[], int N)
{
    int sum = 0;
    for ( int j = 0; j < N; j++)
        sum = sum + A[j];
    return sum;
}
```

How should we analyze this?
Analysis of Sum
First
    Describe the size of the input in terms of one or more parameters
    Input to Sum is an array of N ints, so size is N.
Second
    Count how many steps are used for an input of that size
    A step is an elementary operation such as + or < or A[j]

Analysis of Sum (2)
• 1, 2, 8: Once
• 3, 4, 5, 6, 7: Once per each iteration of for-loop
  N iterations
• Total is 5N + 3 operations

We can view this as a function of N
  Complexity function of the algorithm:
  \[ f(N) = 5N + 3 \]

How 5N +3 Grows
  The 5N +3 analysis gives an estimate of the true running time for different
  values of N:

<table>
<thead>
<tr>
<th>N</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>53 steps</td>
</tr>
<tr>
<td>100</td>
<td>503 steps</td>
</tr>
<tr>
<td>1,000</td>
<td>5,003 steps</td>
</tr>
<tr>
<td>1,000,000</td>
<td>5,000,003 steps</td>
</tr>
</tbody>
</table>

As N grows
  The number of steps grows in linear proportion to N, for this Sum function

Methodology
  The example was typical
  • Analyze a program by counting steps
  • Derive a formula
    Based in some parameter N that is the size of the problem
    For example, one algorithm might have a formula of N^2
    Another might be 2^N
Look at the formula to understand the overall efficiency

Questions
  • Why is this Useful?
  • What happens when we double the input size N?

<table>
<thead>
<tr>
<th>N</th>
<th>$\log_2 N$</th>
<th>5N</th>
<th>$N \log_2 N$</th>
<th>$N^2$</th>
<th>$2^N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>40</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>80</td>
<td>64</td>
<td>256</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>160</td>
<td>1024</td>
<td>~10^9</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>320</td>
<td>384</td>
<td>4096</td>
<td>~10^19</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>640</td>
<td>896</td>
<td>16384</td>
<td>~10^38</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>1280</td>
<td>2048</td>
<td>65536</td>
<td>~10^76</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
<td>50000</td>
<td>10^5</td>
<td>10^8</td>
<td>~10^3010</td>
</tr>
</tbody>
</table>

Isn't This Totally Bogus?
Need to run faster?
  • Buy a faster computer
  • Or just wait a while: CPU speed doubles every 18 months or so
    "Moore's Law"

Suppose we could make the CPU 1,000,000 times faster
How much would that help if the algorithm is $2^N$?

Let's look at the following chart
If We Sped Up the CPU...
Even speeding up by a factor of a million, $10^{3010}$ is only reduced to $10^{3004}$

<table>
<thead>
<tr>
<th>N</th>
<th>$\log_2 N$</th>
<th>5N</th>
<th>$N \log_2 N$</th>
<th>$N^2$</th>
<th>$2^N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>40</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>80</td>
<td>64</td>
<td>256</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>160</td>
<td>1024</td>
<td>~10^9</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>320</td>
<td>384</td>
<td>4096</td>
<td>~10^19</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>640</td>
<td>896</td>
<td>16384</td>
<td>~10^38</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>1280</td>
<td>2048</td>
<td>65536</td>
<td>~10^76</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
<td>50000</td>
<td>10^5</td>
<td>10^8</td>
<td>~10^3010</td>
</tr>
</tbody>
</table>

Big Numbers
Suppose a program has run time proportional to $n!$, $n$ factorial
Suppose the run time for $n = 10$ is 1 second
Let's do a few back of the envelope calculations
The time for 12 is $12! = 12 \times 11 \times 10!$
Which is 132 times longer than 1 second: 132 seconds

For $n = 12$, the run time is 2+ minutes

For $n = 14$, the run time is 6 hours
$14 \times 13 \times 12 \times 11$ times longer
For $n = 16$, the run time is 2 months
For $n = 18$, the run time is 50 years
For $n = 20$, the run time is 200 centuries

**What Dominates?**
What about the 5 in $5N + 3$? What about the +3?
As $N$ gets large, the +3 becomes insignificant
The 5 is inaccurate:
- $<$, $[$, $+$, $=$, $++$ require varying amounts of time;
- Different computers by and large differ by a constant factor
What is fundamental is that the time is *linear* in $N$

**Asymptotic Complexity:**
As $N$ gets large,
Concentrate on the *highest order* term
Drop lower order terms such as $+3$
Drop the constant coefficient of the highest order term
Asymptotic Complexity
The $5N + 3$ time bound is said to "grow asymptotically" like $N$
This gives us an approximation of the complexity of the algorithm
Ignores lots of details,
Concentrates on the bigger picture

Comparing Algorithms
We can now (partially) answer the question,
"Given algorithms $A$ and $B$, which is more efficient?"
Same as asking
"Which algorithm has the smaller asymptotic time bound?"
For specific values of N, we might get different (and uninformative) answers
Instead, compare the
Growth rates for arbitrarily large values of N
This is the asymptotic case

Comparing Functions
Definition:
If f(N) and g(N) are two complexity functions, we say
f(N) = O(g(N))
read "f(N) is order g(N)"
"f(N) is big-O of g(N)"

If there is a constant c such that
f(N) ≤ c g(N)
for all sufficiently large N.

Big-O Notation
Think of f(N) = O(g(N)) as
"f(N) grows at most like g(N)" or
"f grows no faster than g"
Ignoring constant factors, and for large N
The behaviour of f(n) is bounded by g(n)

We’re interested in the rate of change of a function
Not an absolute value

Big-O is not a function it’s a notation
A means of describing something
Never read = as "equals"!

Definition: Function f(n) is O(g(n)) if there is a constant K and a count n₀, such that f(n) is ≤ K * g(n), for n ≥ n₀.

Examples
5N + 3 = O(N)
Also true: 5N + 3 = O(N²)
That is - grows no faster than N² or N! for that matter
We want the
Tightest bound
Best indication of performance

\[ 37N^5 + 7N^2 - 2N + 1 = O(N^5) \]

Typical questions:
- What is the worst case performance (upper bound) of a particular algorithm?
- What is the average case performance of a particular algorithm?
- What is the best possible performance (lower bound) for a particular type of problem?

Many difficult questions
Complicated mathematics

Common Orders of Growth
Let \( N \) be the input size

| \( \frac{k}{k} \) | \( O(1) \) | Constant Time, complexity is independent of number of data items |
| \( \log_b N \) | \( O(\log N) \) | Logarithmic Time |
| \( N \) | \( O(N) \) | Linear Time, complexity proportional to size |
| \( N \log N \) | \( O(N \log N) \) | |
| \( N^2 \) | \( O(N^2) \) | Quadratic Time |
| \( N^3 \) | \( O(N^3) \) | Cubic Time |
| ... | \( O(k^N) \) | Exponential Time |
| \( N \) \( \text{any integer} \) | | is called “polynomial” time |

Why is this Useful?
As inputs get larger
Any algorithm of a smaller order will be more efficient than an algorithm of a larger order

Big-O Arithmetic

- Remember common functions in order from smallest to largest:
  1, log(N), N, N \log(N), N^2, N^3, ..., 2^N, 3^N, ...
- Ignore constant multipliers
  \[ 300N + 5N^4 + 6 \cdot 2^N = O(N + N^4 + 2^N) \]
- Ignore everything except the highest order term
  \[ N + N^4 + 2^N = O(2^N) \]

Constant Time Statements

Simplest case: \( O(1) \) time statements
- Assignment statements of simple data types
  \( \text{int } x = y; \)
- Arithmetic operations
  \( x = 5 \times y + 4 \times z; \)
- Array referencing
  \( A[j] \)
- Referencing/dereferencing pointers
  \( \text{Cursor} = \text{Head} \rightarrow \text{Next}; \)
- Declarations of simple data types
  \( \text{int } x, y; \)
- Most conditional tests
  \( \text{if ( } x < 12 \text{ )} ... \)

Analyzing Loops

Any loop analysis has two parts:
1. How many iterations are performed?
2. How many steps per iteration?
For Loops

```
int sum = 0;
for (int j = 0; j < N; j++)
    sum = sum + j;
```

Loop executes $N$ times ($0 \ldots N-1$)
4 = $O(1)$ steps per iteration
the sum $\sum j$
the assignment $sum =$
the auto increment $j++$
the comparison $j < N$
Total time is $N \cdot O(1) = O(N \cdot 1) = O(N)$

What about this for-loop?

```
int sum = 0;
for (int j = 0; j < 100; j++)
    sum = sum + j;
```

Loop executes 100 times ($0 \ldots 99$)
4 = $O(1)$ steps per iteration
Total time is $100 \cdot O(1) = O(100 \cdot 1) = O(100)$

That this loop is faster makes sense when $N >> 100$

While Loops

What about while-loops?
Determine how many times the loop will be executed

```
bool done = false;
int result = 1
int n;    // n has some value
while ( !done )
{
    result = result * n;
    n--;
    if ( n <= 1 ) done = true;
}
```

```
Loop terminates when done == true, which happens after n iterations
O(1) time per iteration
O(n) total time

Nested Loops
Treat just like a single loop, and evaluate each level of nesting as needed:

```c
int j, k, sum = 0;
for ( j = 0; j < N; j++ )
    for ( k = N; k > 0; k-- )
        sum += k + j;
```

Start with outer loop:
How many iterations?  N
How much time per iteration?

Need to evaluate inner loop ...
Inner loop uses O(N) time
Total is N · O(N) = O(N · N) = O(N²)

What if the number of iterations of one loop depends on the counter of the other?

```c
int j, k, sum = 0;
for ( j = 0; j < N; j++ )
    for ( k = 0; k < j; k++ )
        sum += k * j;
```

Analyze inner and outer loops together
Number of iterations of the inner loop is
0 + 1 + 2 + ... + (N-1) = O(N²)
Time per iteration is O(1), for total O(N²)
Sequences of Statements
For a sequence of statements, compute their cost functions individually and add them up

```
for (int j = 0; j < N; j++)
    for (int k = 0; k < j; k++)
        sum = sum + j*k;

for (int l = 0; l < N; l++)
    sum = sum - l;
```

Total cost is $O(N^2) + O(N) + O(1) = O(N^2)$

Conditional Statements
What about a conditional statement such as

```
if (condition)
    statement1;
else
    statement2;
```

where statement1 runs in $O(n)$ time and statement2 runs in $O(n^2)$ time?
We use “worst-case complexity”
Among all inputs of size n, what is the maximum running time?
The analysis for the example above is $O(n^2)$

Cost of Function Calls
$F(b, c)$;
Cost = cost of making the call + cost of passing the arguments + cost of executing the function

- Making and returning from the call $O(1)$
- Passing the arguments Depends on how they are passed
- Cost of execution Must do analysis of the function itself
Efficiency in Parameter Passing

- Pass by value -- copies entire structure
  `Translate(CodeBook cb);`
  What if there's a copy constructor?
- Pass by reference -- does not copy, but allows updates
  `Translate(CodeBook & cb);`
  `Translate(CodeBook * cb);`

Recursive Algorithms

We need to know two things:

- Number of recursive calls
- The work done at each level of recursion

Exponentiation

The running time is $O(n)$, because there are $n$ recursive calls, and the work done at each call is constant

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

We now look at the microscopic view

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

```c
int exp (int x, int n)
{
    if (n==0)
        return 1;
    else
        return x * exp(x,n-1);
}
```
May wish to use
Min in 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
$$T = \sum_{i=1}^{n} \frac{A_i}{T_i}$$

To be able to effectively do instruction counting
Must understand the basic flow of control through a piece of software
Often altering the flow of control involves context switch.

Context
Context is information that characterizes
Current executing environment of the program
Includes items such as,
- Program counter
- Auto variables
- Register contents
- State of globals

Flow of Control and Context Switches

Context Switch
Change from current context to new one
May involve
Saving current one
Retrieving old one
Depending upon tasks to be performed
Takes varying amounts of time
Can be critical in real time system

Flow of Control
As we design and build increasingly complex embedded systems
Not going to be able to develop a solution
Based upon executing a few lines of code
Rather designs are going to be implemented as
Collections of co-operating tasks and threads

Some of the tasks
Scheduled according to a predetermined algorithm
Others will be invoked asynchronously in response to internal or external events

The invocation of each new task or thread
Entail a change from a current context to a new one
May involve
Saving the current context
Retrieving an old one

Depending upon the tasks to be performed
Such a series of operations will take varying amounts of time
Can be critical in a real time system
To be able to effectively design embedded systems
Must thoroughly understand the flow of control
Both inside and outside of our system
Essential to such understanding
Ability to analyze, in time
The flow of control through our system.

The flow of control through most contemporary programs
Can be modeled and analyzed as a composite of 4 basic elements
We identify these as
• Sequential component
- Branch
- Loop
- Procedure call

Let’s look at each of these
  First at a high level then in greater detail

Sequential
  Each instruction executed in sequence

Branch
  Select one of several branches based upon condition
  Graphically
  Type of construct seen in
    if
    else
    switch or case

Loop
  Repeatedly execute set of instructions
  Forever
  Until some condition met
  Can make decision
  Before
    Code may not be executed
  After loop
    Code executed at least once
  Type of construct seen in
    do
    or repeat
    while
    for

Procedure
  Leave current context
  Execute set of instructions
  Return to context
  Type of construct seen for
    Procedure or subroutine call
    Interrupt handler
    Co-routine
Assembler

Let’s now look at assembly language level.

Sequential

The following code fragment represents a typical set of instructions that one might encounter in a C sequential flow.

```c
// make a couple of declarations
Int a = 10;
Int b = 20;
// perform an arithmetic operation followed by an assignment
c = a + b;
```

The C fragment is now expressed in assembler. We annotate each instruction with its execution time. The processor on which these times are computed is running at 20 MHz. Such a clock rate is not unusual for an embedded microcontroller. It is sufficient for many smaller applications. The process we’re using can be applied to any machine.

```
ldbse R0,#0AH  // load 10 into a temp register  400 ns
push R0       // push the local variable onto the stack  600 ns
ldbse R2,#14H // load 20 into a temp register  400 ns
push R2       // push the local variable onto the stack  600 ns
add R0,R2     // c <- a + b  400 ns
push R0       // push the local variable onto the stack  600 ns
```

Total 3000 ns
Branch
if - else construct

```c
if (a == b)
    c = d + e;
else
    c = d - e;
```

Assume a .. e in registers R1..R5

Observe that for the branch construct
We have different execution times for
case when the branch is taken and when it isn’t.
We have to consider these numbers in context
If we have a hard real time constraint associated with this branch
We have to use the worst case number or 2800…
This is the best we can guarantee

```
cmp    R1,R2   //  compare R1 and R2    400 ns
jne    @0002   //  if they are not equal branch to     800 ns for branch taken
@0002: //  label @0002                    400 ns for branch not taken
add    R3,R4,R5 //  R3 <- R4 + R5          500 ns
br     @0003   //  go to label @0003      700 ns
@0003:  //  R3 <- R4 - R5                 500 ns

Total   1700 – 2000 ns
```

On the other hand, if we are looking at general performance
We might consider averaging the two values

Loop

```c
while (myVar < 10)
{
    i = i + 2;
    myVar++;
}
```
We assume that myVar has been loaded into R1 and i into R0

```
@0004:  // start of the while loop
  cmp  R0,#0AH // compare myVar with 10  // 400 ns
  jge  @0005  // if greater than or equal 10  // 800 ns for branch taken
         // jump to label @0005  // 400 for branch not taken
  add  R0,#2 // increment I by 2  // 500 ns
  inc  R1   // increment myVar  // 300 ns
  br   @0004 // branch to start of while loop  // 700 ns
@0005:
```

**Procedure Call**

Most complex of flow of control constructs

Not more difficult

More involved

Will include

Procedures

Subroutines

Co-routines

**Process**

We’ll consider from high level

Program loaded at address 3000

Code executed until address 3053

Procedure encountered

1. Save return address
2. Address of procedure 5000 put into PC
3. Instruction at 5000 begins executing
4. Execution continues until 5053
5. Return encountered

Action similar to call

Stack gets

Return values

---

3000 Code
3053 Procedure Call F1
3054 More Code
5000 F1

Procedure Code
5053 Return
Stack looses
Return address
6. Return Address put into PC
7. Execution continues at 3057

Had procedure call been encountered in procedure F1
Identical process repeated
Can be repeated multiple times
Must be aware that stack can overflow
If too much pushed on
Begin to loose information
Particularly return address

We declare a couple of variables
Pass one to the function
Use the other to hold the return value

```cpp
int myVar0 = 30;
int myVar1 = 40;
myVar1 = myFunction(myVar0);
```

We'll look first at the code in `main()` then in the body of the function.

```cpp
int myFunction(int aVar)
{
    int localVar = 15;
    localVar = localVar + aVar;
    return localVar;
}
```

We assume
myVar0 has been stored in R0 and myVar1 has been stored in R1.
The code in the function body appears as,

```
push myVar0  // push myVar0 onto the stack       600 ns
lcall myFunction  // execute the function call         1100 ns
    // the call pushes the contents of the PC
    // (the return address) onto the stack, adds
    // the displacement between the current PC and
    // the address of the function to the PC
add SP,#2  // increment the stack pointer          400 ns
ld myVar1,Tmp0  // put the function return value into myVar1  400 ns
```

The assembler implementation now given below

```
push Tmp01  // pushes a temp variable onto the stack   600 ns
ld Tmp01,SP  // puts the stack pointer into that local variable  400 ns
ldbse localVar,#0FH  // load 15 into a temp register     400 ns
add localVar,aVar[Tmp01]  // use indexing into stack to get passed in arg  600 ns
ld Tmp0, localVar  // use global temp to return local variable  400 ns
pop Tmp01     // pop the stack         800 ns
ret            // return to calling function                1100 ns
    // puts top of stack into PC
```

Had there been a second procedure call in procedure \textit{myFunction()}
An identical process would have been initiated
Such a process can be repeated multiple times
We must be aware that the stack can overflow if too much pushed on
Under such a circumstance we begin to loose information return address.

We can see that the function call has consumed almost 7 \(\mu\)s.
Observe how the authors of the compiler for this processor
Handled the return value
From the times we see above
The push and pop operations consume 1.4 \(\mu\)s
In contrast to the load which requires 400ns
By using a global temporary register rather than the stack
They improved performance by almost 300% on that sequence

Coroutine
Special kind of procedure call
Mutual call exchange between
Co-operating procedures
Exclusive procedures
Sharing time
Mechanics are the same as procedure call
Major difference
Conventional procedure executes until end
Unless leaves under extraordinary circumstances
Co-routines
Exit and return
Throughout body of procedure
Usually under direction
Third process or procedure

Graphically process appears as shown

Control procedure starts process
Context switch determined by
Control procedure
External event
Timing signal
Internal event
Data value
Process continues until procedures completed
With each switch
Appropriate information from current context
Must be saved

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

Timers
Can connect timer to various busses in system
Typically
Memory address and data busses
Can use flag to start and stop timer
Best for timing blocks of code
For long program
Not particularly effective

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 Considerations
   We will now continue our discussion of
      Performance considerations in embedded systems
   We identified throughput, time loading, and memory loading
      As three simple measures of performance
   We have discussed the first two
   We'll now examine memory management
   Then move to
      Performance optimization
      Identify some of the common mistakes people make
         When evaluating performance
      Conclude with some tricks of the trade for optimizing performance.