Abstract
Performance Design is based on the application on many performance oriented patterns. Patterns are a way are to consolidate experience: what solution fits to what problem in what situation? Pitfalls are also a way to consolidate experience: what are common design mistakes?
Case 6

Common Platforms and Bloating

Generic nature of platforms

Most SW implementations are way too big

Performance suffers from oversize and generic provisions
>90% of all Software statements are not needed, but caused by:
  over-specification
  bad design
  too generic
  dogmatic rules
  legacy remains

core function
less than 10%

legend
overhead
value
Necessary Functionality $\gg$ Intended Regular Function

- testing
- regular functionality
- instrumentation
  - diagnostics
  - tracing
  - asserts
- boundary behavior:
  - exceptional cases
  - error handling

Performance Patterns, Pitfalls, and Approach
©2006, Embedded Systems Institute
©2006, Gerrit Muller
The Danger of Being Generic: Bloating

"Real-life" example: redesigned Tool super-class and descendants, ca 1994
Problem Propagation via Copy & Paste

needed code

bad code

new needed code

needed code

code not relevant for new function

repair code

bad code

new bad code

copy
paste
modify
Example of Problem Propagation

Class Old:
    capacity = startCapacity
    values = int(capacity)
    size = 0
    def insert(val):
        values[size]=val
        size+=1
        if size>capacity:
            capacity*=2
            relocate(values, capacity)

Class New:
    capacity = 1
    values = int(capacity)
    size = 0
    def insert(val):
        values[size]=val
        size+=1
        capacity+=1
        relocate(values, capacity)

Class DoubleNew:
    capacity = 1
    values = int(capacity)
    size = 0
    def insert(val):
        values[size]=val
        size+=1
        capacity+=1
        relocate(values, capacity)
    def insertBlock(v,len):
        for i=1 to len:
            insert(v[i])
Overhead Penalty of Modularity

- Overhead
- Performance Patterns, Pitfalls, and Approach
- Copyright © 2006, Embedded Systems Institute
- Embedded Systems Institute
- March 6, 2013
- EASRT Call Tree

- modular
- fine grain
- monolithic
- coarse grain
- medium grain
- value
- overhead

- 81%
- 63%
- 44%

- version: 0.1
- ©2006, Gerrit Muller
Function Call Overhead

Load and depth dependent (hidden) side effects

pipeline flush
I-cache disturbance
D-cache disturbance

overhead

value

legenda

Performance Patterns, Pitfalls, and Approach
©2006, Embedded Systems Institute

version: 0.1
March 6, 2013
EASRTcallOverhead
Suppose:

Call Overhead = 10µs
Call graph branching factor = 2
Depth = 12

What is the Call overhead when all branches are followed?
Exercise Frame Rate for Layered SW

Suppose:

Function call = 10µs
Call layer depth = 20
1024 calls per image

What is the maximum frame rate possible assuming that the complete CPU time is available for function calls?
Common Platforms and Bloating

Platforms are overprovisioned and very generic

Are benefits > disadvantages?

Performance loss is significant and can be measured and modelled
Multi-Dimensional Viewing of many Images: Greedy and Lazy Design Patterns
Greedy and Lazy systems

Greedy: pre-fetched lots of data:
System tries to have data available for the requesting system

Lazy: hardly or no pre-fetching of data:
System tries to set data available for the requesting system only when asked for
**Example Greedy / Lazy (1)**

<table>
<thead>
<tr>
<th>META DATA</th>
<th>META DATA</th>
<th>META DATA</th>
<th>META DATA</th>
<th>META DATA</th>
<th>META DATA</th>
<th>META DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>META DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>META DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>META DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>META DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>META DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**META DATA**
- Patient name
- Slice nr. / position
- Annotation
- Explanation
- Date / time
Design Options

Example Greedy / Lazy (2)

Lazy: Fetch only the requested image

Greedy: Fetch all the images in the set

In between options:
- Fetch requested image + surrounding images
- Fetch requested image + only meta information of images
Example Greedy / Lazy (3)

Lazy:
- low load on system
- long waiting time for next image

Greedy:
- high load on system
- possible long initial wait
- short response time insteady state

In between options:
- medium system load
- fast response for initialization and common image fetches
Initialization, Steady State and Finalization
Start-up, Steady State, Shut Down
Start-up, Steady State, Shut Down Scheme

- **Load**
  - Configure
  - Initialise, start

- **Discover Kernel HW**
  - Initialise kernel data structures
  - Determine next layer

- **Load and Initialise Loader**
  - Determine loading HW
  - Determine next layer

- **Bring in Initial State**
  - Load and initialise firmware

- **Configure Services**
  - Allocate resources
  - Load, initialise and start services

- **Configure UI**
  - Allocate resources
  - Load, initialise and start UI

- **Connect to Outside**
  - Detect external services
  - Publish internal services
  - Connect where needed

- **User Interface**
  - Connect to outside

- **Services**
  - Configure services
  - Allocate resources
  - Load, initialise and start services

- **Kernel**
  - Discover kernel HW
  - Initialise kernel data structures
  - Determine next layer

- **Boot-Loader**
  - Load and initialise loader
  - Determine loading HW
  - Determine next layer

- **HW**
  - Bring in initial state
  - Load and initialise firmware

- **Power**
  - Stop in safe sequence

- **Start Up**
  - Power

- **Shut Down**
  - Shut down
  - Close connections
  - Save persistent data
  - Free resources
  - Stop

**HW SW Interface**
Trade-off:

**Optimize on steady state**
may result in
poor performance for initialization
and process finish

**Optimize on Initialization and/or finish**
may result in
poor steady state performance
Common Performance Pitfalls

- Overhead
- Data bloating
- Cache thrashing
- Layering
- Process communication
- Conversions
- Serialization
- Backfiring optimalisations
- Hidden loads (bus, DMA etc)
- Poor algorithms
- Wrong dimensioning
The ASP ™ course is partially derived from the EXARCH course developed at Philips CTT by Ton Kostelijk and Gerrit Muller.

Extensions and additional slides have been developed at ESI by Teun Hendriks, Roland Mathijssen and Gerrit Muller.