Multicore processors and hard real-time applications: Mixing oil and water?
Mark Pitchford, Technical Specialist
mark.pitchford@ldra.com
1. The significance of open- and closed-loop control

2. Multicore and WCET

3. A holistic approach: Execution time analysis

4. A holistic approach: Dealing with the implications

5. Summary and conclusions: Mixing oil and water
The significance of open- and closed-loop control

Multicore and WCET

A holistic approach: Execution time analysis

A holistic approach: Dealing with the implications

Summary and conclusions: Mixing oil and water
Multicore processors – part of our everyday lives

- Multicore processors (MCPs) have been around in personal computers since the early 2000s
- They address the problem of single core designs hitting the ceiling of their physical limitations
- Moving extra cores onto a single processor chip effectively multiplies the amount of data handled

The open loop control process runs at a particular speed with no deadline. Want increased speed? Then increase the power of the processor. Simple!
The closed loop process must complete its execution within an allotted time. Failure to do so will degrade system performance.
The industrial pyramid: closed loop meets open loop

- Open loop
- mSecs to Secs
- uSecs to mSecs

Levels:
- Level 0: Field Level
- Level 1: Direct Control
- Level 2: Plant Supervisory
- Level 3: Production Control
- Level 4: Production Scheduling

- Close-loop Decisions
  - Even Slower Response
  - Security

- Supervisory Control
  - Slower Real-time
  - Security

- Real-time Control
  - Safety
  - Security

- Industrial Automation Cell
- General Processing Plant
- Industrial Warehouse and AGVs
Conventionally, single-core processors have been utilized for decades in the context of hard real-time critical applications.

But multicore processors (MCPs) offer several significant benefits:
- Performance
- Power consumption
- Form-factor & size
- Availability

However, the use of a multicore processor introduces the risk of possible interference between processing cores, due to shared resources.
1. The significance of open- and closed-loop control
2. Multicore and WCET
3. A holistic approach: Execution time analysis
4. A holistic approach: Dealing with the implications
5. Summary and conclusions: Mixing oil and water
Execution time is shared so that these tasks each has short bursts of the execution time on a single core.
Task properties

- Single-core processors have been utilized for decades in the context of functional safety.
- In this environment, it is possible* to calculate a “worst case” approximation to allow sufficient CPU headroom (capacity).

Multi-tasking real-time control systems (MCP)

(SIMPLISTICALLY SPEAKING)
Now we can give each of these tasks their own core to run on!

Processing

Communication

Actuation

1 Hz task flowchart

10 Hz task flowchart

MCPs and WCET: A holistic approach
On a single-core processor, memory is dedicated to that core.

The introduction of additional cores results in those resources being shared between them. Time-related delays occur as users wait for access.

These interference channels cause the execution-time distribution to spread.

Instead of a tight peak, the distribution of execution times becomes wide with a long tail.
Multi headaches!

We are left with:

- An approximated, “worst case” applicable to a single core
- A situation where applying that calculation across multiple cores would introduce more imprecision
- Potential interference that renders that calculation irrelevant anyway
- Multiple potential interference channels across Hardware Shared Resources (HSRs) including:
  - Memory
  - Buses
  - Cache
  - I/O
  - Software components like OS objects and services

If traditional, approximated WCET calculations are compromised, then how else might we go about this?
Thoughts from the civil aviation sector

- Guidance on how to address these issues could initially be found in the CAST-32A position paper.
- They have already been formalised into the AMC 20-193 document in Europe.
- The principles it upholds are sector agnostic, and there are potential lessons for all.
1. The significance of open- and closed-loop control

2. Multicore and WCET

3. A holistic approach: Execution time analysis

4. A holistic approach: Dealing with the implications

5. Summary and conclusions: Mixing oil and water
The holistic approach

- As we will see, dealing with this issue has implications throughout the development lifecycle.
Static analysis and multicore WCET

- Static analysis does have a contribution to make to a more precise, reliable approach.
- For example, Halstead’s metrics reflect the implementation or expression of algorithms in different languages to evaluate:
  - the software module size
  - software complexity, and
  - the data flow information

The only way to ascertain how that information translates into time elapsed is by measuring it in the environment in which the code will ultimately run.
The wrapper harness allows the code under test to be exercised repeatedly while simultaneously stressing the target device.
Stressing the target: CAST-32A & A(M)C 20-193

- **CAST-32A** suggests identifying HSRs and mapping them to stressors
- A test harness “wrapper” approach includes the flexibility to specify a preferred stressor mechanism – perhaps using Stress-ng
- There are specific stressors for components such as:
  - CPU core operations,
  - built-in registers,
  - cache,
  - ram,
  - virtual memory…

### Stress-ng Repository

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stress-lockbus.c</td>
<td>core-mwc: add stress_mwc(mod()) functions for modulus(d) range</td>
</tr>
<tr>
<td>stress-locof.c</td>
<td>stress-lock(a):(fo(d)): terminate contention process with SIGALRM</td>
</tr>
<tr>
<td>stress-locofd.c</td>
<td>stress-lock(a):(fo(d)): terminate contention process with SIGALRM</td>
</tr>
<tr>
<td>stress-longimp.c</td>
<td>Update copyright to 2023</td>
</tr>
<tr>
<td>stress-loops.c</td>
<td>rework stress_strmnd()</td>
</tr>
<tr>
<td>stress-beach.c</td>
<td>Update copyright to 2023</td>
</tr>
<tr>
<td>stress-madrix.c</td>
<td>core-mwc: add stress_mwc(mod()) functions for modulus(d) range</td>
</tr>
<tr>
<td>stress-madric.c</td>
<td>core-mwc: add stress_mwc(mod()) functions for modulus(d) range</td>
</tr>
<tr>
<td>stress-matrix-3d.c</td>
<td>stress-matrix(n): remove redundant redeclaration of variable (j)</td>
</tr>
<tr>
<td>stress-matrixc.c</td>
<td>move common (x86) assembler into core-asn-x86.h</td>
</tr>
<tr>
<td>stress-membbarrier.c</td>
<td>Update copyright to 2023</td>
</tr>
<tr>
<td>stress-memcpy.c</td>
<td>rework stress_strmnd()</td>
</tr>
<tr>
<td>stress-memfd.c</td>
<td>rework stress_strmnd()</td>
</tr>
<tr>
<td>stress-memhotplug.c</td>
<td>stress-(n): replace hard coded time constants with #defined constants</td>
</tr>
<tr>
<td>stress-memrate.c</td>
<td>stress-memrate: use registers for rep stks, reduces stack loads</td>
</tr>
<tr>
<td>stress-memtrash.c</td>
<td>stress-memtrash: add memsetstood memory zero method</td>
</tr>
<tr>
<td>stress-mergesort.c</td>
<td>Update copyright to 2023</td>
</tr>
<tr>
<td>stress-minoror.c</td>
<td>stress-(n): replace hard coded time constants with #defined constants</td>
</tr>
<tr>
<td>stress-misalignd.c</td>
<td>Update copyright to 2023</td>
</tr>
<tr>
<td>stress-misnmod.c</td>
<td>core-mwc: add stress_mwc(mod()) functions for modulus(d) range</td>
</tr>
</tbody>
</table>

[https://github.com/ColinIanKing/stress-ng](https://github.com/ColinIanKing/stress-ng)
This “wrapper” principle affords the flexibility to perform execution time analysis from complete system behaviour, through a thread or process, right down to class/function/procedure level.

This approach provides the ability to “drill in” to problem areas, and not just analyse the system as a whole.
WCET – without interference

Timing Summary

- Number of Tests: 100
- Best-Case Execution Time: 57321953.100000
- Worst-Case Execution Time: 338863903.400000
- Minimal Observed Execution Time: 63691059 - Test Case 1 (Rep 1)
- Maximal Observed Execution Time: 308058094 - Test Case 1 (Rep 2)
- Mean Observed Execution Time: 115596348
WCET – with interference

Timing Summary

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Tests</td>
<td>100</td>
</tr>
<tr>
<td>Best-Case Execution Time</td>
<td>59658020.100000</td>
</tr>
<tr>
<td>Worst-Case Execution Time</td>
<td>495044356.400000</td>
</tr>
<tr>
<td>Minimal Observed Execution Time</td>
<td>66286689 - Test Case 1 (Rep 63)</td>
</tr>
<tr>
<td>Maximal Observed Execution Time</td>
<td>450040324 - Test Case 1 (Rep 38)</td>
</tr>
<tr>
<td>Mean Observed Execution Time</td>
<td>228836793</td>
</tr>
</tbody>
</table>
- Interference can be seen to have a detrimental effect on execution time.
- If WCET falls within bounds, then the interference mitigation is adequate.
- If not, then the resulting data provides the information required to further optimize the system.
CAST-32A & AMC 20-193 recommendations: WCET considerations

**MCP_Resource Usage**: The applicant has identified the available resources of the MCP and of its interconnect in the intended final configuration, has determined the demands for the resources of the MCP and of the interconnect, and has verified the available resources when all the hosted software is executing on the target MCP.

**MCP_Software_1**: The applicant has verified that all the software components hosted by the MCP comply with the Applicable Software Guidance. In particular, the applicant has verified that all the hosted software components function correctly and have sufficient time to complete their execution when all the hosted software is executing in the intended final configuration.

The way in which the applicant should demonstrate compliance with this objective depends on the type of the MCP platform:

- **MCP Platforms With Robust Partitioning**: Applicants who have verified that their MCP Platform provides both Robust Resource and Time Partitioning (as defined in this document) may verify applications separately on the MCP and determine their WCETs separately.

- **All Other MCP Platforms**: Applicants may verify separately on the MCP any software component or set of requirements for which the interference identified in the interference analysis is mitigated or is precluded by design.

Software components or sets of software requirements for which interference is not avoided or mitigated should be tested on the target MCP with all software components executing in the intended final configuration, including robustness testing of the interfaces of the MCP.

There is, however, a caveat...
Here’s how robust partitioning is configured

Configuration of Virtual Machines
“Guests run in VMs. Each VM is a complete computing environment configured by a system designer, who specifies every VM component: CPU cores, graphics controller, memory, virtual, paravirtual and physical device, etc.”


1:1 relationship between “Virtual Machines” and “partitions” is assumed for simplicity
But there’s more to it than just memory…

- This is the Xilinx ZYNQ UltraScale+ MPSoC block diagram
- EVERYTHING shaded on this diagram is a Hardware Shared Resource (HSR)!

ROBUST PARTITIONING IS DEAD – WHAT NOW? (lynx.com)
Any hardware structure inside a MCP that is shared by the CPU cores is a Hardware Shared Resource (HSR).

HSRs on the Xilinx Zynq UltraScale+ with its Arm Cortex-A53 CPU cores include AT LEAST:

- snoop control unit,
- accelerator coherency port,
- L2 cache,
- cache coherent interconnect,
- central switch,
- direct memory access controller,
- DDR RAM
- bus interface unit,
- on-chip interconnect and
- translation control unit
AND...

- The majority of HSRs cannot be cleanly partitioned because they are implicitly allocated the instant a software partition (SWP) accesses them.

- SWPs compete for this type of HSR typically on a first-come-first-served basis, and stall for numerous cycles if the HSR is busy.

- This makes restricting a SWP to its allocation of HSRs practically unachievable.

Robust partitioning DOES help because it deals with some HSRs. But...
… can this position ever be achieved?

“The majority of HSRs cannot be cleanly partitioned because they are implicitly allocated the instant a Software Partition (SWP) accesses them. SWPs compete for this type of HSR typically on a first-come-first-served basis, and stall for numerous cycles if the HSR is busy. This makes restricting a SWP to its allocation of HSRs practically unachievable and is the root of our statement that robust partitioning is dead.”

**MCP Platforms With Robust Partitioning:**

Applicants who have verified that their MCP Platform provides both Robust Resource and Time Partitioning (as defined in this document) may verify applications separately on the MCP and determine their WCETs separately.

This makes restricting a SWP to its allocation of HSRs practically unachievable and is the root of our statement that robust partitioning is dead.

For hard real-time critical systems, collating evidence of adequate resourcing is therefore highly recommended in ALL cases.

Agenda

1. The significance of open- and closed-loop control
2. Multicore and WCET
3. A holistic approach: Execution time analysis
4. A holistic approach: Dealing with the implications
5. Summary and conclusions: Mixing oil and water
Here are two definitions from DO-178C:

**Control coupling** – The manner or degree by which one software component influences the execution of another software component.

**Data coupling** – The dependence of a software component on data not exclusively under the control of that software component.

We are looking to show that the software modules affect one another only in the ways intended by the software design, ensuring that there are no unplanned, anomalous, or erroneous behaviours...

...and unplanned, anomalous, or erroneous behaviours will impact timing!
MCP Software 2: The applicant has verified that the data and control coupling between all the individual software components hosted on the same core or on different cores of the MCP has been exercised during software requirement-based testing, including exercising any interfaces between the applications via shared memory and any mechanisms to control the access to shared memory, and that the data and control coupling is correct.
AMC 20-193 recommendations: Data and Control Coupling analysis

Notes:

a) When this objective cannot be completely met during the software verification, applicants may propose to use system-level testing to exercise the data and control coupling between software components hosted on different cores.

b) Interference may occur between tasks of a single component when the tasks execute on different cores.

- Data coupling may be affected when tasks of a single component execute on different cores.
CAST recommendations: Data and Control Coupling analysis

- Control coupling is also referenced in the document.
- However, the use of MCPs is less significant to control coupling because generally no core controls any other core.

**MCP_Software 2:** The applicant has verified that the data and control coupling between all the individual software components hosted on the same core or on different cores of the MCP has been exercised during software requirement-based testing, including exercising any interfaces between the applications via shared memory and any mechanisms to control the access to shared memory, and that the data and control coupling is correct.
An iterative approach

- Potential interference challenges mean that verification and validation in the MCP environment is more empirical than for SCPs.
- There is a resulting need for feedback loops in the development process.
  
  
  - It is highly likely that interference research will lead to changes in system or software requirements.
  - Conversely, changes in system functional requirements will likely drive new interference channels or affect existing ones.
- It is imperative that project managers are equipped to adapt readily to changing requirements and configurations.
Automating requirements traceability helps project managers to keep on top of this iterative process.
1. The significance of open- and closed-loop control
2. Multicore and WCET
3. A holistic approach: Execution time analysis
4. A holistic approach: Dealing with the implications
5. Summary and conclusions: Mixing oil and water
The interference problem demands a holistic approach across the development lifecycle.

Static analysis provides focus for execution time analysis.

A wrapper test harness provides flexibility for that analysis - from single function through to system test.

Control and data coupling analysis helps to minimize timing variation.

Requirements traceability keeps tabs on the resulting iterative process.
Need more information?