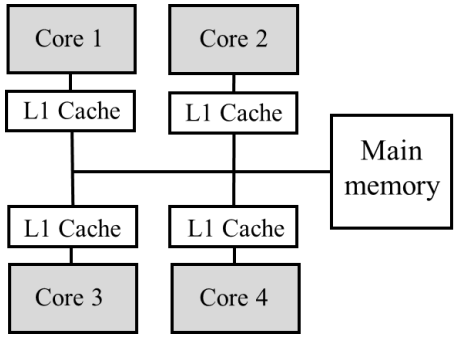
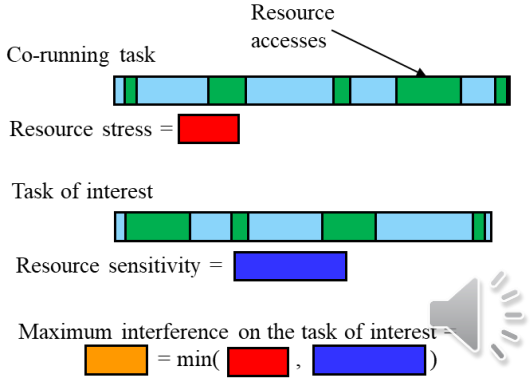
Schedulability Analysis for Multi-core Systems Accounting for Resource Stress and Sensitivity

Rob Davis, David Griffin, Iain Bate

Real-Time Systems Research Group, University of York

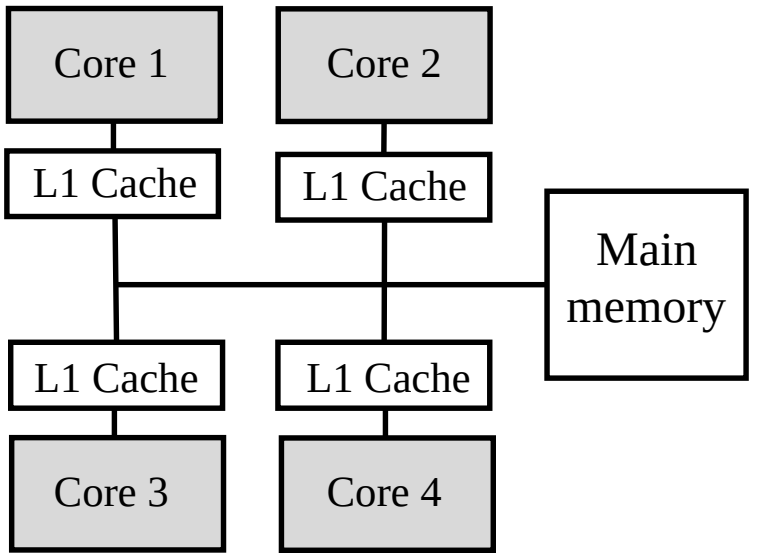




Cross-core contention in multi-cores

- **COTS Multi-core hardware**
  - Designed to deliver high average-case performance at low cost
  - Achieved by sharing hardware resources (such as cache, interconnect or bus, and main memory) between cores

- **Cross-core contention**
  - When co-running tasks attempt to access shared hardware resources at the same time
  - This ***contention*** extends task execution times compared to the stand-alone case
  - The increase in task execution times is referred to as ***interference***
  - Problem of cross-core contention and interference has led to timing verification of multi-cores becoming a hot topic of real-time systems research over the last 10 years

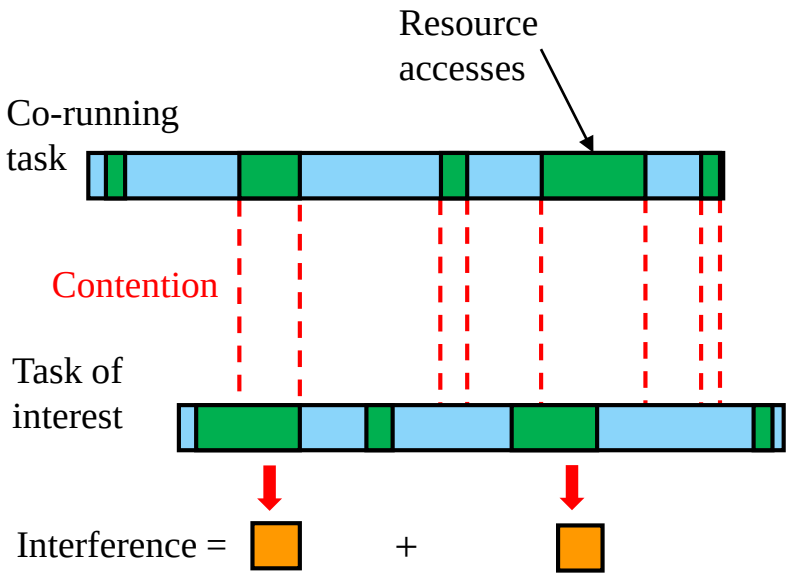


Timing verification for multi-cores

- **Academia**
  - Summarized by Maiza et al. in a survey [32] published in 2019 that classified over 120 research papers on the topic
  - The majority of the research discussed in the survey relies on detailed timing information about shared hardware resources and their arbitration policies
  - Substantial difficulties:
    - Information is often not disclosed by hardware vendors, or is incomplete, or even incorrect
    - Overall the behaviour can be so complex as to preclude a static analysis that provides meaningful bounds rather than gross over-estimates
- **Industry**
  - Common practice is to use measurement-based timing analysis [1]
    - Viable for well-designed systems on single core platforms
    - BUT, its simple extension to multi-core systems cannot provide an adequate solution that correctly bounds cross-core interference

Cross-core contention in multi-cores

- **Simple (naïve) approach**
  - Run the tasks in parallel, measure their execution times and hence interference
- **Maximum Interference**
  - Depends on precise timing of contention i.e. the accesses to shared hardware resources by the task under analysis and also the co-running tasks on other cores
  - Timing depends on the inputs, the paths taken, the data accessed etc.
  - Also depends on scheduling, task start times, and any pre-emptions
  - Impossible in practice to find the combination of behaviours that gives the maximum interference



To solve this problem we need to break it down

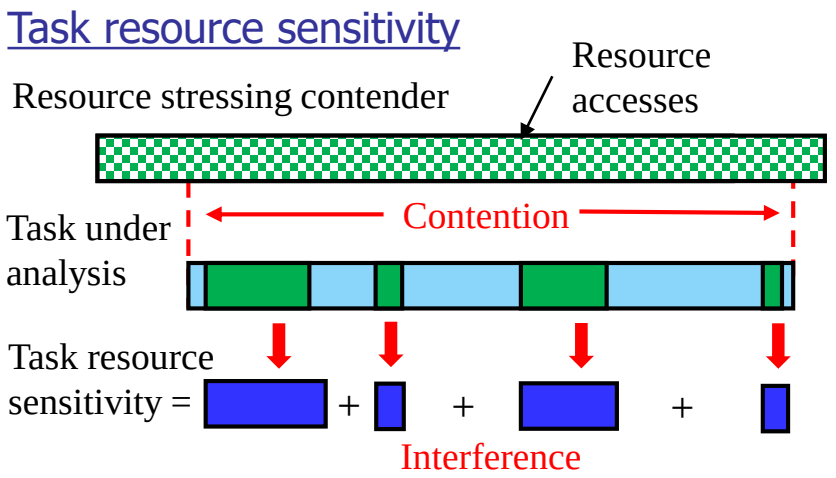
Concepts: resource stress and sensitivity

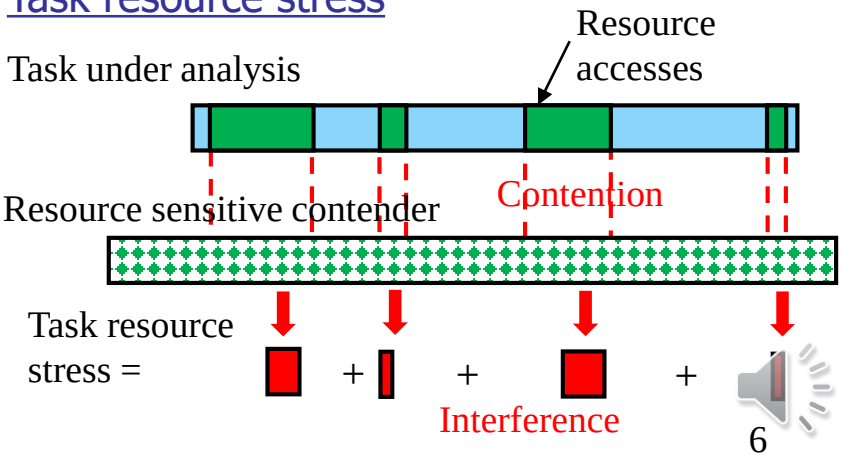
- **Resource stressing contender**
  - Maximizes the stress on a resource *r* by continuously making accesses to it that cause the most contention
  - Running a resource stressing contender in parallel with a task creates the maximum increase in execution time for the task due to contention over resource *r* emanating from any possible (single) co-runner
  - **Resource Sensitivity** X_i^r of a task τ_i equates to this increase in execution time
- **Resource sensitive contender**
  - Suffers the maximum possible interference by repeatedly making accesses to the resource *r* that suffer from the most contention
  - Running a resource sensitive contender in parallel with a task creates the maximum increase in execution time for any (single) co-running contender due to contention over the specific resource *r* emanating from that task
  - **Resource Stress** Y_i^r of a task τ_i equates to this increase in execution time of the resource sensitive contender

Note, resource stressing and resource sensitive contenders for a given shared hardware resource are not necessarily one and the same

Quantifying task resource sensitivity and task resource stress

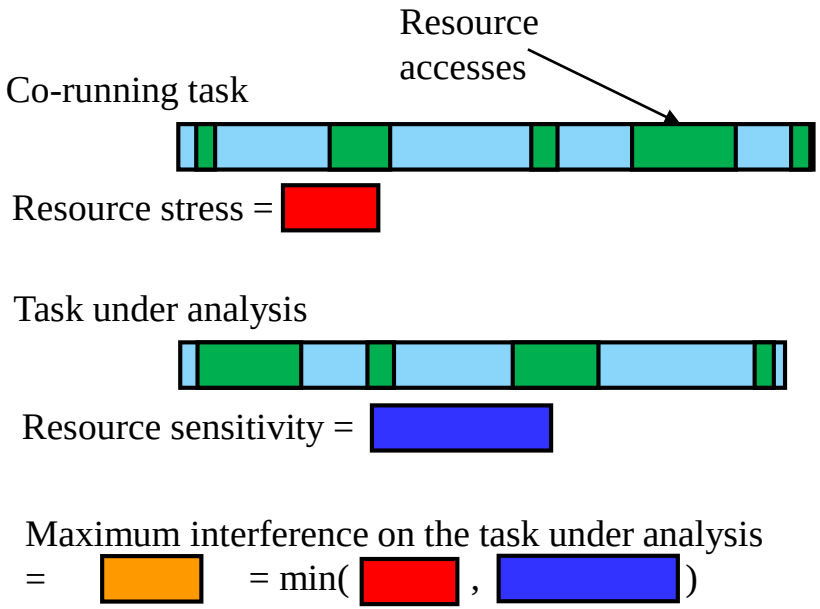
- **Using contenders**
  - Removes problematic alignment issues as the contenders continuously access the resource
  - Task resource sensitivity is obtained using a resource stressing contender
  - Task resource stress is obtained using a resource sensitive contender
  - Task resource sensitivity and task resource stress are not necessarily the same
  - Interference can be
    - Sub-additive (contention partly ameliorated by parallelism in the hardware)
    - Additive (directly reflecting the bandwidth used)
    - Super-additive (contention changes the state of the resource so subsequent operations take longer)





Quantifying cross-core contention and interference

- **Proposed approach**
  - Using resource stress and resource sensitivity for each task
- **Maximum Interference (two tasks)**
  - Bounded by resource sensitivity of the task under analysis and by the resource stress of the co-running task
  - Follows directly from how these terms are defined and also how resource stressing and resource sensitive contenders are defined



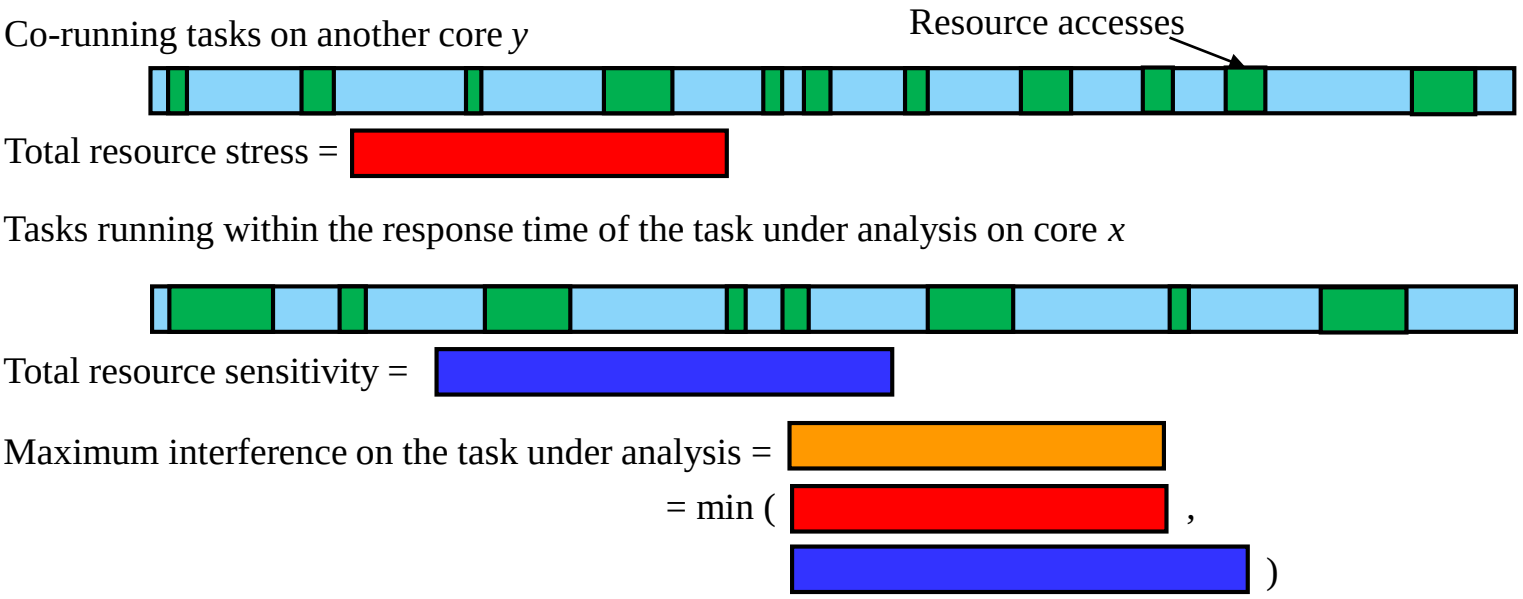
Considering interference as inflating individual execution times can be grossly pessimistic, so we need to extend these concepts to multiple tasks and their scheduling

System model

- **Processor and scheduling**
  - Multi-core with *m* homogenous cores
  - Partitioned fixed priority pre-emptive and non-pre-emptive scheduling
- **Multi-core Resource Stress and Sensitivity (MRSS) Task model**
  - Sporadic tasks, each task τ_i with stand-alone execution time C_i, minimum inter-arrival time T_i , and constrained deadline D_i
  - Set of shared hardware resources $r \in H$
  - Each task characterized by its resource sensitivity X_i^r and resource stress Y_i^r to each shared hardware resource *r*

Schedulability analysis concept

- **Consider interference over time frame of task response times**
  - Using total resource stress and total resource sensitivity occurring within that time frame
- **Worst-case interference**
  - Bounded by total resource sensitivity of jobs executing on the same core as the task under analysis within its response time. Also bounded by the total resource stress of jobs executing on another core within that response time



Schedulability analysis (pFPPS)

- **Response time analysis for partitioned Fixed Priority Pre-emptive Scheduling (pFPPS)**
  - Derives from standard RTA [5,25]

$$R_i = C_i + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \sum_{r \in H} I_i^r(R_i)$$

  - Interference $I_i^r(R_i)$ is an upper bound on the interference that may occur within the response time of task τ_i via shared hardware resource *r*, due to tasks executing on other cores

$$I_i^r(R_i) = \sum_{\forall y \neq x} \min(E_i^r(R_i, y), S_i^r(R_i, x))$$

  - Where:

$E_i^r(R_i, y)$ is the total resource stress emanating from core *y* in the response time of task τ_i

$S_i^r(R_i, x)$ is the total resource sensitivity of the tasks executing on core *x* during the response time of task τ_i

Total Resource Sensitivity

- **Formulation for total resource sensitivity** $S_i^r(R_i, x)$
  - Considering the jobs executing on core *x* within the response time of task τ_i

$$S_i^r(R_i, x) = X_i^r + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil X_j^r$$

  - Derives directly from standard RTA
  - Only jobs of same or higher priority than task τ_i can execute within its response time

Total Resource Stress

- **Formulation for total resource stress** $E_i^r(R_i, y)$
  - Consider jobs executing on another core *y* within the response time of task τ_i
  - Include jobs from all tasks on core *y* irrespective of their priority
  - **Cp-FPPS-*m*-R test** (context dependent)
    - Contention may emanate from a job anytime up to its own response time

$$E_i^r(R_i, y) = \sum_{j \in \Gamma_y} \left\lceil \frac{R_i + R_j}{T_j} \right\rceil Y_j^r$$

  - **Cp-FPPS-*m*-D test** (context dependent)
    - Contention may emanate from a job anytime up to its own deadline

$$E_i^r(R_i, y) = \sum_{j \in \Gamma_y} \left\lceil \frac{R_i + D_j}{T_j} \right\rceil Y_j^r$$

  - **Cp-FPPS-*m*-fc test** (context independent)
    - Fully composable, assumes any level of resource stress could be possible

$$E_i^r(R_i, y) = \infty$$

Dominance relation between the tests: **Cp-FPPS-*m*-R → Cp-FPPS-*m*-D → Cp-FPPS-*m*-fc**

Schedulability analysis (pFPNS)

- **Response time analysis for partitioned Fixed Priority Non-Pre-emptive Scheduling (pFPNS)**
  - Derives from a sufficient test [11]

$$R_i = \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)} (C_k) + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left(\left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) C_j + C_i + \sum_{r \in H} I_i^r(R_i)$$

- **Interference** $I_i^r(R_i) = \sum_{\forall y \neq x} \min(E_i^r(R_i, y), S_i^r(R_i, x))$
- **Formulation for total resource sensitivity** $S_i^r(R_i, x)$
  - Derives from considering the jobs executing on core *x* within the response time of task τ_i

$$S_i^r(R_i, x) = \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)} (X_k^r) + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left(\left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) X_j^r + X_i^r$$

- **Formulation for total resource stress** $E_i^r(R_i, y)$
  - Same as before forming three tests with dominance relations

    Cp-FPNS-*m*-R → Cp-FPNS-*m*-D → Cp-FPNS-*m*-fc

Priority assignment

- **Preemptive case:**
  - **Cp-FPPS-*m*-fc** (context independent): equates to standard RTA for FPPS with the execution time of each task increased according to its resource sensitivity therefore Deadline Monotonic Priority Ordering is optimal
  - **Cp-FPPS-*m*-D** (context dependent): Deadline Monotonic Priority Ordering is optimal (proof in the paper)
  - **Cp-FPPS-*m*-R** (context dependent): Audsley's Optimal Priority Assignment (OPA) algorithm is not applicable (proof in the paper)

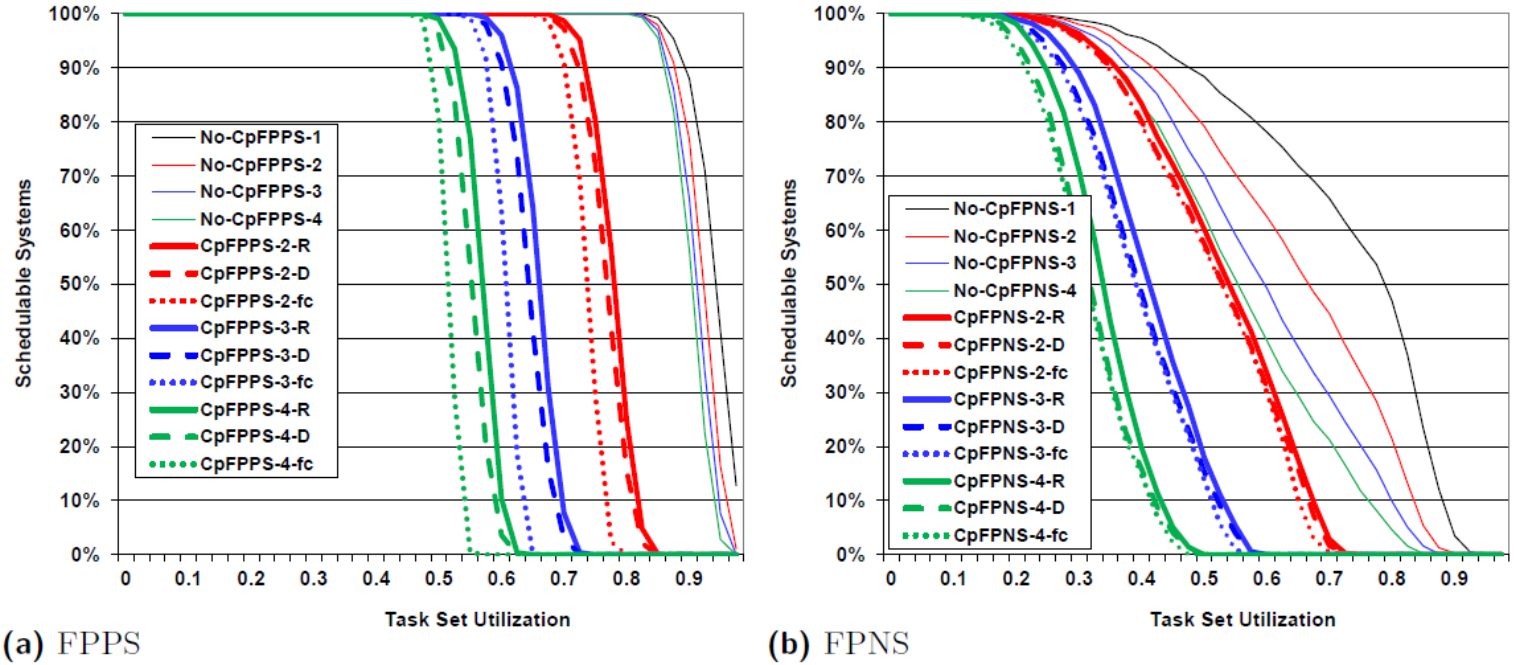
- **Non-preemptive case:**
  - **Cp-FPNS-*m*-fc** (context independent): equates to standard RTA for FPNS with the execution time of each task increased according to its resource sensitivity therefore Audsley's OPA algorithm is optimal
  - **Cp-FPNS-*m*-D** (context dependent): Audsley's OPA algorithm is optimal (proof in the paper)
  - **Cp-FPNS-*m*-R** (context dependent): Audsley's OPA algorithm is not applicable (proof in the paper)

Schedulability test performance

- **Parameter settings**
  - Considered tasks on 1-4 cores of a 4 core system with 10 tasks on each core
  - Same task set utilization was used on each core, but different task sets
  - Task utilizations were generated using the DRS algorithm [20] summing to the total utilization required for each core
  - Task periods generated according to a log normal distribution in the range 1-100ms (10-100ms for non-pre-emptive scheduling)
  - Deadlines equal to periods
  - Stand-alone execution time $C_i = U_i \cdot T_i$
  - Resource sensitivity values were generated using the DRS algorithm such that the total resource sensitivity utilization of each task set was Sensitivity Factor (SF) times the task set utilization (default SF = 0.25) and the individual task resource sensitivity values did not exceed the stand-alone execution times
  - Task resource stress values were set to Stress Factor (RF) times the task resource sensitivity values (default RF = 0.5)
  - Task priorities were set according to Deadline Monotonic Priority Ordering
  - Per core task set utilization was varied from 0.05 to 0.95 in steps of 0.025
  - 1000 task sets were generated per utilization value

Evaluation results

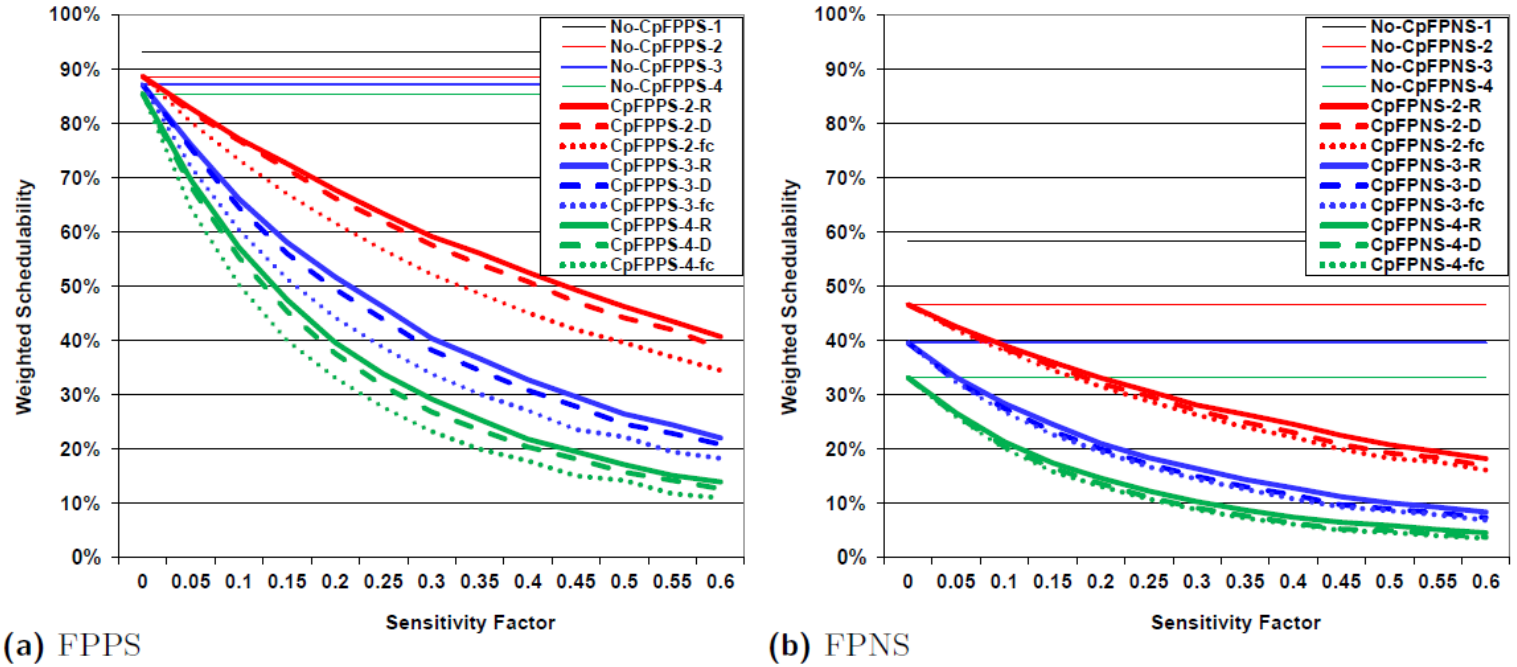
- **Success ratio: varying utilization**



- More cores equate to more interference and hence lower schedulability for all tests: 2 cores (red) vs. 3 cores (blue) vs. 4 cores (green)
- Dominance relations between the three types of test evident in the comparison between the thick solid lines (-R tests), dashed lines (-D tests), and dotted lines (-fc tests)

Evaluation results

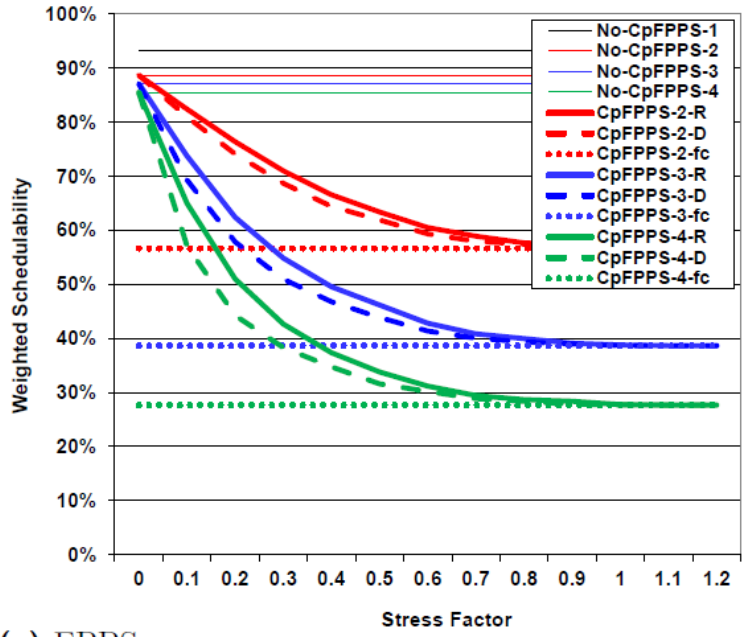
- **Weighted schedulability: Sensitivity Factor**



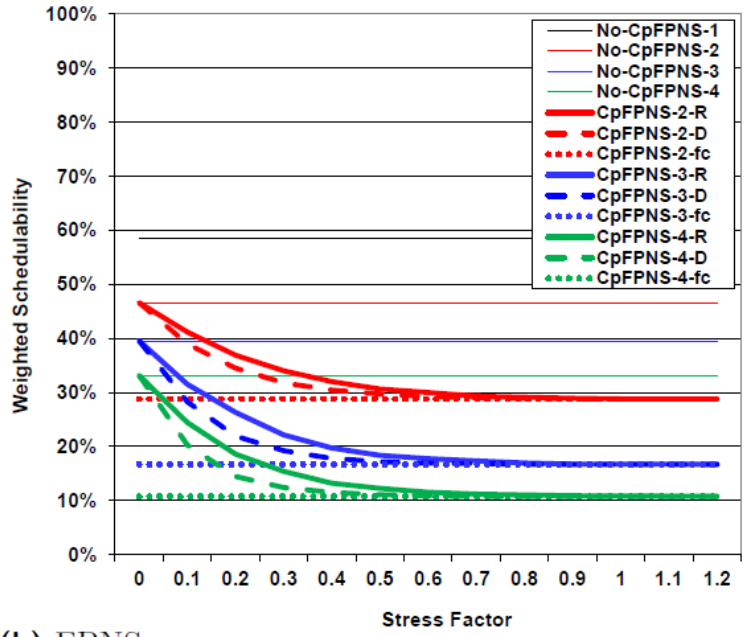
- Higher Sensitivity Factor equates to more interference and hence lower schedulability
- Broad comparison between the different tests is similar to the success ratio graphs

Evaluation

- **Weighted schedulability: Stress Factor**



(a) FPPS



(b) FPNS

- Higher Stress Factor equates to more interference and hence lower schedulability
- As the Stress Factor exceeds 1, total resource stress $E_i^r(R_i, y)$ nearly always exceeds the total resource sensitivity $S_i^r(R_i, x)$ reducing all tests that consider contention to the same level as the context independent tests

Conclusions

- **Main contribution:** Multi-core Resource Stress and Sensitivity (MRSS) task model and its accompanying schedulability analyses
- **The MRSS task model**:
  - Characterizes how much each task stresses shared hardware resources and how much it is sensitive to such resource stress
  - Provides an effective interface between timing analysis and schedulability analysis, retaining the advantages of the traditional two-step approach to timing verification
  - Caters in a generic and versatile way for a variety of different shared hardware resources
- **Schedulability analyses**:
  - Provide efficient context-dependent and context independent schedulability tests for both fixed priority pre-emptive and non-pre-emptive scheduling
  - Exhibit dominance relationships illustrating the trade-off between context independence and schedulability
  - Proven compatible or incompatible with efficient optimal priority assignment algorithms
  - Subject to systematic evaluation illustrating their effectiveness across a wide range parameter values

Preliminary industrial case study

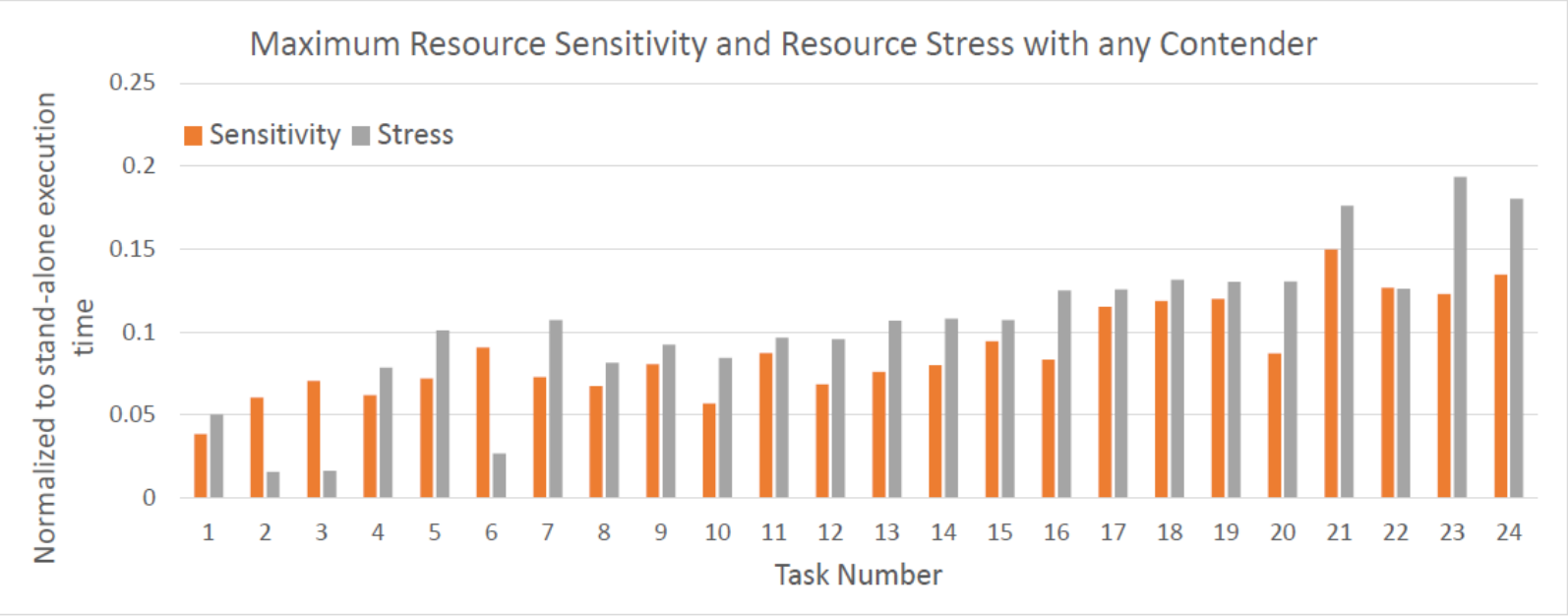
- **Purpose validation of the model**
  - Obtain resource stress and resource sensitivity data for tasks from an industrial application to act as a proof-of-concept for the MRSS task model
  - NOT intended to provide definitive values (doing so is a challenging research problem)
- **Application**:
  - 24 tasks from a Rolls-Royce aero-engine controller (object code 300 Kbytes to 40 Mbytes)
  - Developed in SPARK-Ada and verified according to DO-178C standards (level A)
  - Originally designed to run on a specific packaged processor, ported to run on Raspberry PI
- **Hardware and OS**:
  - Raspberry PI 3B+: Broadcom BCM2837 System-on-Chip with a quad-core ARM Cortex-A53 processor (16 KByte L1 data cache, 16 KByte L1 instruction cache, 512 KByte L2 shared cache, and 1 GByte of DDR2-DRAM)
  - Raspberry Pi OS Lite and the Linux Kernel 5.10.11-v7+
  - Configured to run at 600MHz to avoid any thermal throttling
  - Focus was on main memory (DDR2-DRAM) as the shared hardware resource, since the L2 cache was not used by the CPUs

Case study experimental setup

- **Configuration**
  - Examined 10,000 *traces* for each of the 24 tasks
  - Each *trace* is a job of a task with random values chosen for its input parameters controlled via a random seed for repeatability
  - Each run was repeated 9 times to ensure consistency and to enable anomalies caused by the kernel scheduler tick and clock synchronization interrupt to be eliminated
  - Each experiment paired a task with either a single contender or with a single different task
- **Contenders**
  - Three synthetic contenders were used that made Read-Read, Read-Write, or Write-Write accesses to main memory
  - 100 accesses were done in the body of each contender loop to ensure that the loop overhead was small compared to the resource contention
  - Each Read or Write access compiled down to a single instruction
  - A handshaking protocol was used to ensure that the contender always started before the task and finished after it

Case study results

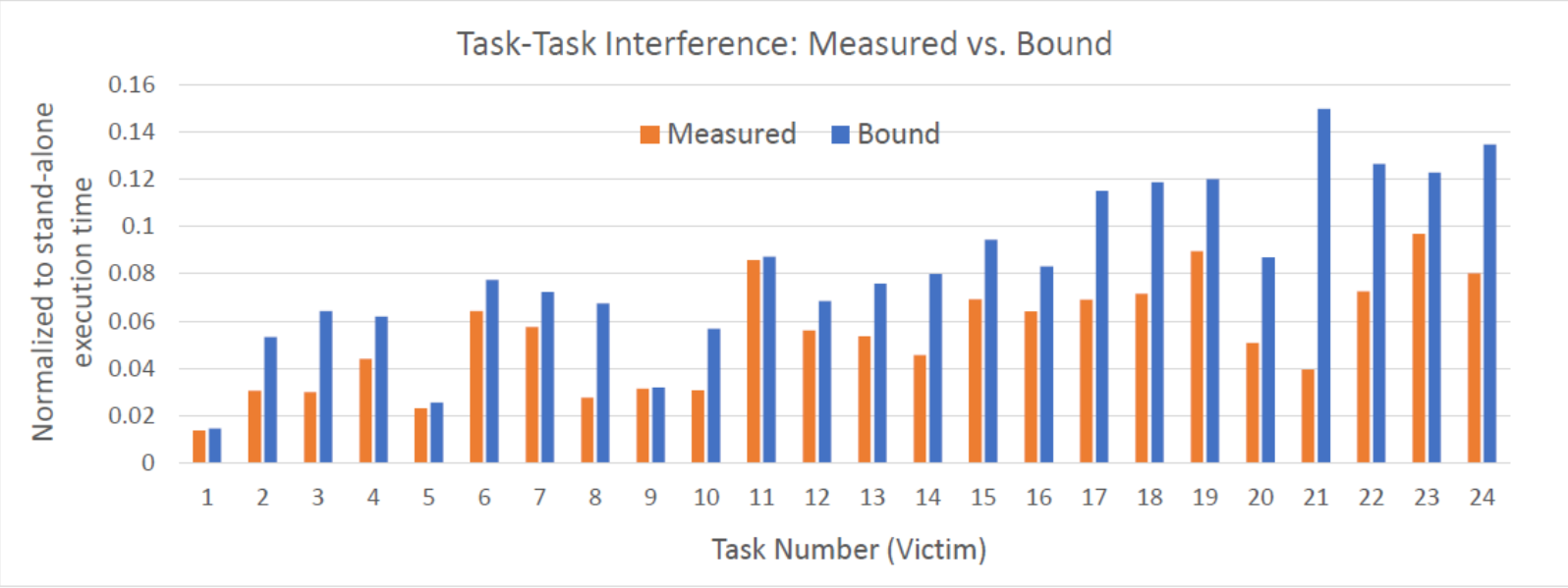
- **Experiment A.1: Task vs. Resource Contender**
  - For each task the maximum values for stand-alone execution time, resource sensitivity, and resource stress were obtained over the 10.000 traces and each of three contenders



  - Resource sensitivity varied from 3.8% to 15% of the task's stand-alone execution time
  - Resource stress varied from 1.5% to 19.3% of the task's stand-alone execution time
  - Ratio of task resource stress to sensitivity varied from 0.23 to 1.58

Case study results

- **Experiment A.2: Task vs. Task**
  - For each task pairing the maximum interference (increase in stand-alone execution time), was obtained over 10,000 traces and compared to the bound computed from resource stress and sensitivity values obtained in Experiment A.1

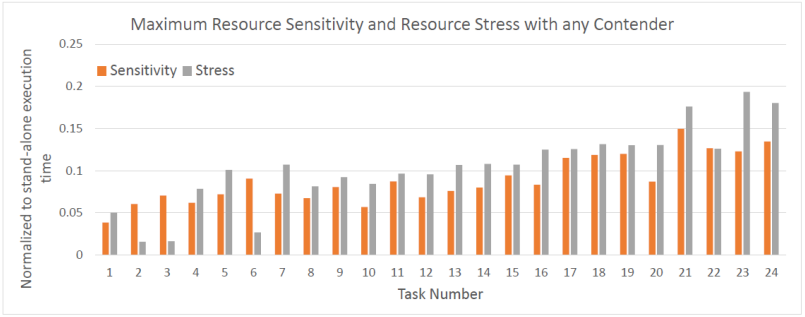


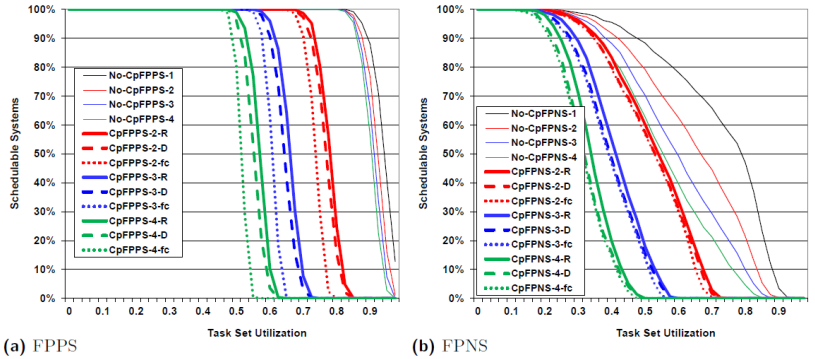
  - The maximum measured increase in execution time was no greater than the computed bound. On average it was 69% of the bound, and varied between 26% and 99%.

THE UNIVERSITY of York

RTS York

Questions?





(a) FPPS (b) FPNS

(rob.davis@york.ac.uk)