Mixed Criticality Systems with Varying Context Switch Costs

Robert I. Davis[1], Sebastian Altmeyer [2], Alan Burns[1]

[1]*Real-Time Systems Research Group, University of York, UK*
[2]*University of Amsterdam (UvA), Amsterdam, Netherlands*

Mixed Criticality Systems

- ## Mixed Criticality System (MCS)
    - A system comprising two or more applications with different criticality levels
    - Most complex embedded real-time systems are evolving into MCS for reasons of size, weight, power consumption and cost

- ## Key research question
    - How to reconcile the conflicting requirements of *separation* for assurance and *sharing* for efficient resource usage?
    - Most research has looked at resource sharing
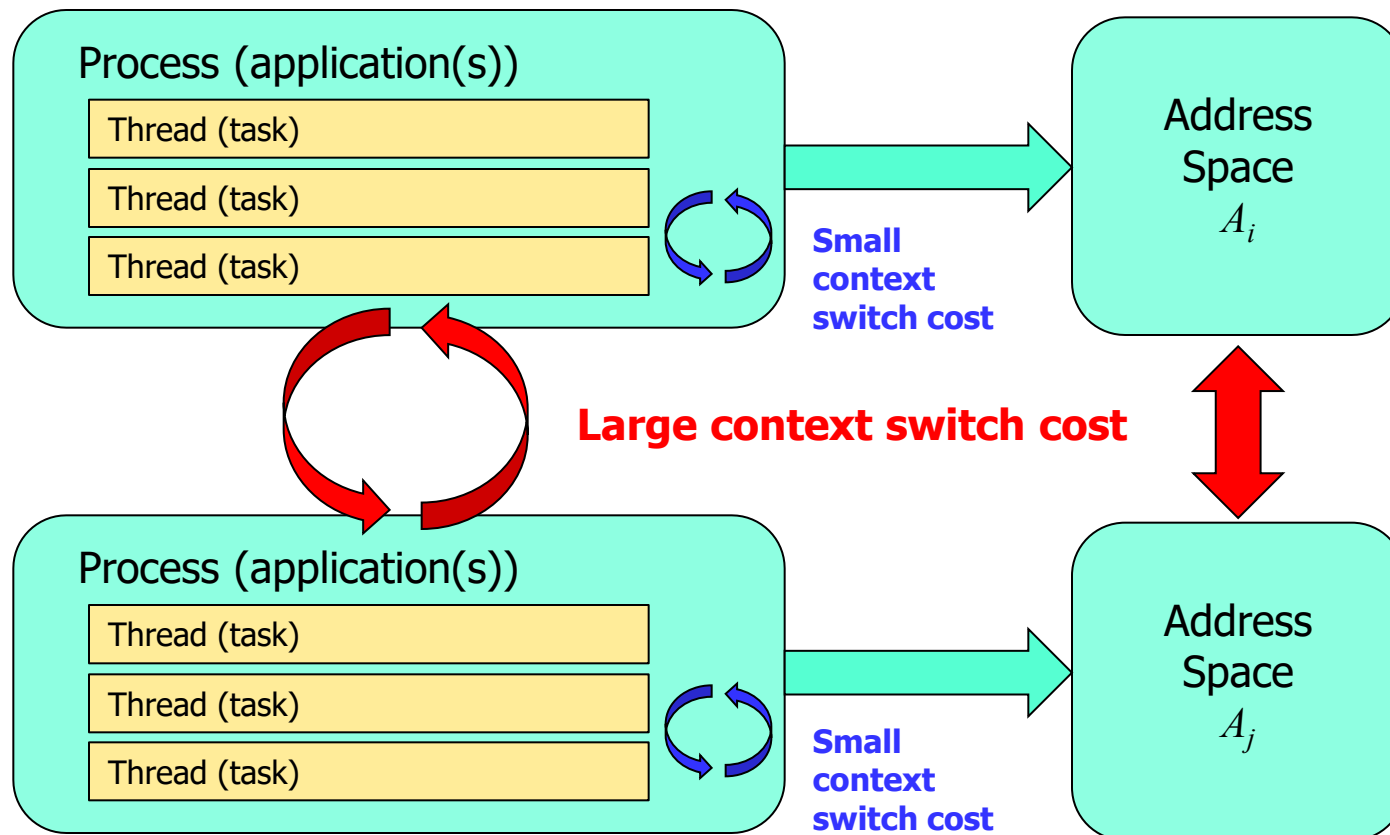   - In this work we consider some issues arising from separation

MCS and Separation

- ## Separation is vitally important
  - Safety standards (IEC61508, DO-178C, ISO26262) require that either all applications are developed to the standard required for the highest criticality application, or that independence between different applications is demonstrated in both spatial and temporal domains

- ## Memory Address Spaces
  - In the spatial domain, the memory address space(s) used by HI-criticality applications must be inaccessible to LO-criticality applications

- ## Process and Thread model
  - Each process has a separate address space
  - Threads within a process share the same address space
  - Achieve separation by mapping HI-criticality tasks to threads in one process and LO-criticality tasks to threads in another process
  - Alternatively, map all tasks from a given application to a distinct process (one process per application)
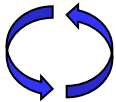
Processes and Threads

Process (application(s))

Thread (task)

Thread (task)

Thread (task)

Small context switch cost

Address Space A_i

Large context switch cost

Process (application(s))

Thread (task)

Thread (task)

Thread (task)

Small context switch cost
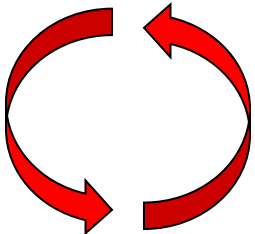
Address Space A_j

Context Switch Costs

- ### Switching threads within the same process
  - This is the context switch between tasks of the same application (or same criticality)
  - Low cost – switch only the resources unique to threads
    e.g. processor state (program counter, stack pointer etc.)
    typically have hardware support for this

- ### Switching between processes
  - This is the context switch between tasks of different applications (or different criticality)
  - High cost – switch all resources related to the process
    e.g. in addition to thread-level context switch, also switching memory address space, and since cache contents may not be valid, potentially there are additional cache operations. As the memory mapping changes it may be that some Translation Lookaside Buffer (TLB) entries are invalidated – TLB may need to be completely or partly flushed

Example System & motivation

- ## Requirement
  - Isolate the cache usage of different applications
- ## Hardware Configuration
  - Assumes explicit cache management saving and restoring cache state on process-level context switches (Whitham et al. [52])
  - Tasks belonging to the same process are allocated distinct cache partitions
- ## Properties
  - Partitioning means that thread-level context switches cause no Cache-Related Pre-emption Delays (CRPD)
  - Process-level context switches save and restore the cache state, so cache contents are unchanged when a task resumes
  - Only impact that a LO-criticality task in one process can have on a HI-criticality task in another process is via its execution time budget, which is strictly enforced by the RTOS (temporal separation)
  - Avoids security hazards – a low security process cannot use the cache as a side channel to obtain information about other high security processes

Key point: Two very different context switch costs

This work

- Provides schedulability analysis for MCS with two different context switch costs (process-level and thread-level)

- Three scheduling policies considered
  - Fixed Priority Pre-emptive Scheduling (FPPS)
  - Static Mixed Criticality (SMC)
  - Adaptive Mixed Criticality (AMC)

- Three flavours of analysis provided for each policy
  - Simple
  - Refined
  - Multiset

System Model

- Uniprocessor
- Fixed priority pre-emptive scheduling (FPPS, SMC, AMC)
- Sporadic tasks (Vestal's model for MCS)
- Each task τ_i
  - T_i – Period or minimum inter-arrival time (sporadic behaviour)
  - D_i – Constrained relative deadline
  - L_i – Criticality level (LO or HI)
  - HI-criticality tasks have both $C_i(HI)$ and $C_i(LO)$ worst-case execution time estimates with $C_i(HI) > C_i(LO)$
  - LO-criticality tasks need only have $C_i(LO)$
- Additionally
  - Each task is mapped to an address space A_i (and process)
  - When one task τ_i pre-empts another task τ_j
    same address space $(A_i = A_j)$ implies a small context switch cost C^S
    change in address space $(A_i \neq A_j)$ implies a large context switch cost C^C
    (Here costs are for switching from and later back to the pre-empted task)

Response Time Analysis for FPPS: (Very) Simple Analysis

- ## Method
  - Use large context switch cost C^C for every pre-emption
  - Equivalent to subsuming context switch times into WCET bounds
  - Response time for task τ_i

$$R_i = C_i + C^C + \sum_{\forall j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + C^C)$$

  - Fixed point iteration (converges or ends when value exceeds D_i)

Response Time Analysis for FPPS: Simple Analysis

- Example:
 - Three tasks with parameters $(C_i, D_i, T_i, L_i, A_i)$

$$\tau_A = (10, 50, 100, LO, A^L)$$
$$\tau_B = (10, 100, 200, HI, A^H)$$
$$\tau_C = (200, 265, 300, LO, A^L)$$

  - Further $C^C = 5$ and $C^S = 0$
  - Deadline Monotonic Priority Order (DMPO) is optimal
  - With priority order $\{A, B, C\}$ then $R_C = 280$ hence task set is not schedulable
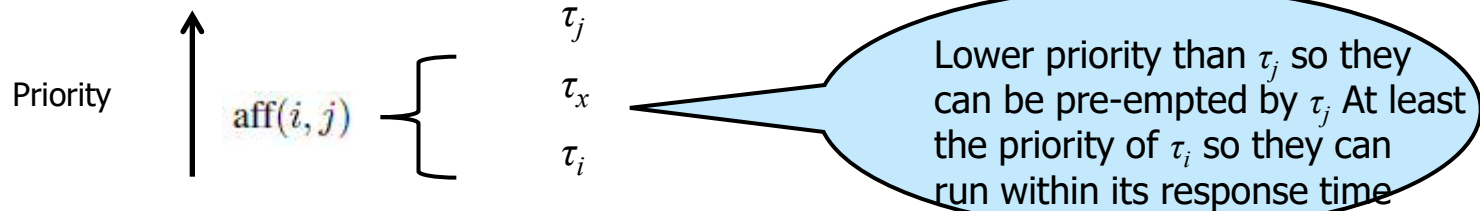
  - Part of schedule illustrating context switch costs

τ_A

τ_B

τ_C

HI-criticality execution

Large context switch cost (every time)

LO-criticality execution
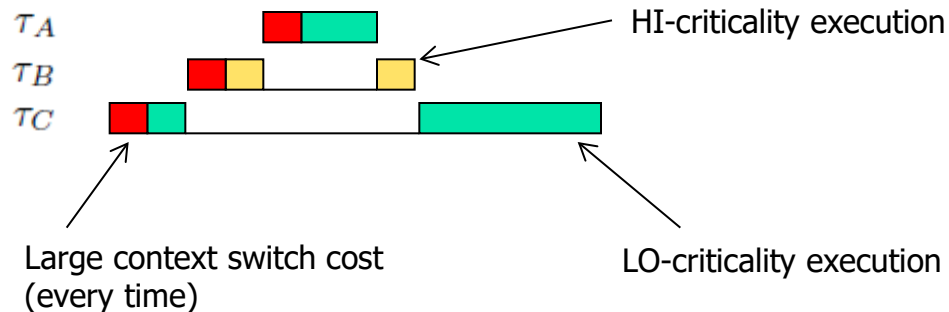
Response Time Analysis for FPPS: Refined Analysis

- ## Method

    - Consider the set of tasks $\mathrm{aff}(i,j) = \mathrm{hep}(i) \cap \mathrm{lp}(j)$ that can be *affected* by pre-emption by task τ_j during the response time of task τ_i

    Priority

    $$\mathrm{aff}(i,j) \left\{ \begin{array}{l} \tau_j \\ \tau_x \\ \tau_i \end{array} \right.$$

    Lower priority than τ_j so they can be pre-empted by τ_j At least the priority of τ_i so they can run within its response time

    - Only get a large context switch cost for pre-emption by task τ_j if there is some task τ_h that can be pre-empted by task τ_j during the response time of task τ_i that belongs to a different process and hence different address space

    $$\gamma_{i,j} = \left\{ \begin{array}{ll} C^C & \text{if } \exists h \in \mathrm{aff}(i,j) | A_h \neq A_j \\ C^S & \text{otherwise} \end{array} \right.$$

    $$R_i = C_i + C^C + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + \gamma_{i,j})$$

11

Response Time Analysis for FPPS: Refined Analysis

- Example:
  - Three tasks with parameters $(C_i, D_i, T_i, L_i, A_i)$

$$\tau_A = (10, 50, 100, LO, A^L)$$
$$\tau_B = (10, 100, 200, HI, A^H)$$
$$\tau_C = (200, 265, 300, LO, A^L)$$

  - Further $C^C = 5$ and $C^S = 0$
  - Deadline Monotonic Priority Order (DMPO) is **not** optimal
  - With priority order $\{A, B, C\}$ then $R_C = 280$ hence task set is not schedulable
  - With priority order $\{B, A, C\}$ then $R_C = 265$ and task set is schedulable

  - Part of schedule illustrating context switch costs

Shared process and address space implies small context switch cost

  - Audsley's Optimal Priority Assignment algorithm is not applicable, since response time depends on priority order of higher priority tasks

12

Response Time Analysis for FPPS: Multiset Analysis

- ## Method
  - Accounts for the number of times that tasks with intermediate priorities may be pre-empted by task τ_j during the response time of task τ_i
  - Avoids over-counting the number of large context switches that are possible
- ## Avoids pessimism
  - Example again: DMPO $\{A, B, C\}$,

$$\tau_A = (10, 50, 100, LO, A^L)$$
$$\tau_B = (10, 100, 200, HI, A^H)$$
$$\tau_C = (200, 265, 300, LO, A^L)$$

  - τ_A can only pre-empt τ_B once each time τ_B executes, since $R_B = 30$ and $T_A = 100$ so we can only get two (not three) large context switches due to pre-emptions by τ_A during the response time of τ_C

Response Time Analysis for FPPS: Multiset Analysis

How many jobs of τ_k can run during the response time of τ_i
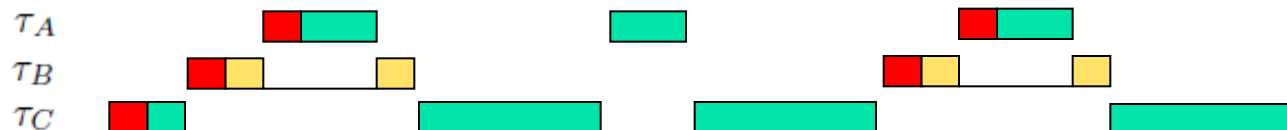
- Method

  - (Let $E_j(R_k) = \lceil \frac{R_k}{T_j} \rceil$ denote the maximum number of times that a task τ_j can execute during the response time of some lower priority task τ_k)

  How many times τ_j can pre-empt each job of τ_k

  - Account for the fact that task τ_j can pre-empt each intermediate task τ_k a maximum of $E_j(R_k)E_k(R_i)$ times during the response time of task τ_i

  - Form a multiset with $E_j(R_k)E_k(R_i)$ copies of the context switch time for task τ_j pre-empting task $\tau_k | k \in \mathrm{aff}(i,j)$

  Multiset contains costs for all possible direct pre-emptions by τ_j

$$M_{i,j} = \bigcup_{k \in \mathrm{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} \left\{ \begin{array}{ll} C^C & \text{if } A_k \neq A_j \\ C^S & \text{otherwise} \end{array} \right\} \right)$$

  - From the multiset, obtain an upper bound on the total context switch costs caused by the maximum number $E_j(R_i)$ of pre-emptions by task τ_j that can occur within the response time of task τ_i

$$\gamma_{i,j}^M = \sum_{q=1}^{E_j(R_i)} F(q, M_{i,j})$$

  where $F(q, M_{i,j})$ returns the q-th largest value from the multiset

14

Response Time Analysis for FPPS: Multiset Analysis

- ## Method (continued)
  - Include the term for *all* context switches due to task τ_j in the response time analysis

  $$R_i = C_i + C^C + \sum_{\forall j \in \mathbf{hp}(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j + \gamma_{i,j}^M \right)$$

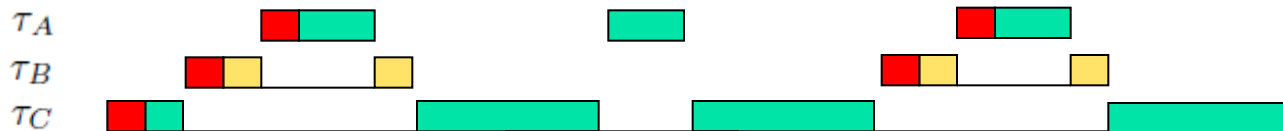- ## Example
  - DMPO $\{A, B, C\}$.

  $$\tau_A = (10, 50, 100, LO, A^L)$$
  $$\tau_B = (10, 100, 200, HI, A^H)$$
  $$\tau_C = (200, 265, 300, LO, A^L)$$

  - Multiset $M_{C,A}$ contains the value C^C twice since $E_A(R_B) = 1$ and $E_B(R_C) = 2$ and the value C^S three times since $E_A(R_C) = 3$
  - The three largest values then give the overall context switch cost due to pre-emptions by τ_A leading to $R_C = 275$ rather than $R_C = 280$
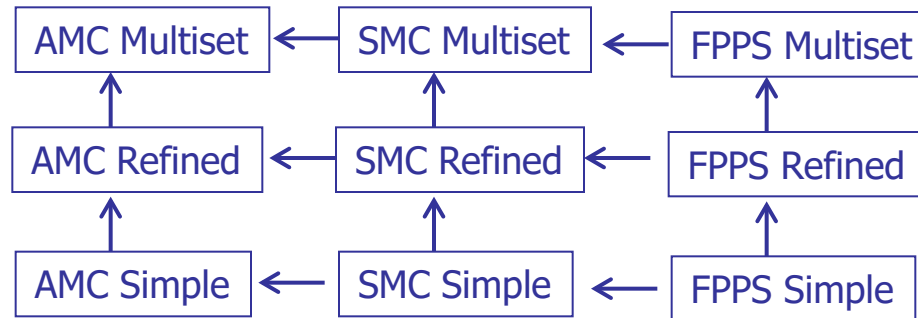
Analysis for SMC and AMC

- ## SMC and AMC
  - Scheduling policies for MCS

- ## In the paper
  - Derived simple, refined, and multiset analysis assuming two different context switch costs based on the ideas presented for FPPS

- ## Key point
  - Accounting for when LO-criticality tasks can execute and so pre-empt or be pre-empted
  - With SMC: LO-criticality tasks can still execute in HI-criticality mode, and so can pre-empt or be pre-empted in that mode leading to large context switch costs. They can also have longer response times and miss their deadlines in HI-criticality mode
  - With AMC: LO-criticality tasks cannot execute in HI-criticality mode, so large context switch costs are avoided once that mode is established

Dominance relations

AMC Multiset	←	SMC Multiset	←	FPPS Multiset
↑		↑		↑
AMC Refined	←	SMC Refined	←	FPPS Refined
↑		↑		↑
AMC Simple	←	SMC Simple	←	FPPS Simple

- ## Meaning of Dominance
  - A schedulability analysis X is said to dominate an analysis Y (denoted by X←Y) if all tasks sets that are deemed schedulable by Y are also deemed schedulable by X, and there are also task sets that are deemed schedulable by X, but not by Y

  - By construction
    AMC dominates SMC which dominates FPPS
    Multiset Analysis dominates Refined Analysis dominates Simple Analysis

Priority Assignment

- ## Optimality
    - Deadline Monotonic Priority Ordering (DMPO) is optimal for FPPS ignoring context switch costs and also with the simple analysis
    - Audsley's OPA algorithm is optimal for SMC and AMC ignoring context switch costs and also with the simple analysis
- ## Non-Optimality
    - The refined and multiset analyses for FPPS, SMC, and AMC are all incompatible with Audsley's OPA algorithm since the response time of a low priority task depends on the relative priority ordering of higher priority tasks breaking Condition #1 for OPA-compatibility (see [29]).
- ## Hints
    - For two classes of task (i.e. two processes / address spaces) empirical brute-force evaluation of all possible priority assignments on small task sets shows that if a schedulable priority assignment exists then there is often a schedulable assignment similar to DMPO with only a few tasks swapped in the ordering

Priority Assignment Heuristic

- ## Idea
  - Start with DMPO swap at most two pairs of tasks i.e. check n^2 rather than $n!$ assignments

```
Algorithm 1: PriorityAssignmentHeuristic({τ₁ ... τₙ})
 1: bool isSchedulable = false;
 2: int i = 1;
 3: while (¬ isSchedulable ∧ i < n − 1) do
 4:    {Outer loop, swaps the priority of two consecutive tasks};
 5:    swapPriority(i, i + 1);
 6:    isSchedulable = checkSchedulability();
 7:    if (isSchedulable) then
 8:        break;
 9:    end if
10:    int j = i;
11:    while (¬ isSchedulable ∧ j < n − 1) do
12:        {Inner loop, swaps the priority of two consecutive tasks};
13:        swapPriority(j, j + 1);
14:        isSchedulable = checkSchedulability();
15:        if (isSchedulable) then
16:            break;
17:        end if
18:        swapPriority(j + 1, j);
19:        j = j + 1;
20:    end while
21:    {If not successful, roll back};
22:    swapPriority(i + 1, i);
23:    i = i + 1;
24: end while
25: return isSchedulable;
```

  - Much shorter runtime than exhaustive approach with reasonably good results

Context Switch Costs

- Hardware configuration
    - Explicit cache management approach of Whitham et al. [52]
    - RTOS initiates cache save/restore which is done in hardware
    - Cache Budget Register (CBR) records number of lines to save/restore
    - Save/Restore Stack (SRS) holds tags for cache lines used by pre-empted tasks
    - Prototype FPGA implementation

    - Context switch $30\mu s$ (no save/restore)
    - Context switch $600\mu s$ (assuming 64Kbyte data and instruction caches)
    - Values used in evaluation
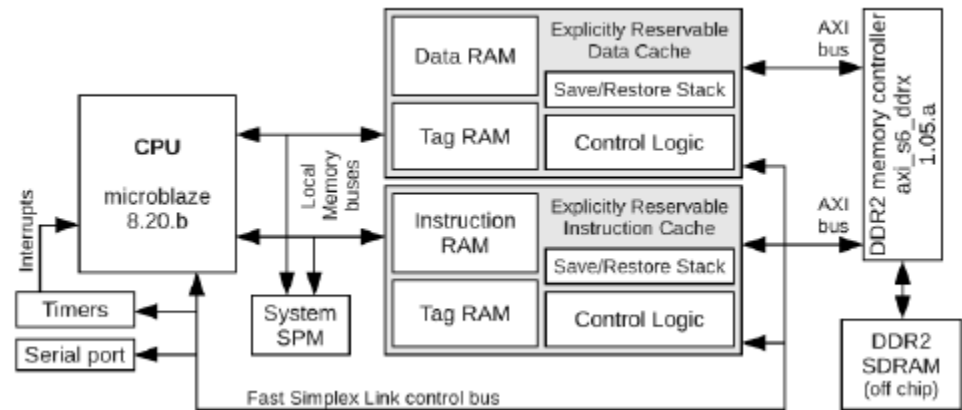
Evaluation

- Generated synthetic task sets
  - Number of tasks (Default $n = 10$)
  - Periods: Log-uniform distribution (Default $10ms - 1s$)
  - Deadlines: Implicit
  - Utilisation values U_i generated using Uunifast
  - LO-criticality execution times set via $C_i(LO) = U_i T_i$
  - HI-criticality execution times $C(HI) = CF. C(LO)$ where CF is the criticality factor (Default $CF = 2.0$)
  - Probability CP of a task being HI-criticality (Default $CP = 0.5$)
  - All LO-criticality tasks mapped to a single process and address space
  - Similarly all HI-criticality tasks mapped to a single process and address space which is distinct from that for the LO-criticality tasks
  - Context switch costs $C^S = 30\mu s$ and $C^C = 600\mu s$
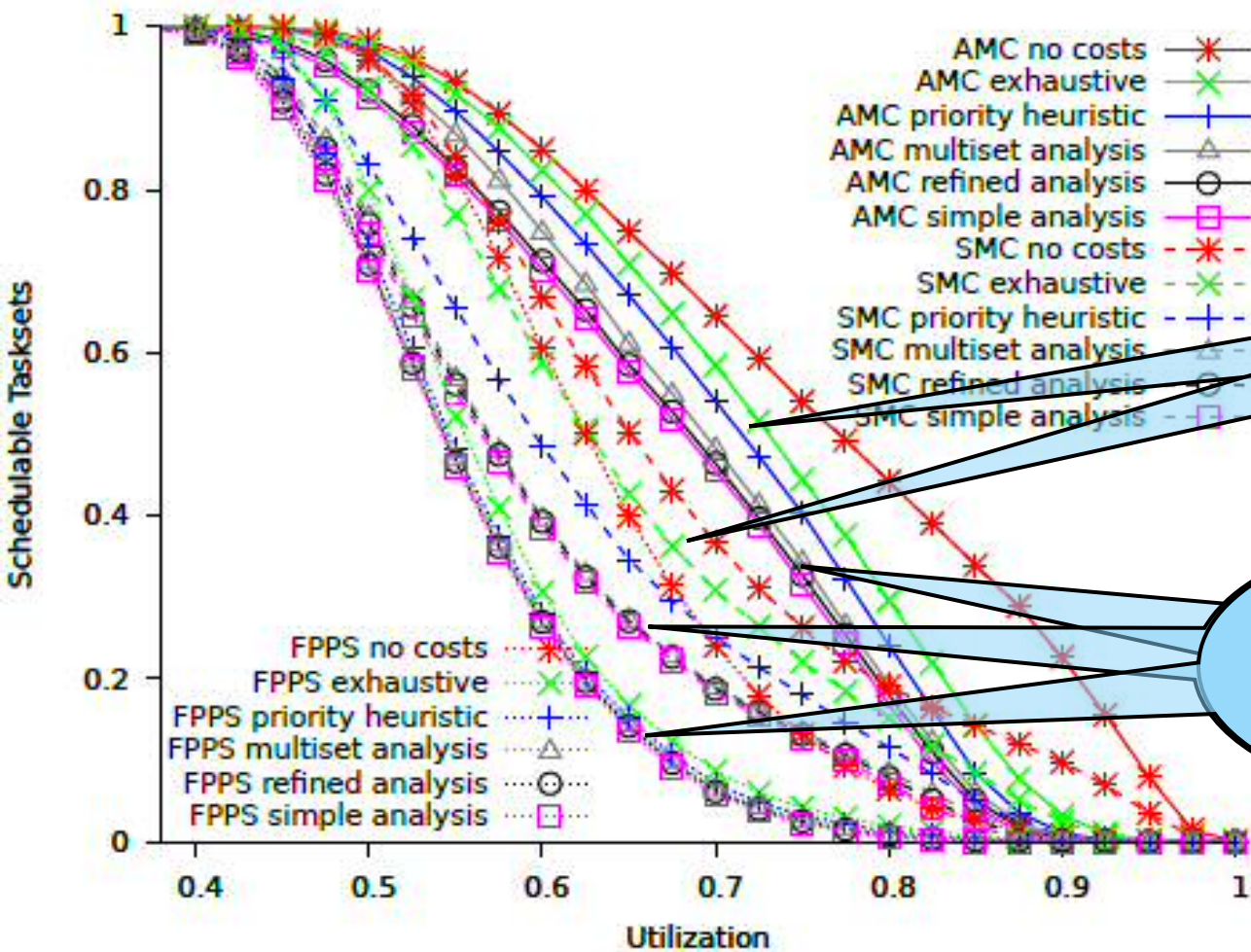 - All values integer units of μs

Evaluation

- Compared the following schemes:
  - Scheduling policies: AMC, SMC, FPPS
  - Analyses: Simple, Refined, Multiset
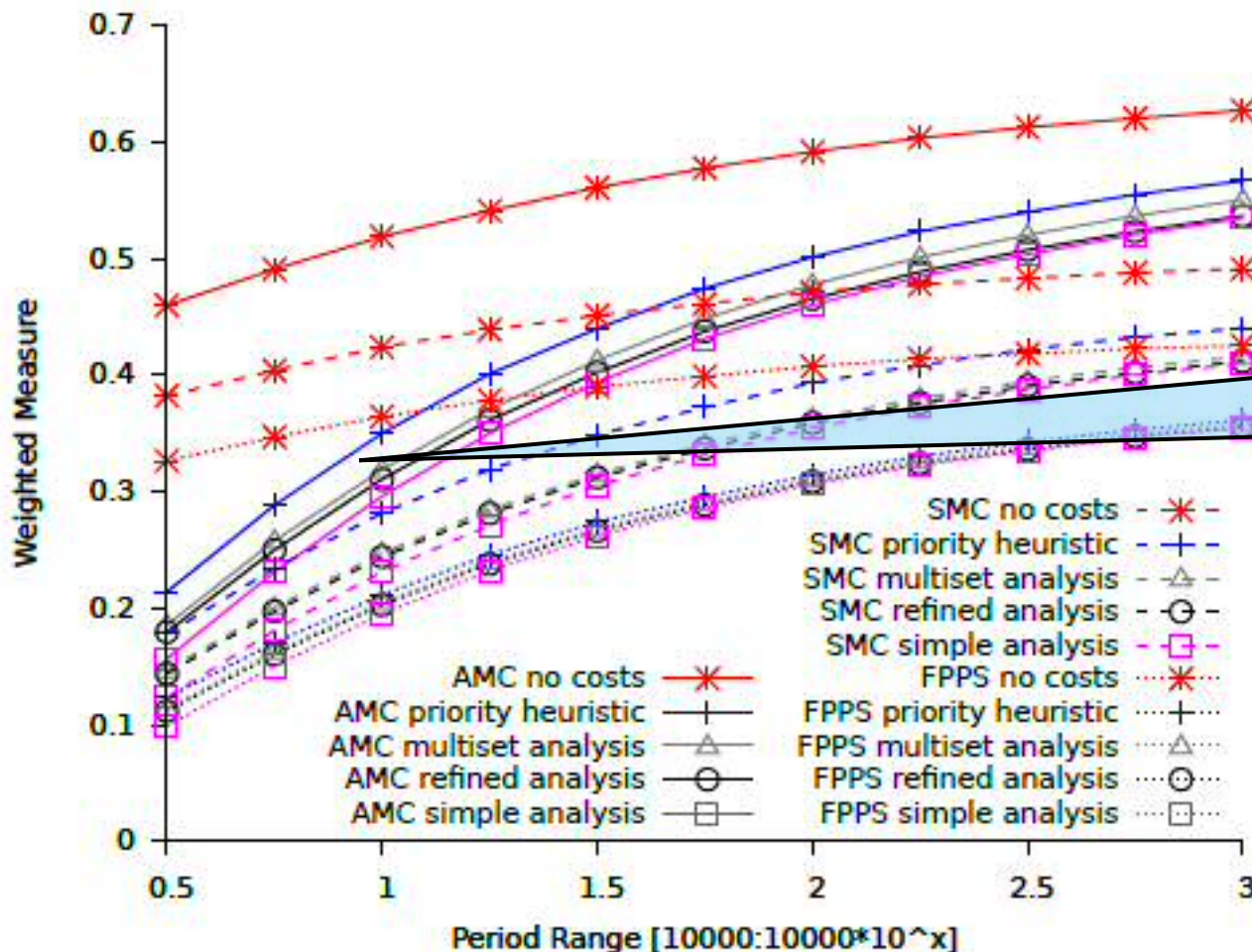 - Priority assignment policies: DMPO, heuristic (two swaps), exhaustive

Success ratio

# Weighted schedulability: varying range of task periods

Varying range of task periods from $10^{0.5} \approx 3$ to $10^3 = 1,000$

Small range of periods relative performance of priority heuristic improves. More scope to re-arrange tasks with similar deadlines

Smaller range of periods schedulability gets worse as small periods imply proportionately larger context switch costs

Weighted schedulability: varying large context switch cost
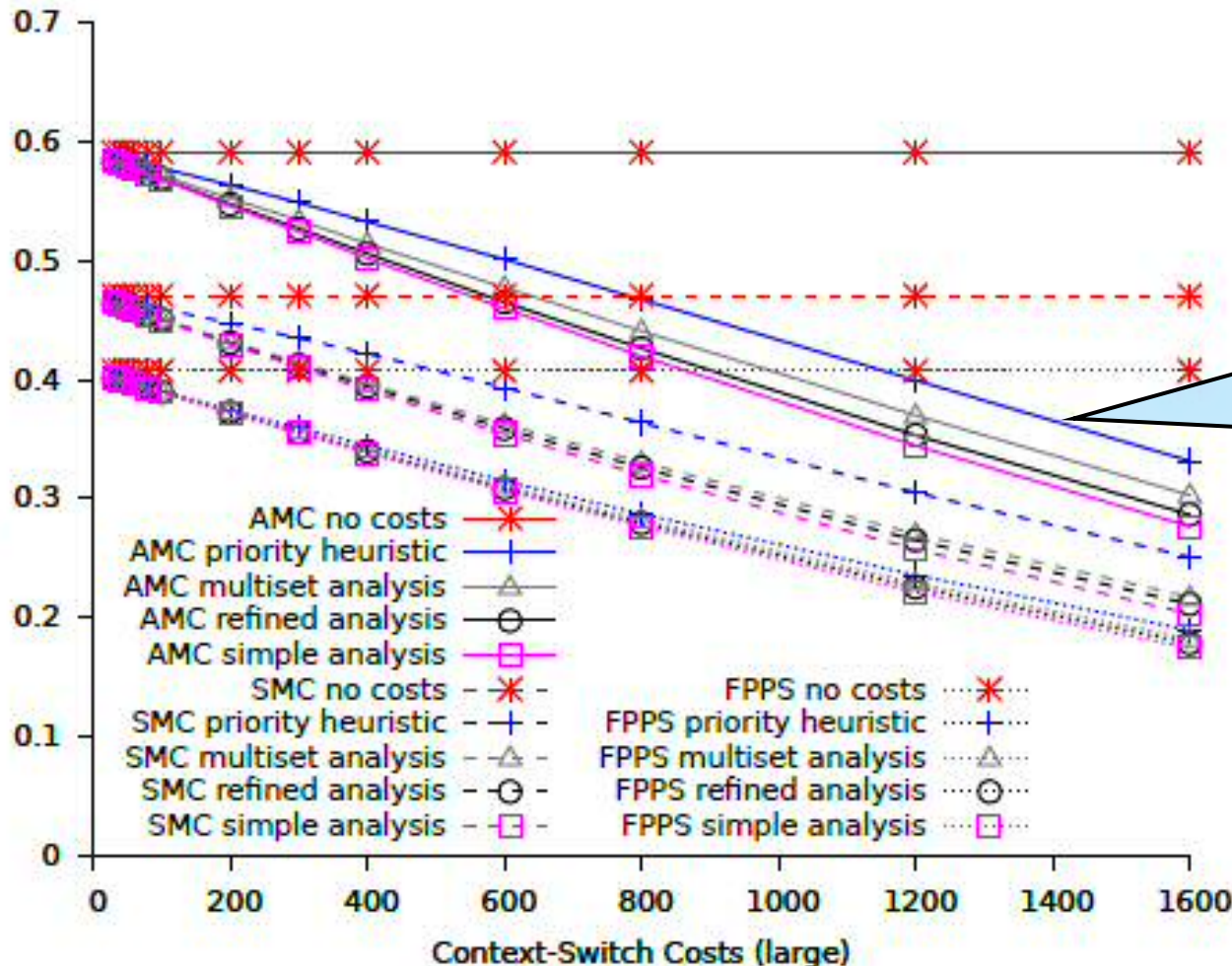
Varying large context switch cost from 0 to 1600μs

With larger costs lower schedulability, and better relative performance using multiset analysis and re-arranging priorities

Conclusions

- Summary
    - Considered a general model (arbitrary groupings of tasks to processes) that assumes process-level and thread-level context switches
    - Integrated two different context switch costs into response time analysis for FPPS, SMC, and AMC scheduling policies
    - Showed that Audsley's Optimal Priority Assignment algorithm is not compatible with the more effective refined and multiset analyses
    - Priority assignment is important for this problem since it can significantly reduce context switch overheads by collecting tasks belonging to the same process together in the priority ordering
    - A simple heuristic was shown to be effective – but exhaustive exploration of priority assignment also indicates that there is more performance that could be obtained

Future work

- Open questions
    - How to assign priorities? Can we find an optimal ordering without having to exhaustively explore all possibilities?
    - One disadvantage of fully pre-emptive scheduling is the large number of context switches – so how best to schedule tasks when there are two different context switch costs (process-level and thread-level)?

Questions?