# PROARTIS for Space

Schedulability Analysis Techniques and
Tools for Cached and Multicore Processors


TEC-ED & TEC-SW Final Presentation Days - 7th December 2015

# Partners


UNIVERSITÀ DEGLI STUDI DI PADOVA


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

RAPITA SYSTEMS LTD

COBHAM

Cobham Gaisler AB

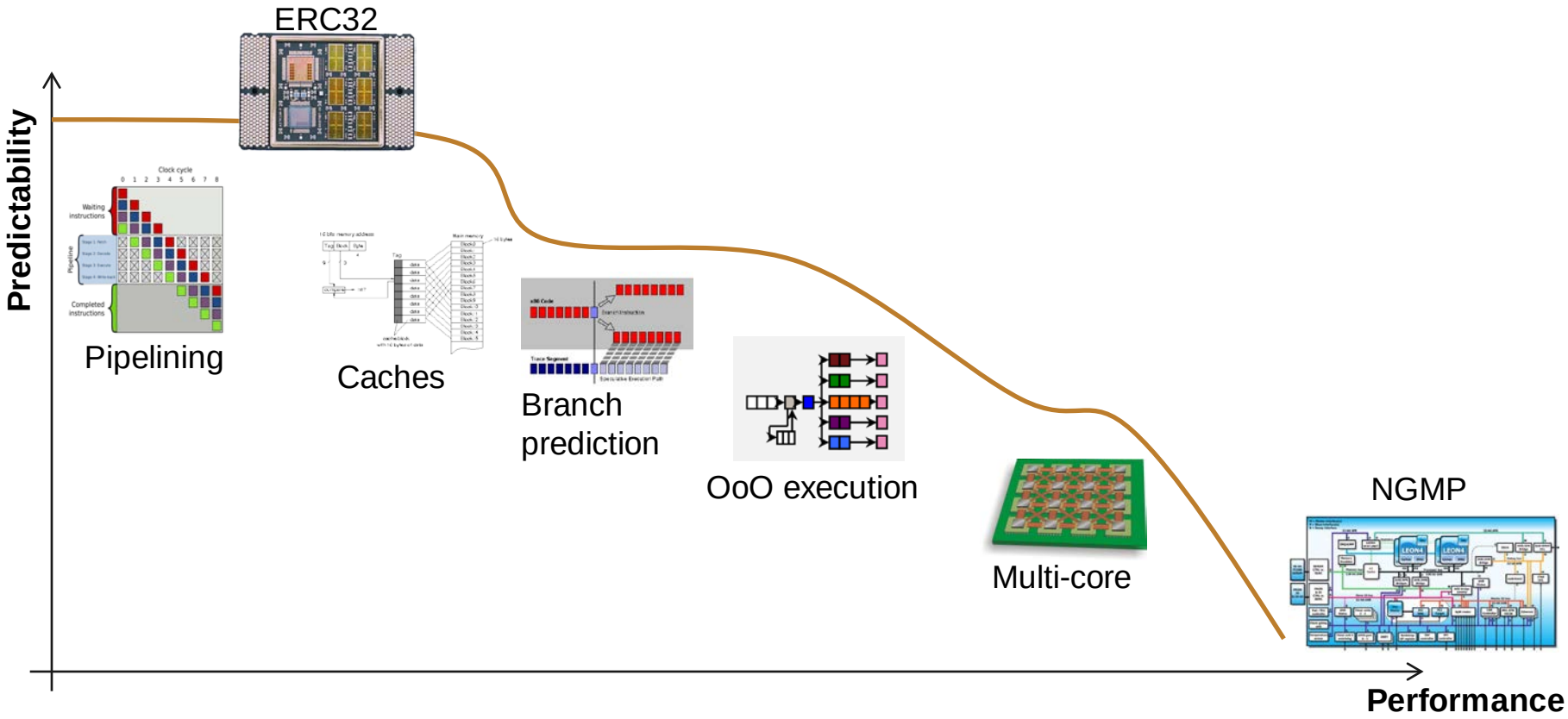
AIRBUS DEFENCE & SPACE

# PROARTIS

- Project funded by the European Commission under FP7 (2011–2013).
  - Probabilistically Analysable Real-Time Systems

- Demonstrated how using probabilistic techniques can aid in the timing analysis of complex software systems

# PROARTIS


ERC32
Predictability
Pipelining
Caches
Branch prediction
OoO execution
Multi-core
NGMP
Performance

# PROARTIS

- HW is not predictable any more. SW is more and more complex
- Intuition is that the randomness of the timing behaviour of a system can be exploited to enable new forms of timing analysis
- If the system behaviour can be characterised bottom-up as random behaviour, then probability theory can be used to predict the overall behaviour of the software and its likelihood of exceeding assigned bounds.

# PROARTIS for Space

- How to use multi-core processors (NGMP) in an effective way
- While achieving adequate levels of guarantee of the timing correctness
  - Investigate the choice of scheduling and schedulability analysis for NGMP
  - Use of software randomisation to enable probabilistic timing analysis
  - Hardware support for observing timing behaviour
  - Improve tool support for timing analysis on NGMP

# Multiprocessor Scheduling


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

# Transition to multiprocessors

- Uniprocessor's body of knowledge does not directly translate to Multiprocessors
- P4S surveyed scientific literature for specific solutions for the use of multiprocessor systems in embedded critical systems
  - Find an **efficient** and **analysable scheduling** algorithm where
    - *Efficient* : low overhead, exploits most of the processing power
    - *Analysable* : is the system schedulable?, account for overhead
  - Find an **efficient** and **analysable** way to **share resources** (e.g., memory) where
    - *Efficient* : low overhead, minimizing blocking on rivals
    - *Analysable* : accounting for blocking time and overhead

# Multiprocessor scheduling paradigms

- Global schedulers
  - PROS: idle only when there is no task to schedule (work conserving)
  - CONS: high overhead, central control bottleneck
- Partitioned schedulers
  - PROS: reuse lots of uniprocessor knowledge, limited overhead
  - CONS: NOT work conserving, task-to-processor assignment problem
- Hybrid schedulers
  - PROS: generally achieve high schedulability
  - CONS: mostly theoretical, very limited ecosystem

# Multiprocessor resource sharing

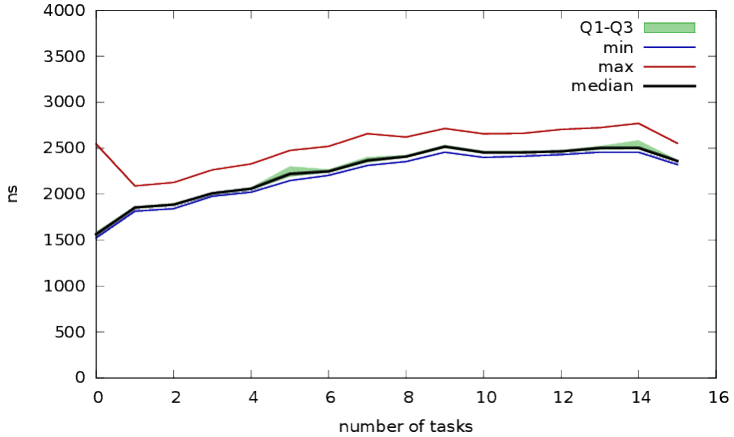
- Sharing SW resources is a necessity when collaborative tasks reside/execute on different cores
- Potential parallelism complicates the problem wrt single core settings
- Different solutions can be deployed
  - Based on non-preemptive sections (Easy implementable, short blocking, defy priority/criticality)
  - Based on priority boosting (Limited blocking, defy priority/criticality)
  - **Multiprocessor Resource Sharing Protocol - MrsP** makes a class on its own
    - Helping mechanism (migration) to release resources as soon as possible
    - Per-core immediate ceiling to preserve priority/criticality

# RTEMS-SMP MrsP

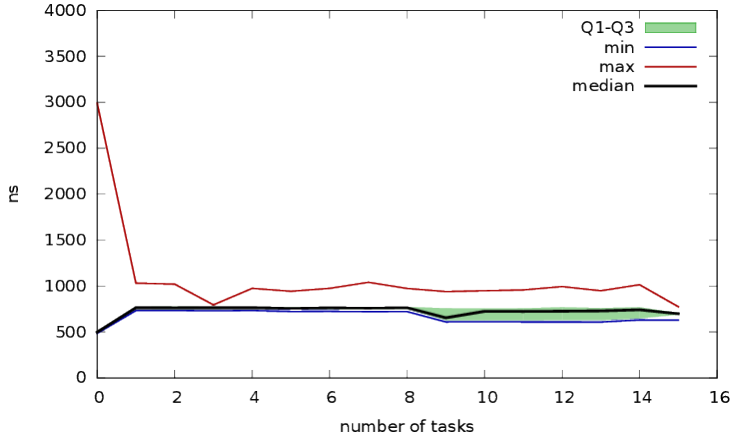
- First real implementation on a industry-ready OS
- Original implementation quite complex
  - Complex data structures, Unused hooks for future implementation of other protocols, …
- New implementation
  - Not easily extendable for other protocols
  - Smaller and time-composable data structures
  - Simpler interaction with the scheduler

# RTEMS-SMP MrsP

- Semaphore release (unlock) procedure:
  - old implementation: dependent on the pending tasks
  - new implementation: constant time



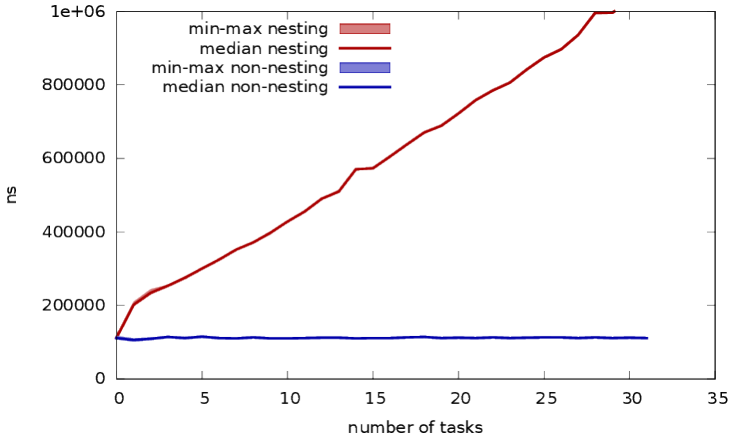

*Original version*



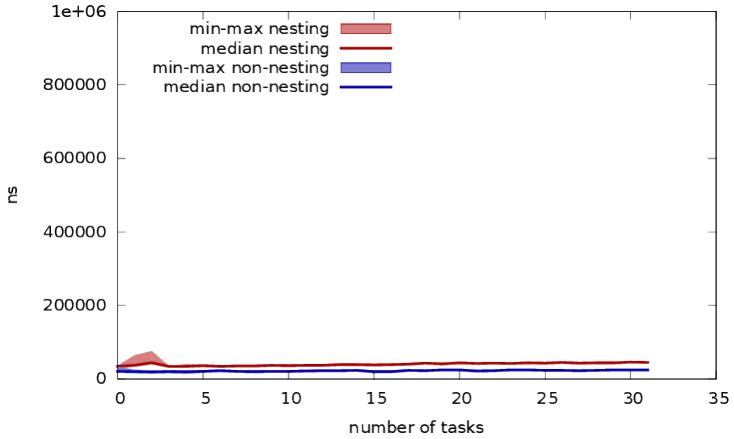

*Modified version*

# RTEMS-SMP MrsP

- Semaphore obtain (lock) procedure when nesting resources:
  - old implementation: dependent on nesting depth and pending tasks
  - new implementation: linear time, dependent only on nesting depth




*Original version*




*Modified version*

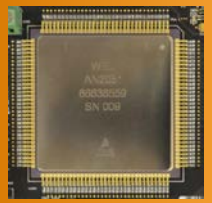
# Software Randomization

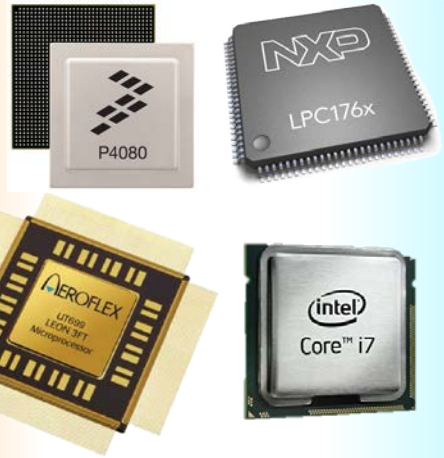
# Why Software Randomization?

***'Classic' Timing Analysis***

***Probabilistic Timing Analysis***

**VisiumCore**



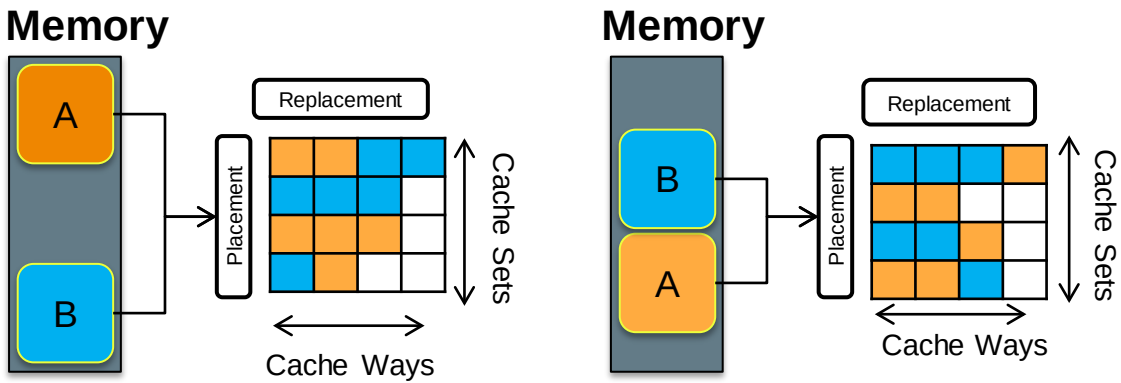



**PROARTIS/PROXIMA (FPGA)**


**Predictable** ← SW-Rand → **Random**

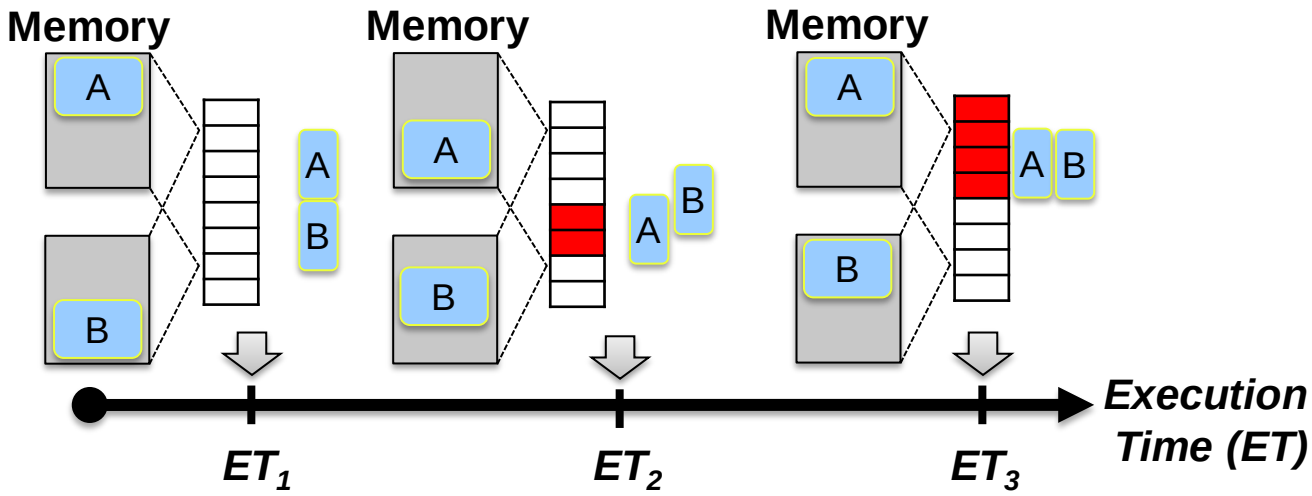
# Software-only Randomization Approach

- The location of objects in memory (memory layout) determines its location in caches (cache layout)
  - The random location of objects in memory makes conventional caches behave as a random ones
  - Random memory layouts → Random cache layouts



# When is randomization applied?

- At every **program execution** a new memory layout is **randomly** generated
  - Different memory layouts results in different execution times



# Software-only Randomization Implementation

- Randomized memory objects
  - Functions (Code Randomisation)
  - Stack frames (Stack Randomisation)

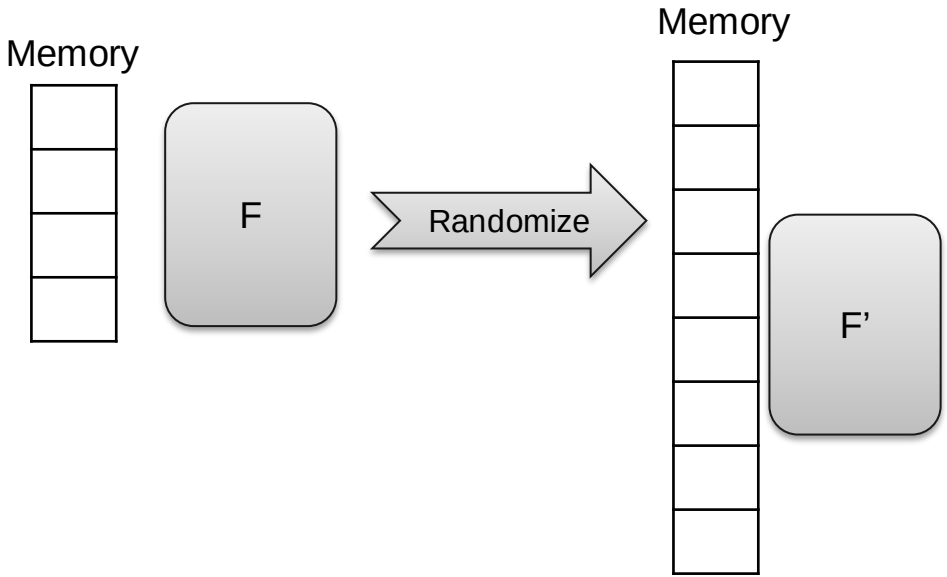
- A **compiler LLVM[1] pass**, a **run-time library** and a **memory allocator** based on TLSF[2] has been developed
  - Software framework (Stabilizer[3,4]) available at https://github.com/ccurtsinger/stabilizer

- SW-Rand is compatible with NGMP and RTEMS
  - some restrictions apply

[1] *The LLVM compiler infrastructure.* http://llvm.org/
[2] *TLSF: Dynamic storage allocation for real-time embedded systems. M. Masmano, I. Ripoll, and A. Crespo*
[3] *Probabilistic Timing Analysis on Conventional Cache Designs, DATE 2013*
[4] *Stabilizer: Statistically Sound Performance Evaluation, ASPLOS 2013*

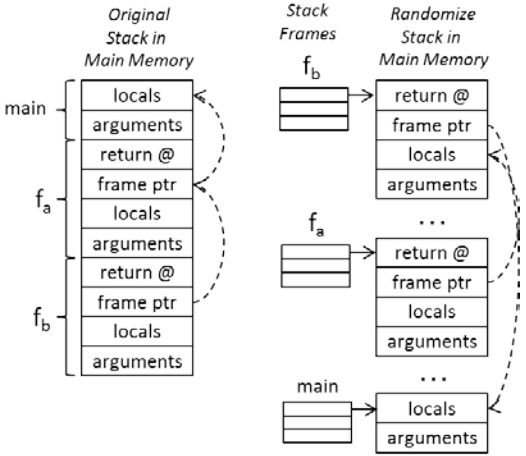
# Code Randomization

- SW-Rand randomizes code at the function granularity
- SW-Rand request enough memory to be able to allocate the function code and to hit in any cache line



# Stack Randomization

- SW-Rand randomizes the stack by making it non-contiguous
  - Each function call moves the stack to a random location
- On every call, the function loads its stack frame address from a frame table



# Tool support for NGMP


Cobham Gaisler AB

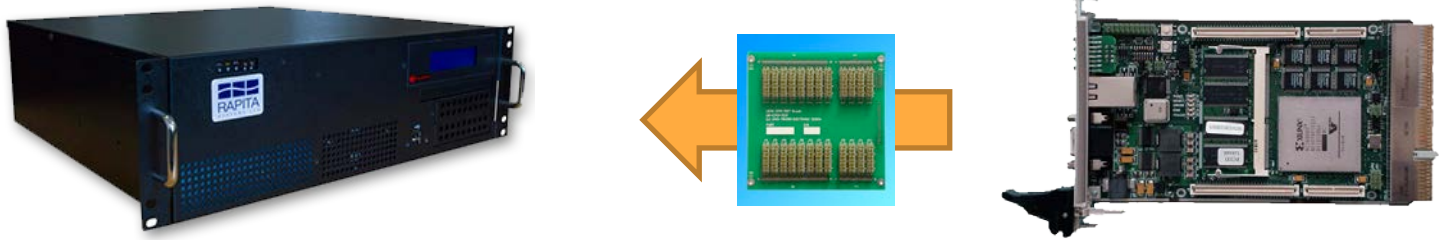
# NGMP Tool Support

- Tool support for timing analysis of NGMP
  - Quad-Core 32-bit LEON4 SPARC V8 processor
  - NGMP Tracing Capabilities + Data Logger
  - Timing analysis tools
    - RapiTime (PROARTIS/PROXIMA extension)
    - RapiTask (multi-core extension)

RVS

# Data Logger for NGMP

- Based on Rapita Systems RTBx-1221
- Capture execution trace via GPIO




- Each core write to 8 pins of GPIO via special opcode
- Recorded all at once as 32 bit data by RTBx
  - Some post processing is required

# Execution Trace

- Load/Store to alternate address space identifier (ASI) 0x80 and above lead to propagation of part of instructions to the trace output (8 bits per core).

Format (3):

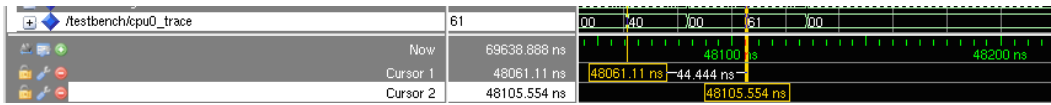
| 11 | rd | op3 | rs1 | i=0 | asi | rs2 |
|---|---|---|---|---|---|---|
| 31 | 29 | 24 | 18 | 13 | 12 | 4 0 |

| 11 | rd | op3 | rs1 | i=1 | simm13 |
|---|---|---|---|---|---|
| 31 | 29 | 24 | 18 | 13 | 12 0 |

Trace output = inst(7:0) when:

| | |
|---|---|
| rd | = don't care |
| op3 | = LD*A/ST*A |
| rs1 | = don't care |
| i | = 0 |
| asi(7) | = inst(12) = 1 |

```
#     36261.11 ns : cpu0: 0xc0000038    nop
#     36283.332 ns : cpu0: 0xc000003c    stba  %g0, [0] 0x41
#     36305.554 ns : cpu0: 0xc0000040    nop
#     36327.776 ns : cpu0: 0xc0000044    nop
#        36350 ns : cpu0: 0xc0000048    or   %g2, %g2, %g0
#     36394.444 ns : cpu0: 0xc000004c    nop
#     36416.666 ns : cpu0: 0xc0000050    nop
#      48061.11 ns : cpu0: 0xc0000054    stba  %g0, [0] 0x42
#     48105.554 ns : cpu0: 0xc0000058    stba  %g0, [%g1] 0x43
#     48127.776 ns : cpu0: 0xc000005c    nop
#        48150 ns : cpu0: 0xc0000060    nop
#     48172.222 ns : cpu0: 0xc0000064    mov  %asr17, %g3  [0x00000587]
#     48194.444 ns : cpu0: 0xc0000068    and  %g3, 0x001f, %g4  [0x00000007]
#     48238.888 ns : cpu0: 0xc000006c    clr  %g4  [0x00000000]
```

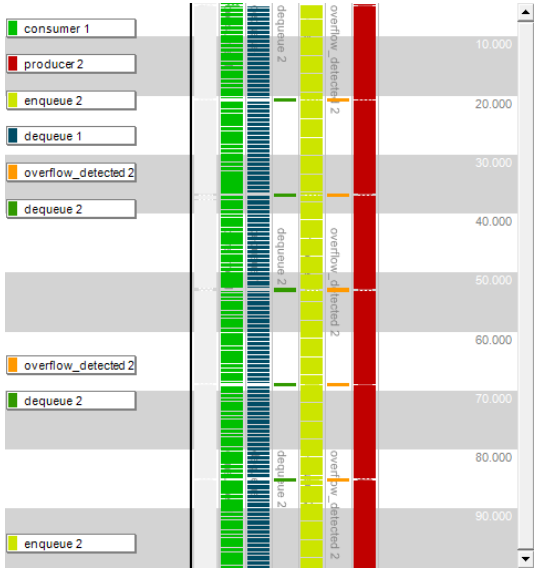


# Execution Trace

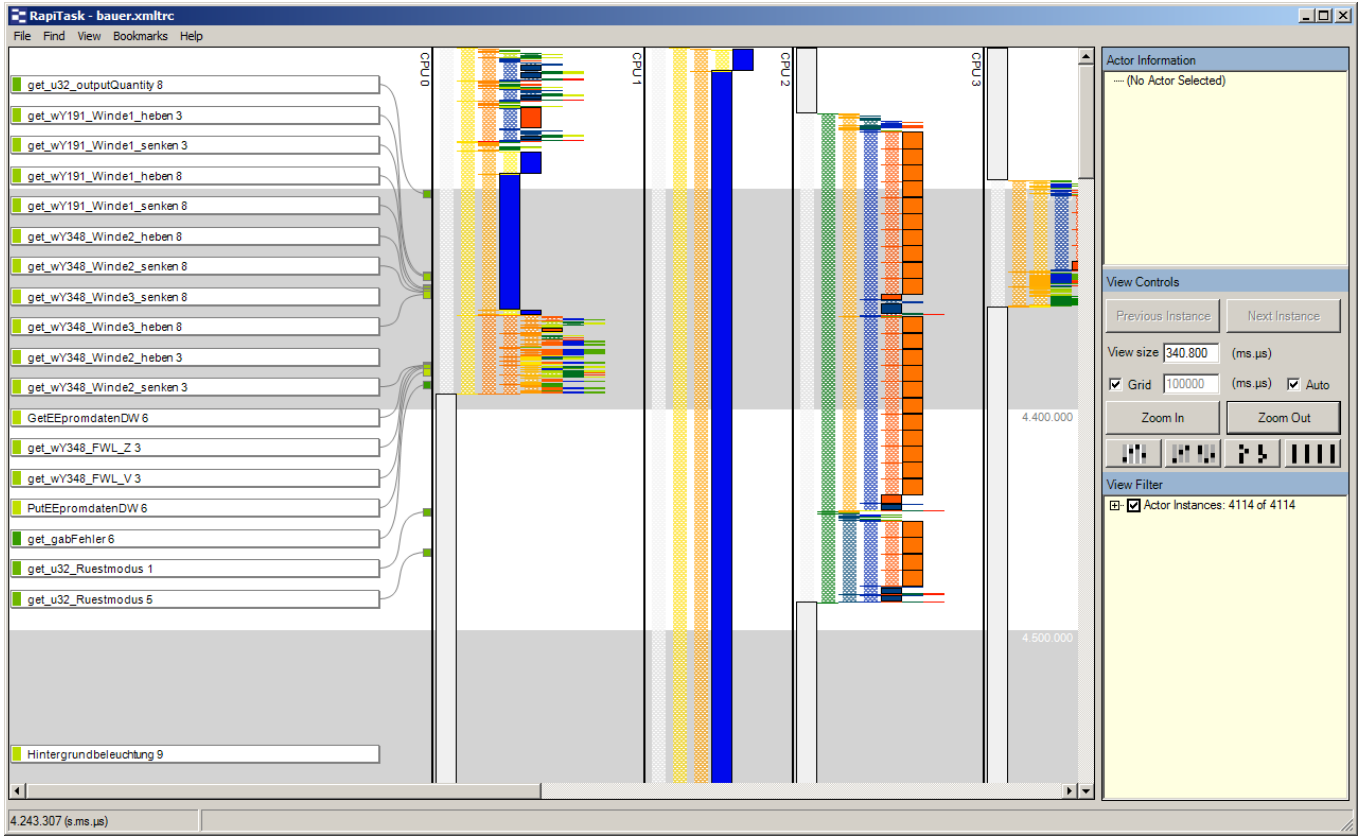
- Support was also added to filter the on-chip instruction trace buffer on load/store alternate instructions.
- This allows instrumentation points to be recorded in the trace buffer and the data can then be continuously dumped through a high-speed debug link and fed into Rapita's toolchain.
- More performance counters to measure: bus congestion, L2 cache events, etc.
- **All improvements generated by this project for the NGMP are now included in the NGMP (GR740) ASIC.**

# Trace visualisation

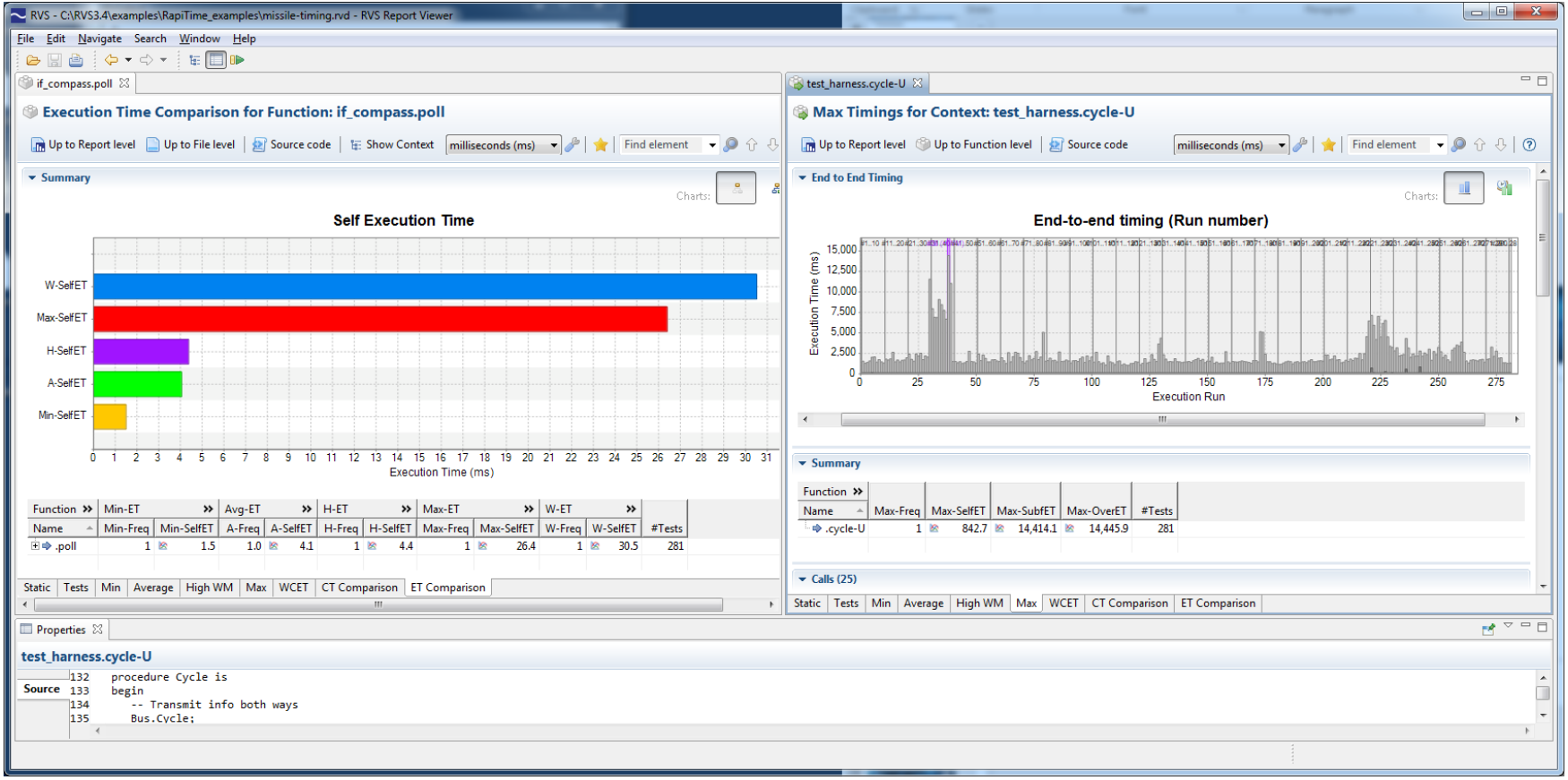
- Visualisation of multicore traces and RTOS schedule
- RapiTask
  - Traces from NGMP are filtered and processed by RapiTask to produce a graphical view of the schedule as it happened.
  - It is also possible to show the execution of individual functions within each task
  - This view can be synchronized with the timing analysis view from RapiTime
  - Allows to view of the multicore scheduling



# Multicore Trace Visualisation

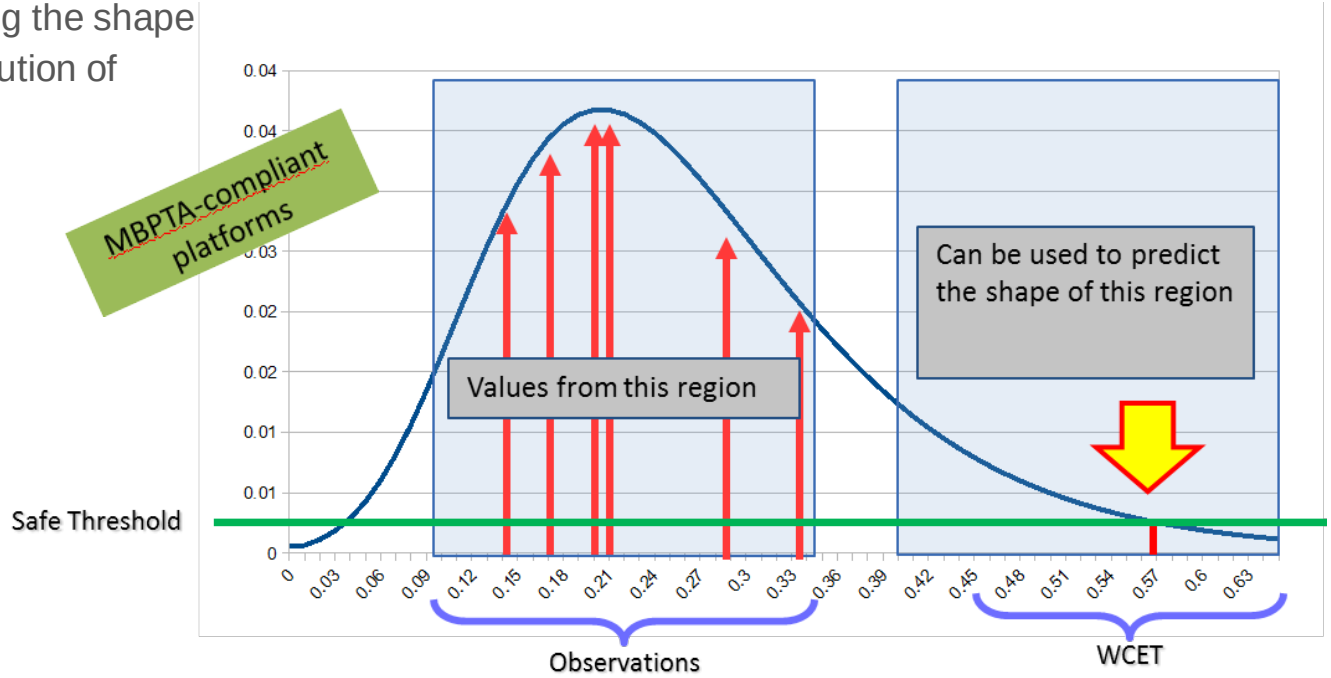


# Timing Analysis (RapiTime)



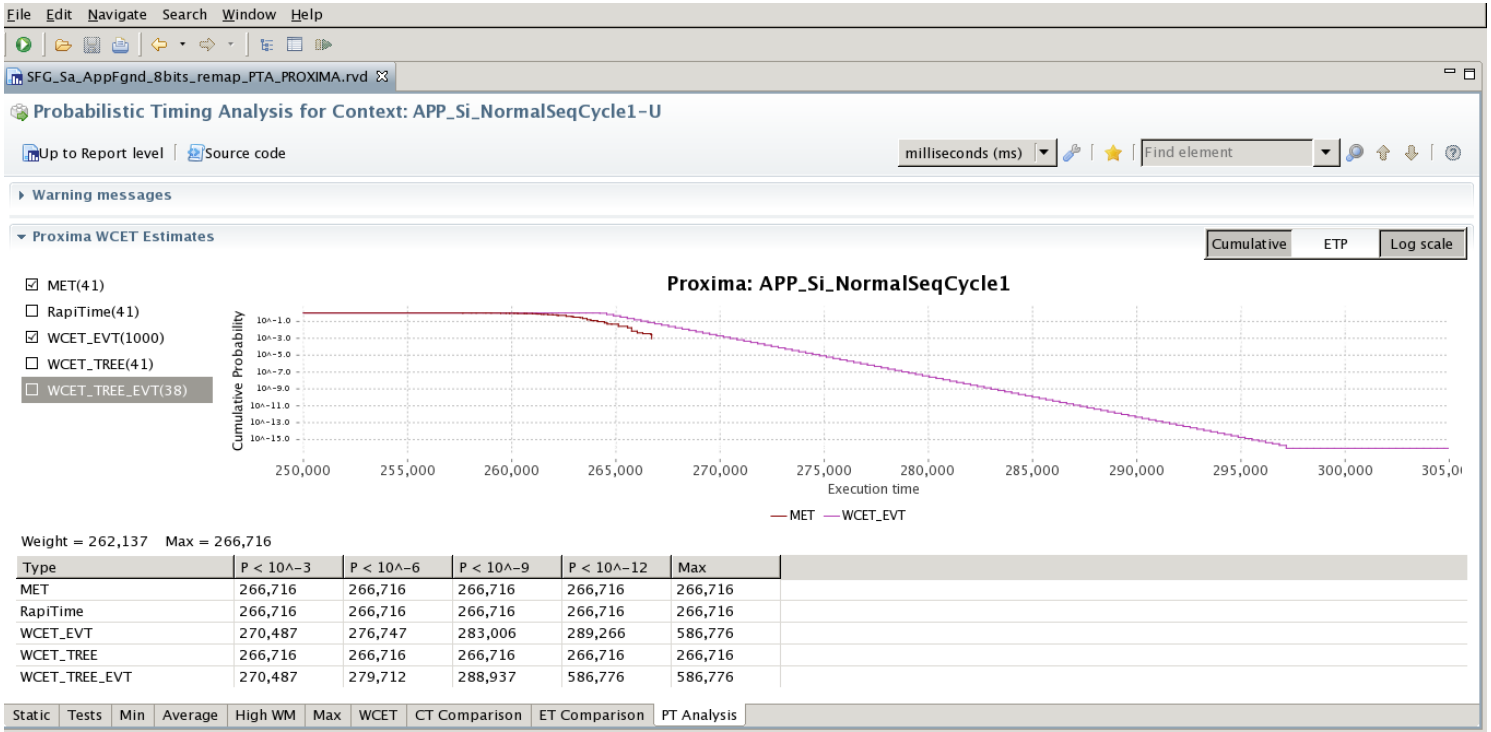
# PROARTIS: MBPTA

- Derived from a branch of Extreme Value Theory (EVT)
  - EVT allows predicting the shape of the tail of a distribution of execution times



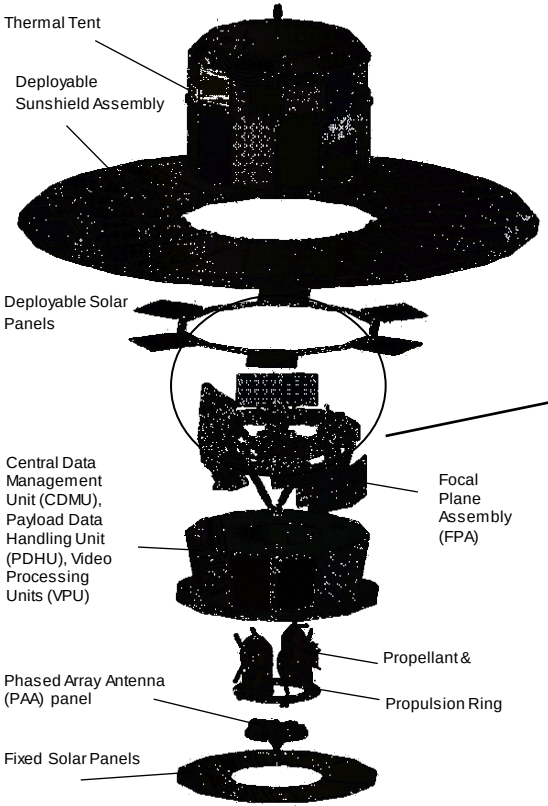
# RapiTime + MBPTA

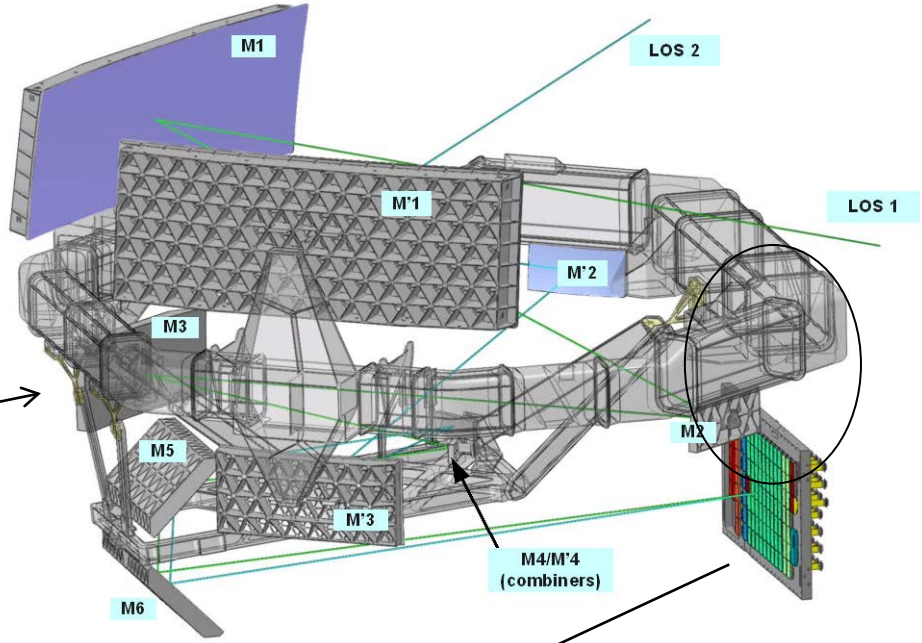
- EVT ‘tail extension’



# Industrial Case Study

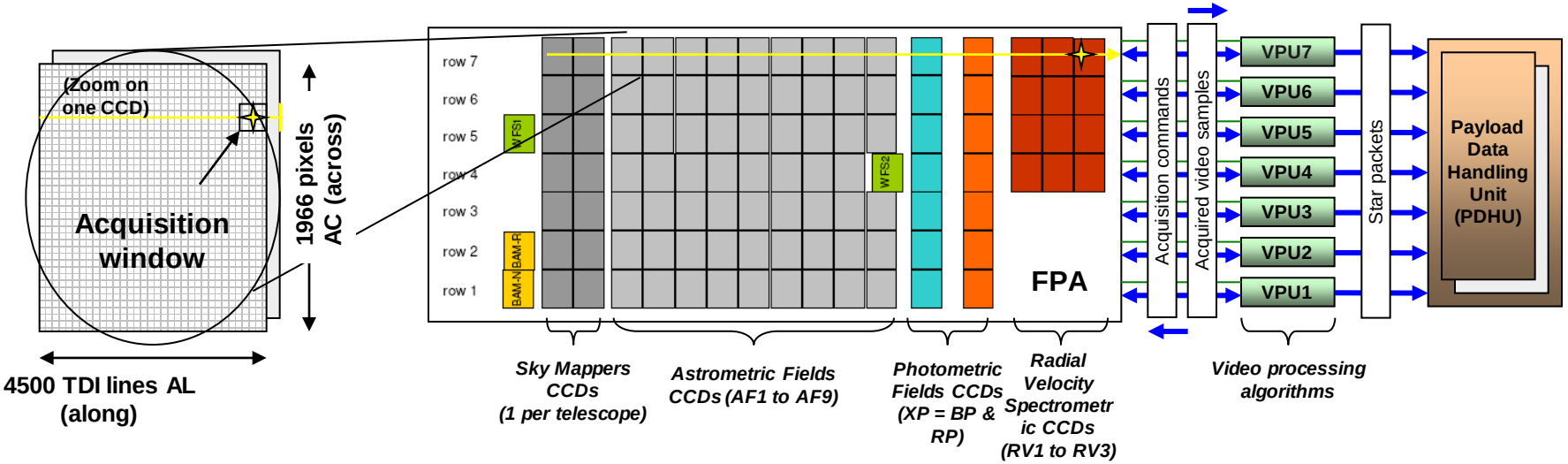
# Gaia Mission




CCD Array commanded by Gaia VPU Software

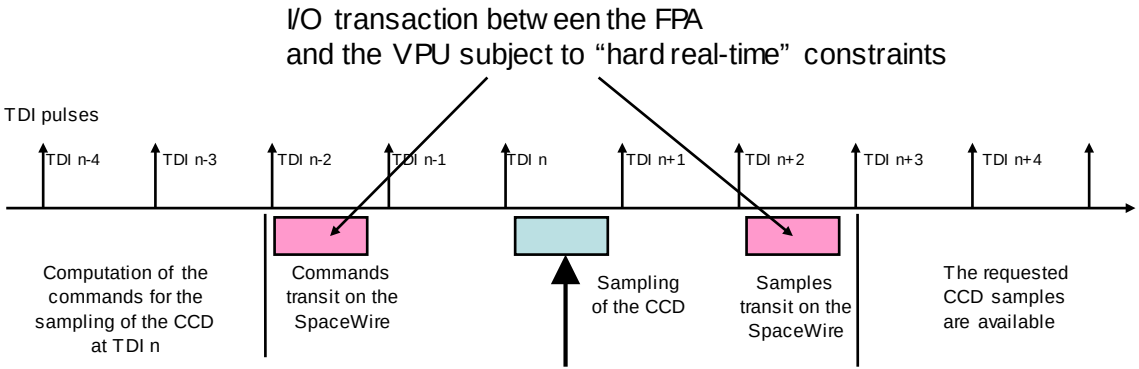
# Gaia VPU




- Each row of CCD Plane is controlled by Video Processing Unit (VPU)
- Each VPU embed an instance of the gaia vpu software with specific parameters
- Stars cross over each CDD thanks to spacecraft slowly rotating
- Each group of CDD gets specific information on stars shifting and properties

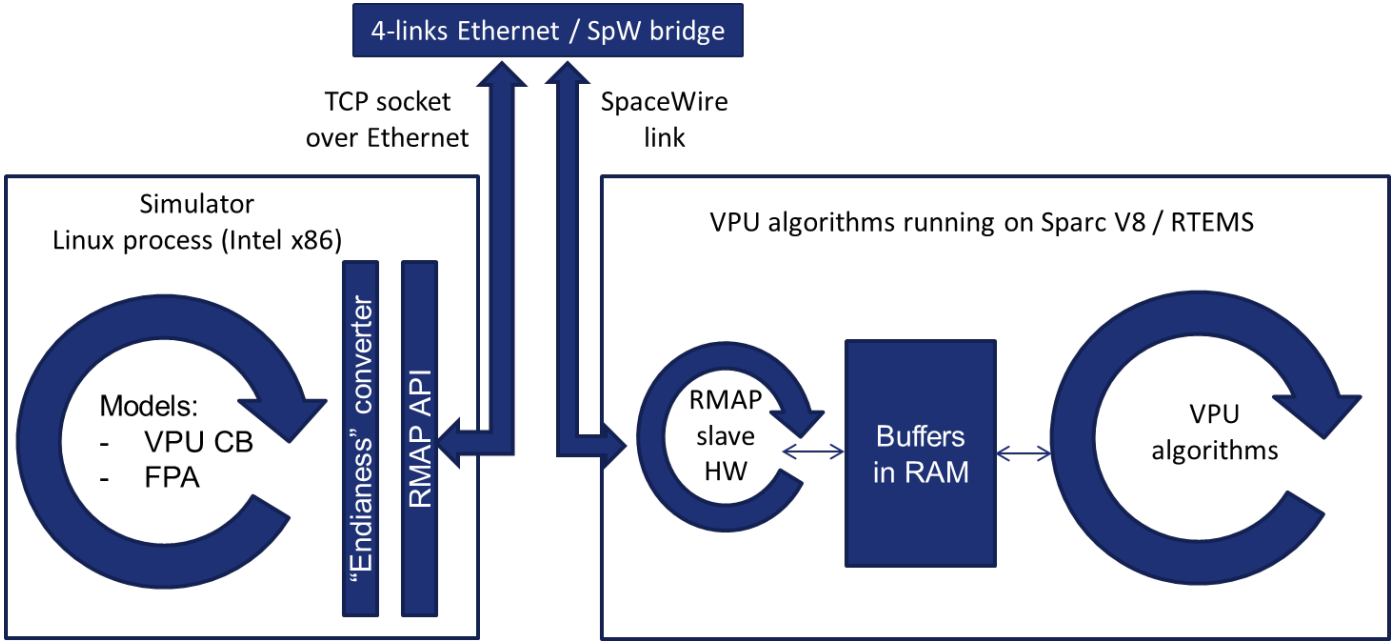
# Software cycle

- Samples from CDD each TDI (1 ms)
- Command for TDI n must be transmit for TDI n-2
- Samples arrive to VPU in TDI n+2
- To manage time constraints, data are buffered
- Data buffered are indexed by TDI



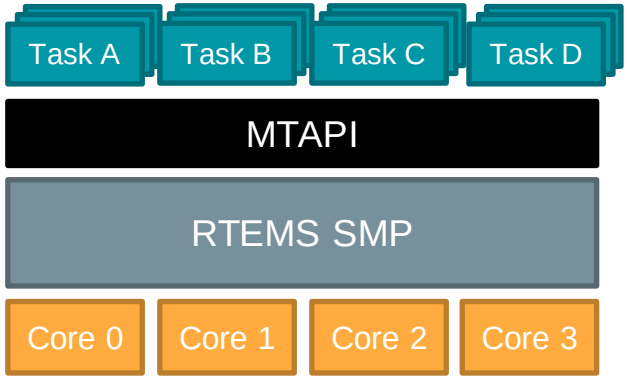
# Use case

- Simulator running on Linux to generate input data
- DMA “emulated” via SpW RMAP



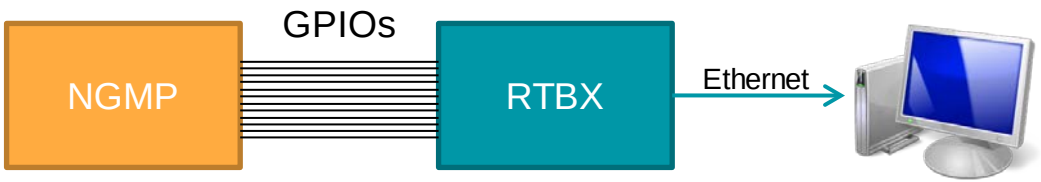
# Execution platform

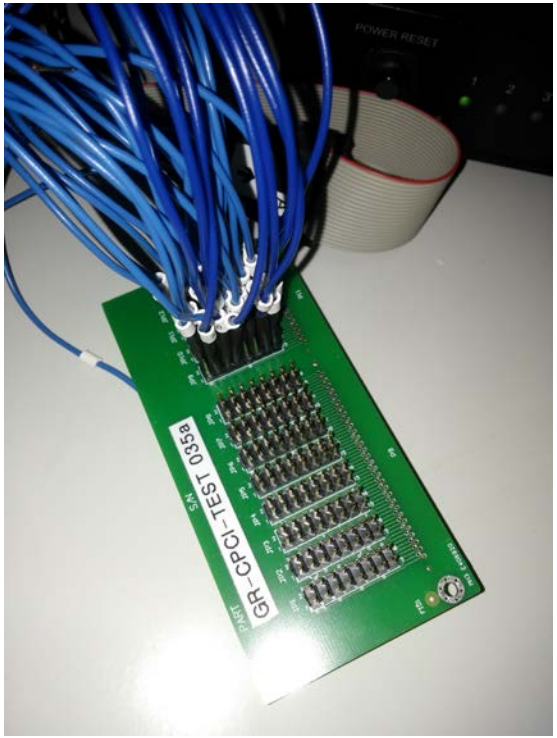
- Runs on NGMP target
- SpW on mezzanine
- RTEMS SMP with MTAPI library




# Instrumentation

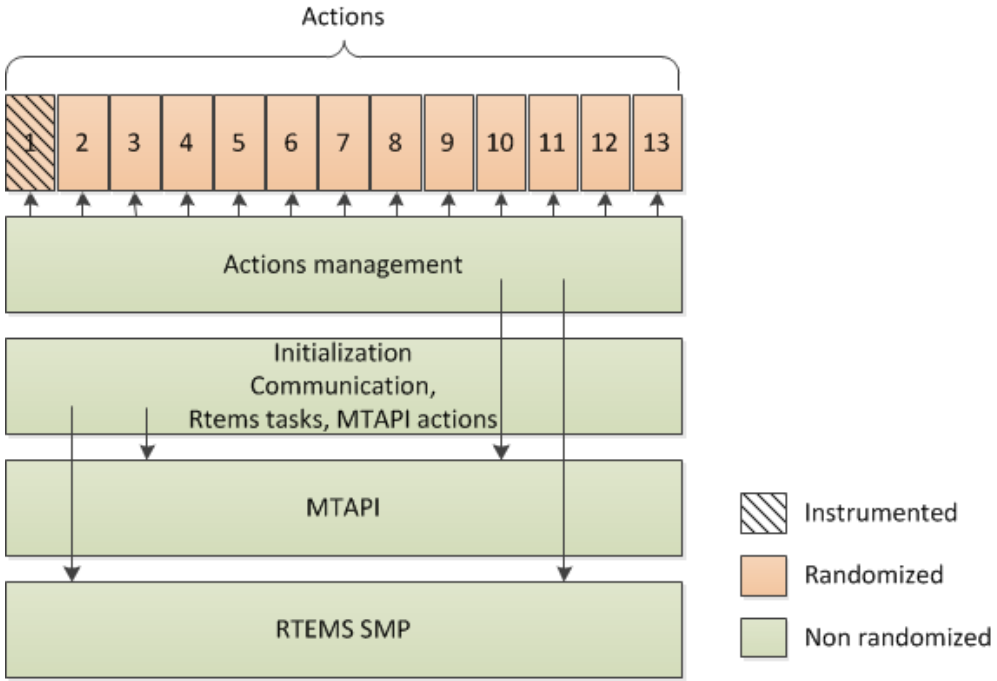
- Hardware
  - 8 bits per core
  - Output through GPIOs
  - Almost no overhead
- Instrumentation points added automatically by RapiTime before compilation
- Several instrumentation profiles



# Software Randomization Library

- Status :
  - Compilation chain working
  - Errors at runtime
  - Bugs under investigation
- Limitations :
  - RTEMS syntax not supported
  - No Dynamic memory allocation
  - Only O0 supported
  - C++ support needed



# Conclusion

- Very challenging use case
  - Complex software (Full application + RTEMS)
  - SMP application
  - Good progress but some work remaining
- Randomization library still under work
- Instrumentation for Timing Analysis is working
  - Ready for gathering results

# Conclusion

- To learn more about this topic:

**PROXIMA Industrial Workshop**

**28th June 2016**

**(http://www.proxima-project.eu)**