XIX Jornada Técnica de Ada-Spain

Virtual, 27 de abril de 2022

Aislamiento, acceso al reloj y comunicaciones en aplicaciones MaRTE OS sobre Linux

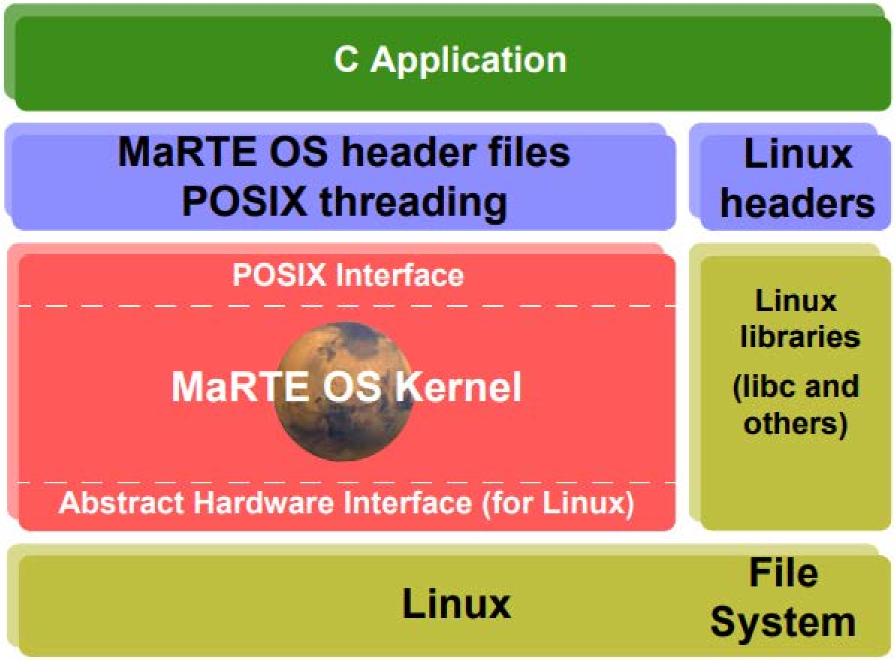
Héctor Pérez
J. Javier Gutiérrez

MaRTE OS

- Main features
  - follows the Minimal Real-Time POSIX.13 subset
  - single address space shared by kernel and application
  - support for concurrent Ada and C applications
    - implements *Ada Real-Time Systems Annex*
- Supported arquitectures

x86	**ARM**	**Linux**
• Bare-machine • XtratuM	• Raspberry pi • STM32 • Lego EV3 (WiP)	• linux • linux_lib

MaRTE OS: *linux_lib* architecture



Original image from Mario Aldea in "MaRTE OS: Overview and Linux Version"

- MaRTE OS behaves as a pthread library for Linux
  - concurrency is provided at library level (not by using Linux threads)
  - no admin privileges required to assign priorities to threads
- Applications are executed as a standard Linux user process
- *Hardware Abstract Layer* (HAL) is based on Linux system calls

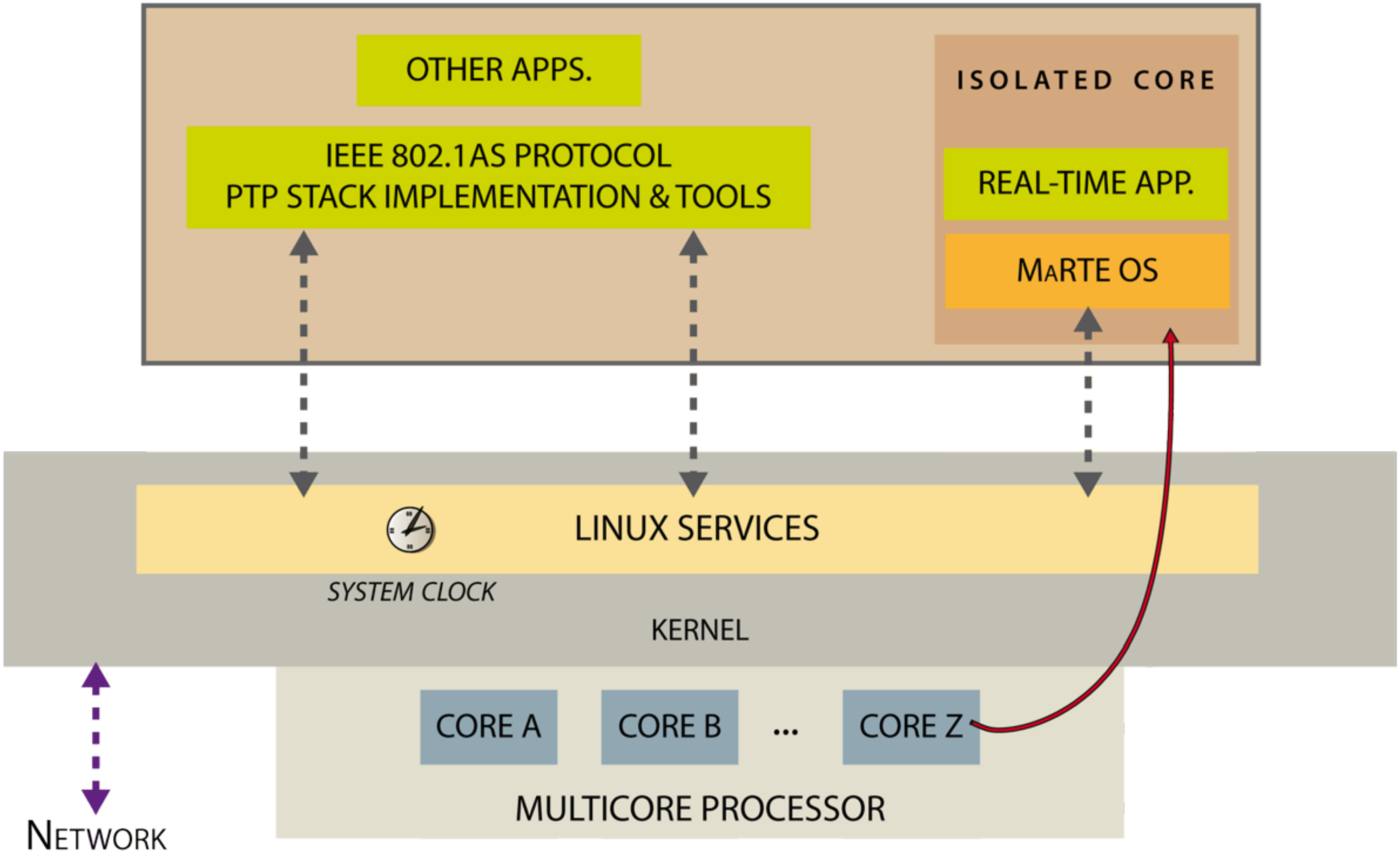
Motivation

- Online teaching during Covid-19 pandemia
  - easy to install and use for students at home
- Research platform for EDF systems
  - global-clock EDF
    - scheduling deadlines can be referred to the release time of tasks allocated in a different node
    - global clock is required
      - clock synchronization protocol and network card drivers implemented in Linux

Problem #1: predictability in Linux

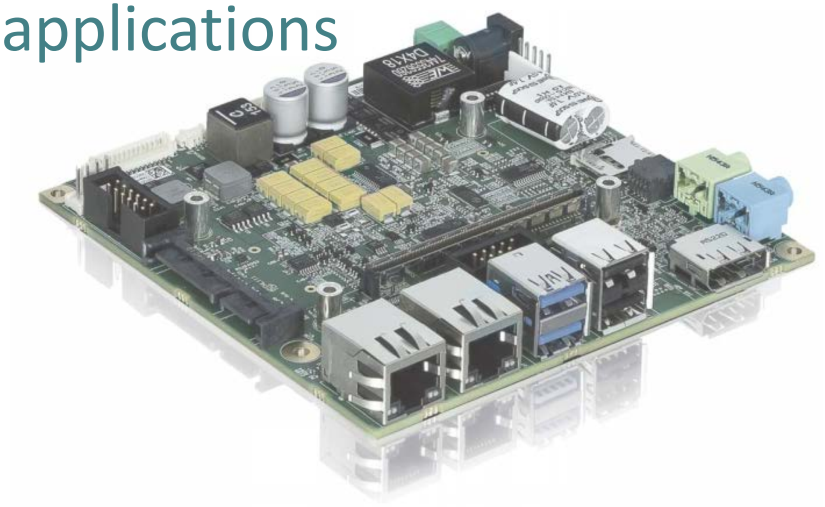
- **Issue to address:** the design of the Linux kernel favors throughput over determinism
    - interferences from other users' workload
    - interferences from the Linux kernel
- **Proposed approach:**
    - core partitioning for real-time tasks
        - *isolcpus* and *cpuset* facilities
        - tickless or *adaptative* tick mode
        - offload interrupts and workload to *non-real-time* cores
    - Linux settings for preemptible kernel
        - *PREEMPT_RT* patch or *PREEMPT* low-latency kernel

Execution framework



Tests platform

- Hardware: quad-core 1.9 GHz
- Software
  - kernel v4.4.256-rt214
  - each experiment is executed with SCHED_FIFO scheduling and maximum priority
  - **core 2** isolated for real-time applications
    - isolcpu=2
    - nohz_full=2
    - shielding with cset
    - other options also enabled



Problem #1: experimental results (1/4)

- Monitoring IRQs with workload (1 hour)
  - workload added by ***stress-ng***
    - 3 threads add synthetic worload (~ 25% system load per thread)
    - 2 threads add I/O operations
    - 1 thread adds timer interrupts
  - user load in the isolated core keeps at 0%

IRQ	CPU0	CPU1	CPU2	CPU3
I/O miscelaneous	807 333	1 198	0	3 566
Local timer interrupts	50 061 503	664 673 816	56 927	30 294 281
Rescheduling interrupts	173 638	28 420	1 247 011	428 492
Function call interrupts	665	622	9 261	668
TLB shootdowns	271	316	200	454

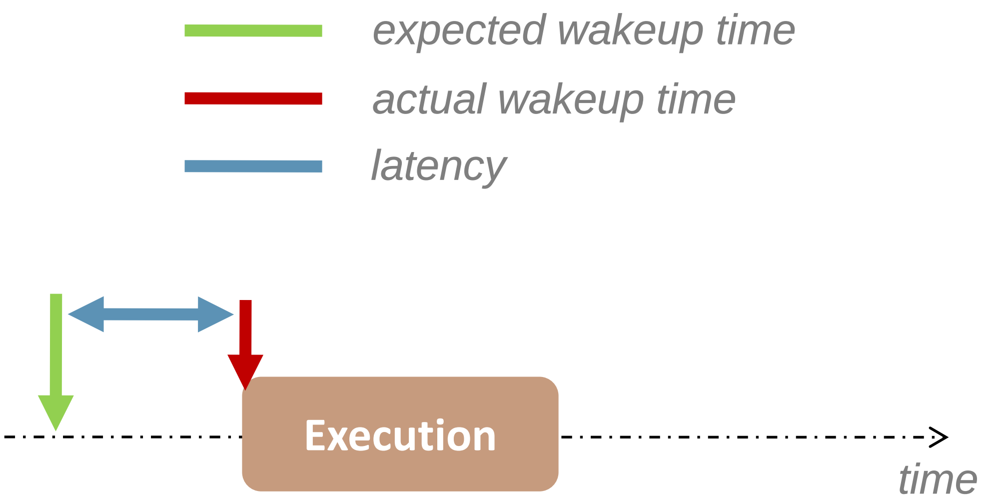
Problem #1: experimental results (2/4)

- Monitoring IRQs with workload (1 hour)
    - using the same workload than in the previous test
    - user load in the isolated core keeps at 0%
    - core 2 and core 3 disabled at boottime for Linux

IRQ	CPU0	CPU1	CPU2	CPU3
I/O miscelaneous	397 212	2 627	0	0
Local timer interrupts	79 986 427	640 569 134	4 274	29 311 597
Rescheduling interrupts	66 832	33 063	1 137 134	173 616
Function call interrupts	588	552	2 210	143
TLB shootdowns	278	330	1	133

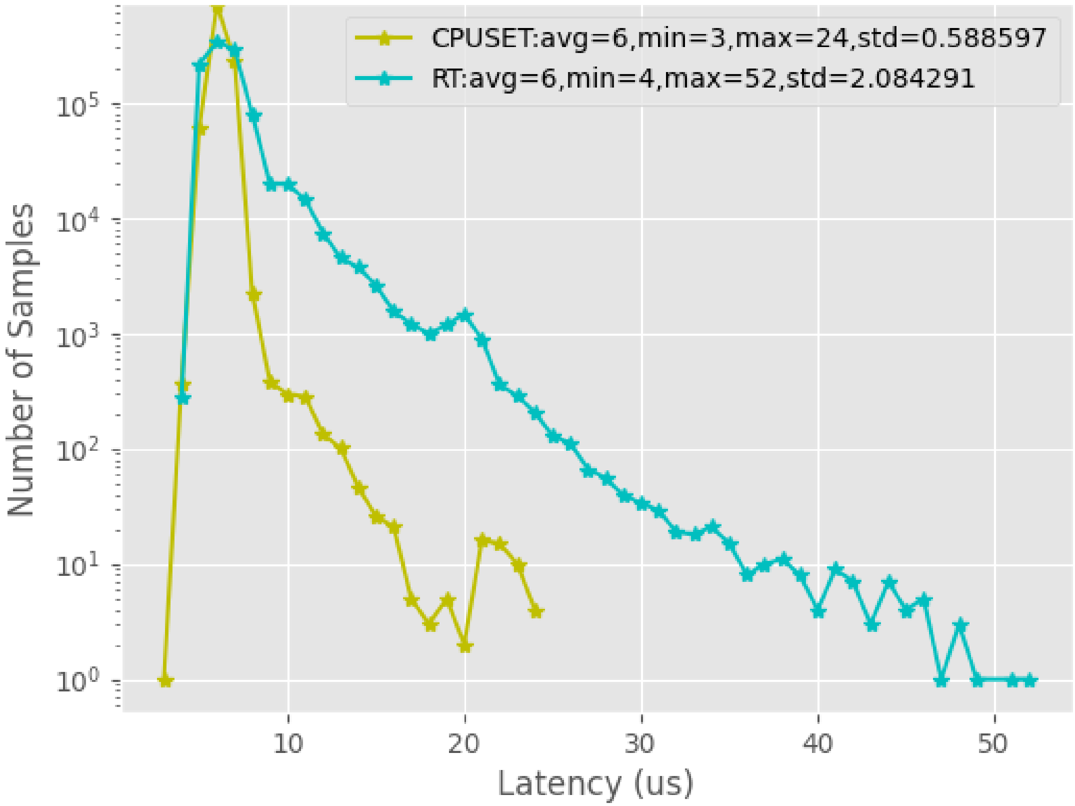
Problem #1: experimental results (3/4)

- Event-handling latency in core 2 using **cyclictest**
    - schedules timer events and compares the expected and actual wakeup time



Problem #1: experimental results (4/4)

- Event-handling latency in core 2 using **cyclictest**
  - using the same workload than in the previous test



Problem #2: clock overhead in *linux_lib* architecture

- **Issue to address:** overhead when reading the clock
  - calling *clock_gettime* is slower in MaRTE OS with *linux_lib*
    - frequent system calls can dominate overall performance
  - implementation details:
    - MaRTE OS with *linux_lib*: the *clock_gettime* function finishes in a Linux system call
    - Linux: the *clock_gettime* function is usually supported as vDSO (virtual dynamic shared object) to improve its performance
      - mechanism to export frequent **read-only** system calls to user space without a mode switch
- **Proposed approach:**
  - update *linux_lib* to use vDSO
  - minimize the number of kernel locks/unlocks in MaRTE

Problem #2: experimental results

- Overhead associated with the reading of the system clock

```
for 1 to 100000000:
    t1 = gettime(clock)
    t2 = gettime(clock)
    overhead = t2 – t1
```

- Executed 100,000,000 times
  - *timing results are expressed in microseconds*

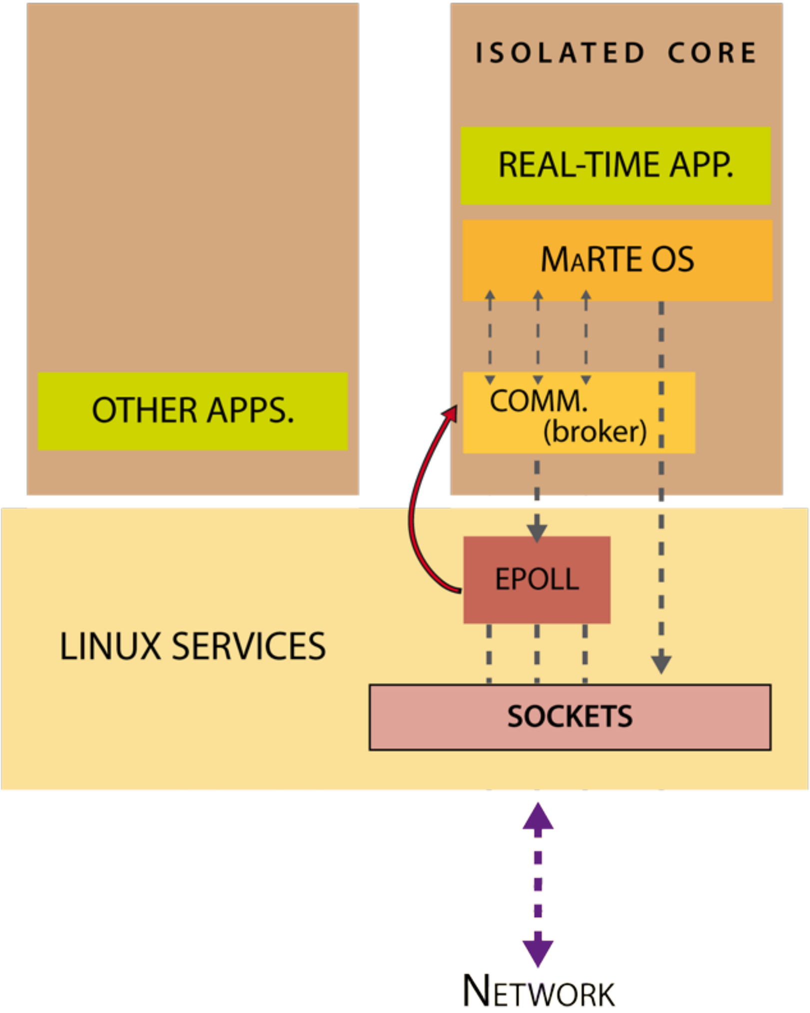
Scenario	Max	Avg	Min	Std Dev
Linux RT	12.79	0.07	0.07	0.04
MaRTE OS over Linux	13.00	2.05	2.00	0.21
MaRTE OS over Linux with vDSO	9.80	0.19	0.18	0.06

Problem #3: Networking in *linux_lib* architecture

- **Issue to address:** the design of *linux_lib* does not favors the use of blocking network calls
  - when a thread blocks in a network call, the whole application blocks
    - e.g., waiting on a socket for incoming messages
- **Proposed approach:**
  - use a new communication layer (acting as a broker) and asynchronous I/O signals
    - user threads are blocked using MaRTE synchronization mechanisms

Problem #3: Proposed approach

- Sockets are configured to be non-blocking and asynchronous
- Linux raises a signal when new incoming messages are available for reading
- epoll syscall for monitoring multiple sockets



Conclusions

- Full isolation cannot be obtained
  - maximum interferences around tens of microseconds
- Clock overhead has been reduced
  - vDSO broadly available in current systems
- Networking functionality has been enhanced
  - *non-negligible* performance penalty compared to other architectures