HIPPEROS

A Song of Research and Development

Antonio Paolillo

Huawei Dresden Research Center

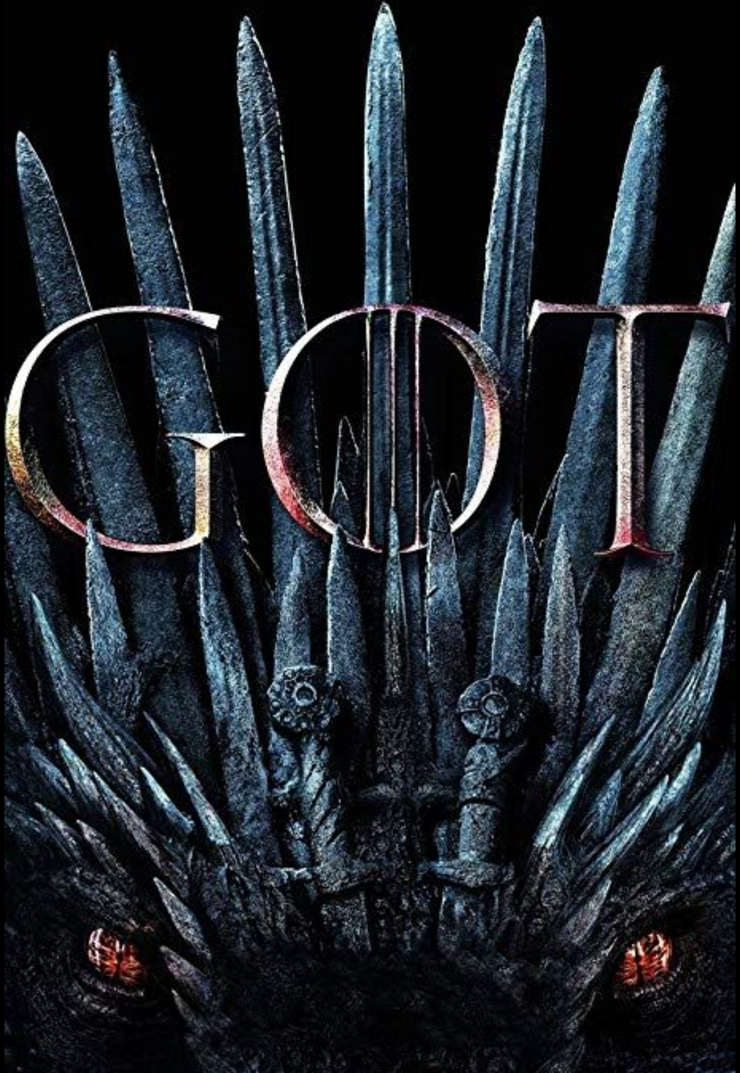
FOSDEM 2020 - micro-kernel devroom

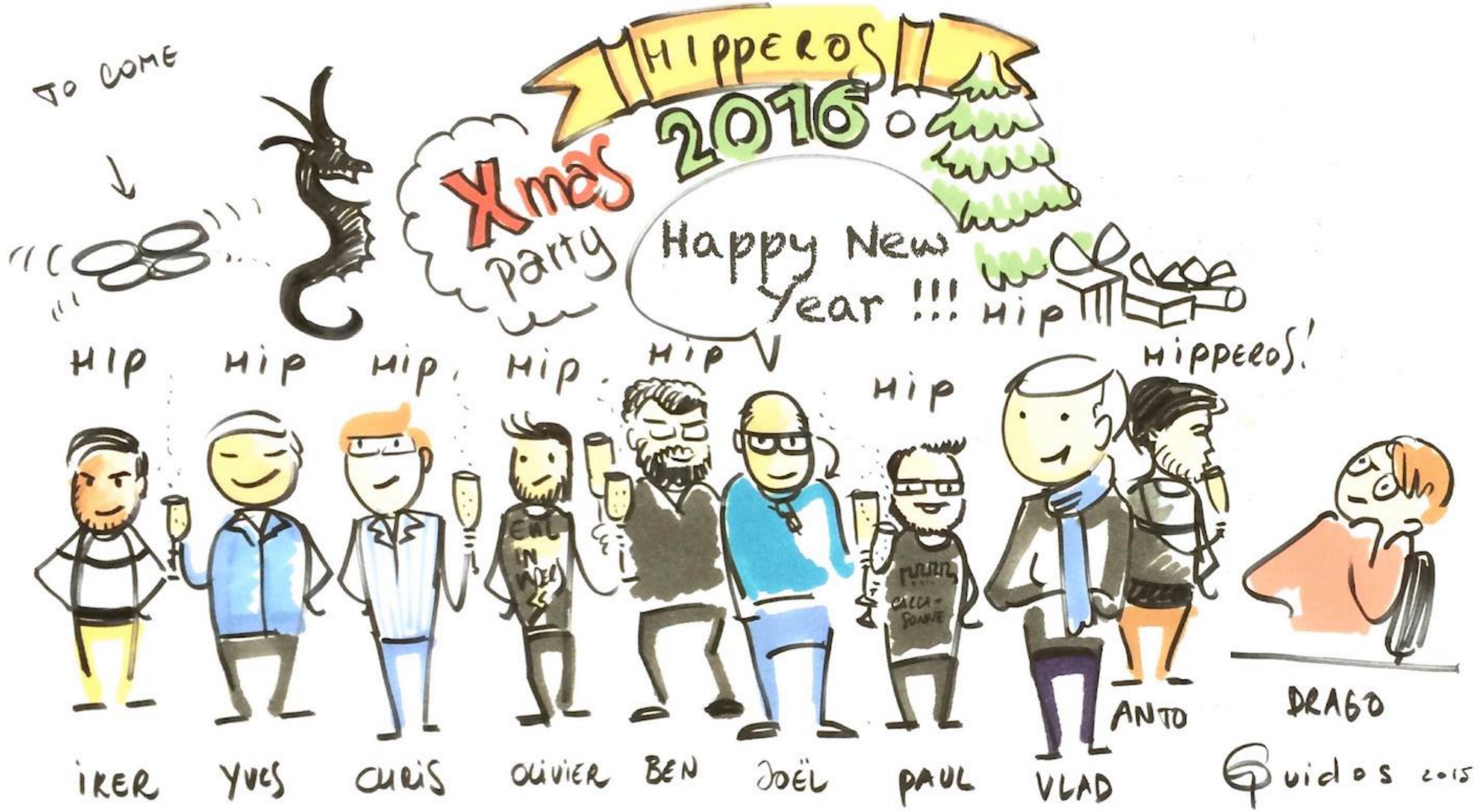
2th February 2020 - Brussels, BELGIUM

Disclaimer

This talk reflects my personal view only,
not the one of my previous or current employer.

A Song of Research and Development


GOT


TO COME
HIPPEROS
2016
Xmas party
Happy New Year !!!
HIP HIP HIP HIP HIP HIP HIP
HIPPEROS!
IKER
YVES
CHRIS
OLIVIER
BEN
JOËL
PAUL
VLAD
ANTO
DRAGO
Guidos 2015

Me? Highlights:

- 2006 - 2011 - Brussels
  - Computer Science master’s degree (ULB)
- 2012 - 2019 - Brussels
  - Back to university and **Ph.D** (ULB)
  - In parallel: joined a **spin-off** project (HIPPEROS)
- 2019 - today - Dresden
  - Huawei researcher in OS / micro-kernel field
  - Dresden Research Center

More info: https://antonio.paolillo.be

Huawei Dresden Research Center

- Applied research on OS & micro-kernels
  - Practical research & internal contributions
  - Academic contributions are targeted
  - Started in February 2019
  - 20+ researchers
- Interests:
  - Dependable OS techniques
    - Scalable kernel architecture
    - Kernel design exploration
    - Formal verification, testing & certification
  - Virtualization technique

- Contact me to know more and have some fun with us :-)

Agenda

1. The human story behind the RTOS
2. HIPPEROS high level features
3. Run-time model & build environment
4. Architecture overview and main design choices
5. Research results achieved
6. Conclusions and bright future

Agenda

1. **The human story behind the RTOS**
2. HIPPEROS high level features
3. Run-time model & build environment
4. Architecture overview and main design choices
5. Research results achieved
6. Conclusions and bright future

Alma mater

Home of research

ULB UNIVERSITÉ LIBRE DE BRUXELLES

University’s missions (or roles)

Teaching

Research

Valorisation

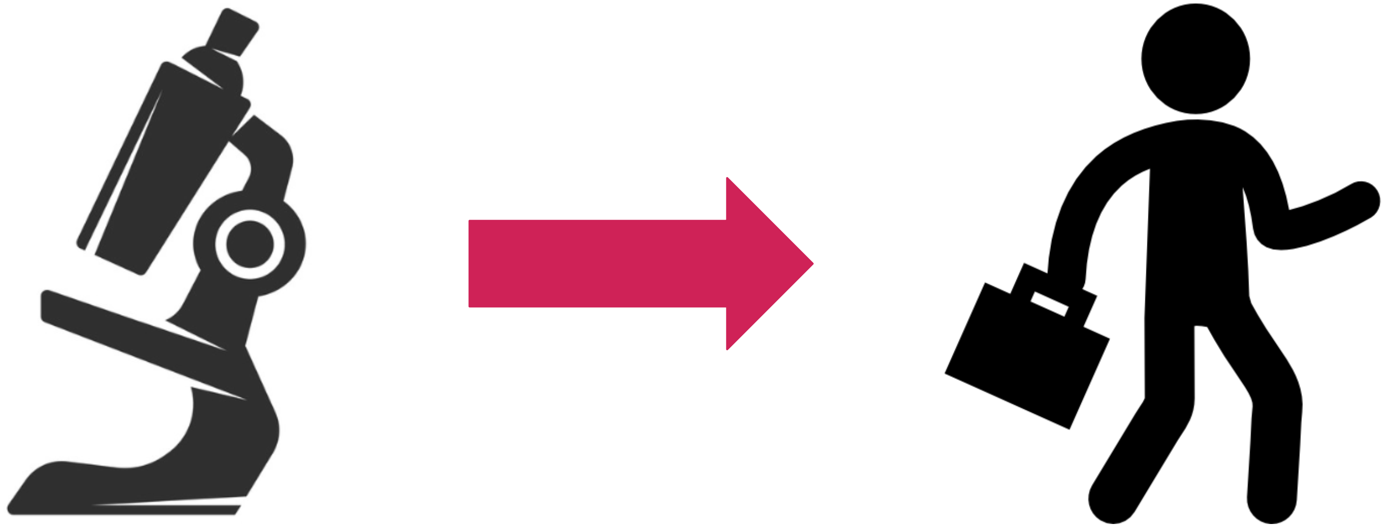
University’s missions

Teaching

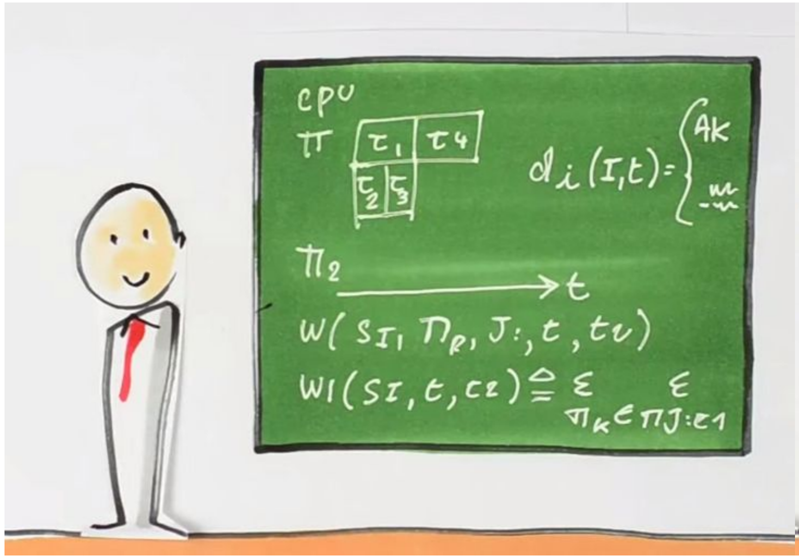
Research

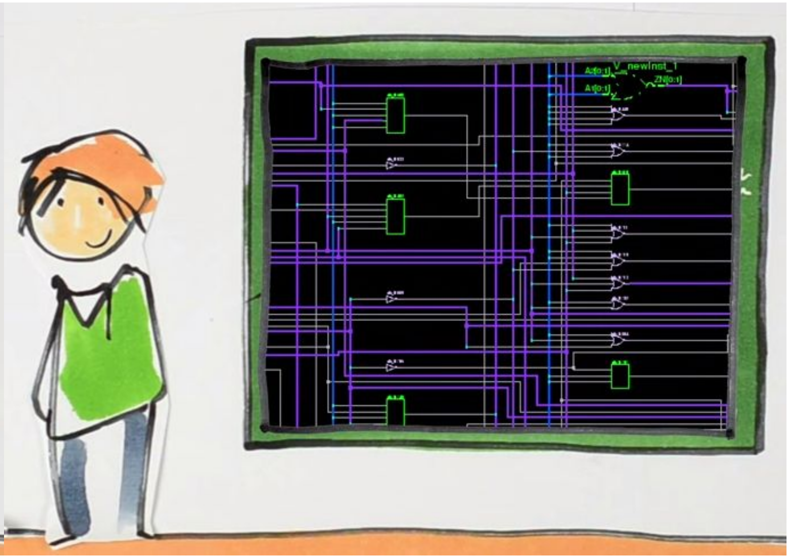
Valorisation

A spin-off: from science to a commercial product



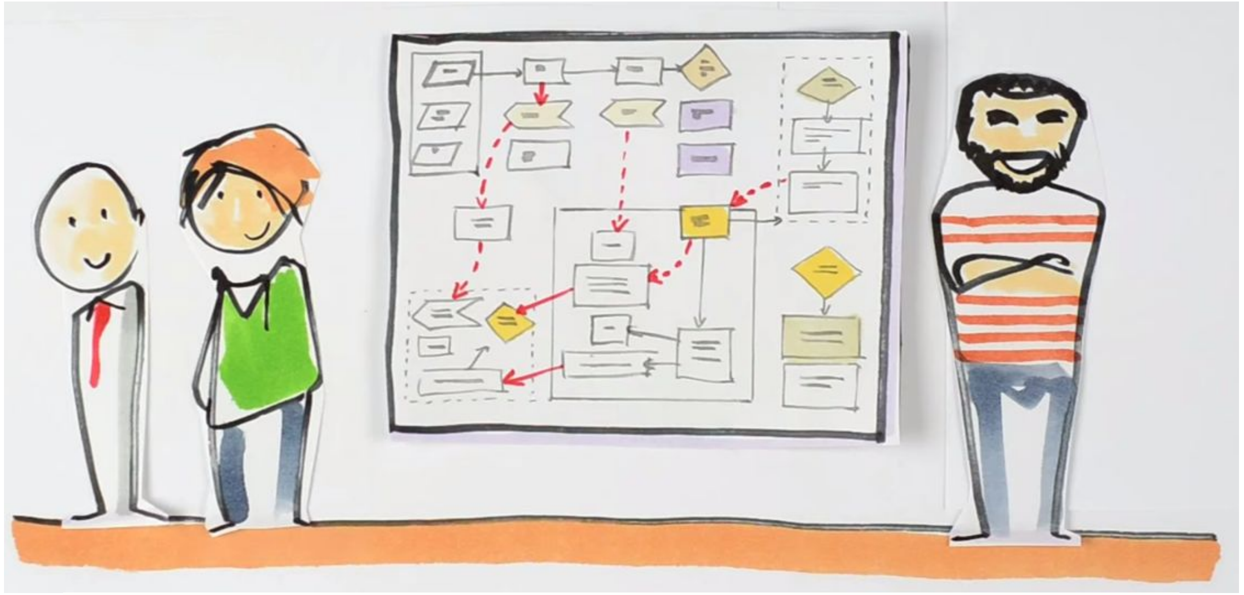
Parallel Architecture for Real-Time Systems





PARTS

Parallel Architecture for Real-Time Systems



HIPPEROS®

Predictable Real-Time, Proven Performance

H igh
I
P erformance
P arallel
E mbedded
R eal-time
O perating
S ystems

H igh
I
P erformance
P arallel
E mbedded
R eal-time
O perating
S ystem<u>**s**</u>

Basic idea

Create a **company** selling products and services around the topic of **Real-Time Operating Systems**, including the creation of **a new micro-kernel** for high-end **embedded systems** with an innovative software architecture, backed-up by research (both theoretical and applied), designed, developed and maintained with “good” (and agile…) software methodology.

Within this business, maintain strong links with universities and the research world, by validating the design in an academic environment and continuous research activities.

The very ambitious objective was to deploy the RTOS in any device imaginable…

2006 - project kickoff

- 2 research labs in ULB university
  - Theory of real-time scheduling
    http://parts.ulb.ac.be

  - Micro-electronics and digital system design
    BEAMS-EE in http://www.bruface.eu/

- Research entrepreneur
  - Fundings

  - Good storytelling

  - First design ideas

Fall 2012

- The developers joined! (including me)
- Fundings provided by the CRAFTERS project
- Foundations for our new kernel

Embedded development

2013 - 2015

- Started “Sprint 0” in April 2013 and hands-on development in June 2013

  “our code-base is returning in user mode”

- Spin-off company “HIPPEROS” created in 2014
- CRAFTERS funded project
- Met Andrew Tanenbaum *“only the beginning”*

2016 - 2019

- Maturing the product, lots of R&D activities
- Finishing the Ph.D [1]
- Tulipp funded H2020 project
  - Use cases for the RTOS
  - Features roadmap (networks, libraries, etc.)
- **4 to 15 people**… depending how (and when) you count ;-)
  → 4-5 developers/testers/designers

[1] Everything related to my thesis is publicly available: https://antonio.paolillo.be/research.html

Why a new kernel / RTOS / …?

Facilitate development of high performance / low-power / safety-critical applications

Modern hardware used safely, exploitation of hardware parallelism

Need a low footprint but feature-rich RTOS

Why a new kernel / RTOS / …?

Facilitate development of high performance / low-power / safety-critical applications

Modern hardware used safely, exploitation of hardware parallelism

Need a low footprint but feature-rich RTOS:

1. multi-thread support
2. power management support
3. real-time scheduling support & real-time guarantees
4. parallelism at kernel level *that scales*
5. support for **heterogeneous** platforms
6. *possibly*, certification

Why a new kernel / RTOS / …?

Facilitate development of high performance / low-power / safety-critical applications

Modern hardware used safely, exploitation of hardware parallelism

Need a low footprint but feature-rich RTOS:

1. multi-thread support
2. power management support
3. real-time scheduling support & real-time guarantees
4. parallelism at kernel level *that scales*
5. support for **heterogeneous** platforms
6. *possibly*, certification

To our knowledge, this combination of requirements was quite rare when the project started.

Agenda

1. The human story behind the RTOS
2. **HIPPEROS high level features**
3. Run-time model & build environment
4. Architecture overview and main design choices
5. Research results achieved
6. Conclusions and bright future

Product Vision

OS for high-end embedded systems

- Performant
- Reliable
- Efficient

Demanding Applications

- Computer Vision
- Embedded AI
- Robotics

C Standard Library

Support for the standard C API

- POSIX-compliant
- User code is standard C/C++

Native API for additional features

- ISRs
- IPC
- ...

Supported toolchains

- GCC
- LLVM
- Xilinx SDSoC

POSIX Compliance

POSIX-compliant (PSE52 + useful bits)

- Multithreading
- File system
- Networking
- …

Required to support large frameworks (e.g.: OpenCV)

Supported architectures

ARMv7-A

- IMX6
- Zynq-7000

ARMv8-A

- Zynq UltraScale+

x86

PowerISA 2.06B (PowerPC 64 bits)

- QorIQ-T series

ARC EM2 (discontinued)

Device Drivers

FPGA Dynamic Partial Reconfiguration

Ethernet

SDIO

- FAT32/FAT16

CAN (experimental)

GPIO, I2C, SPI, UART, timers

Network

IPv4 and IPv6

Control, link, transport protocols, applications

- ICMP, IGMP, ICMPv6, MLD
- PPP, ARP, DNP (IPv6)
- UDP, TCP
- DNS, SNMP, DHCP, HTTP, TFTP servers

Agenda

1. The human story behind the RTOS
2. HIPPEROS high level features
3. **Run-time model & build environment**
4. Architecture overview and main design choices
5. Research results achieved
6. Conclusions and bright future

Task, Process & Threads

Tasks & Processes

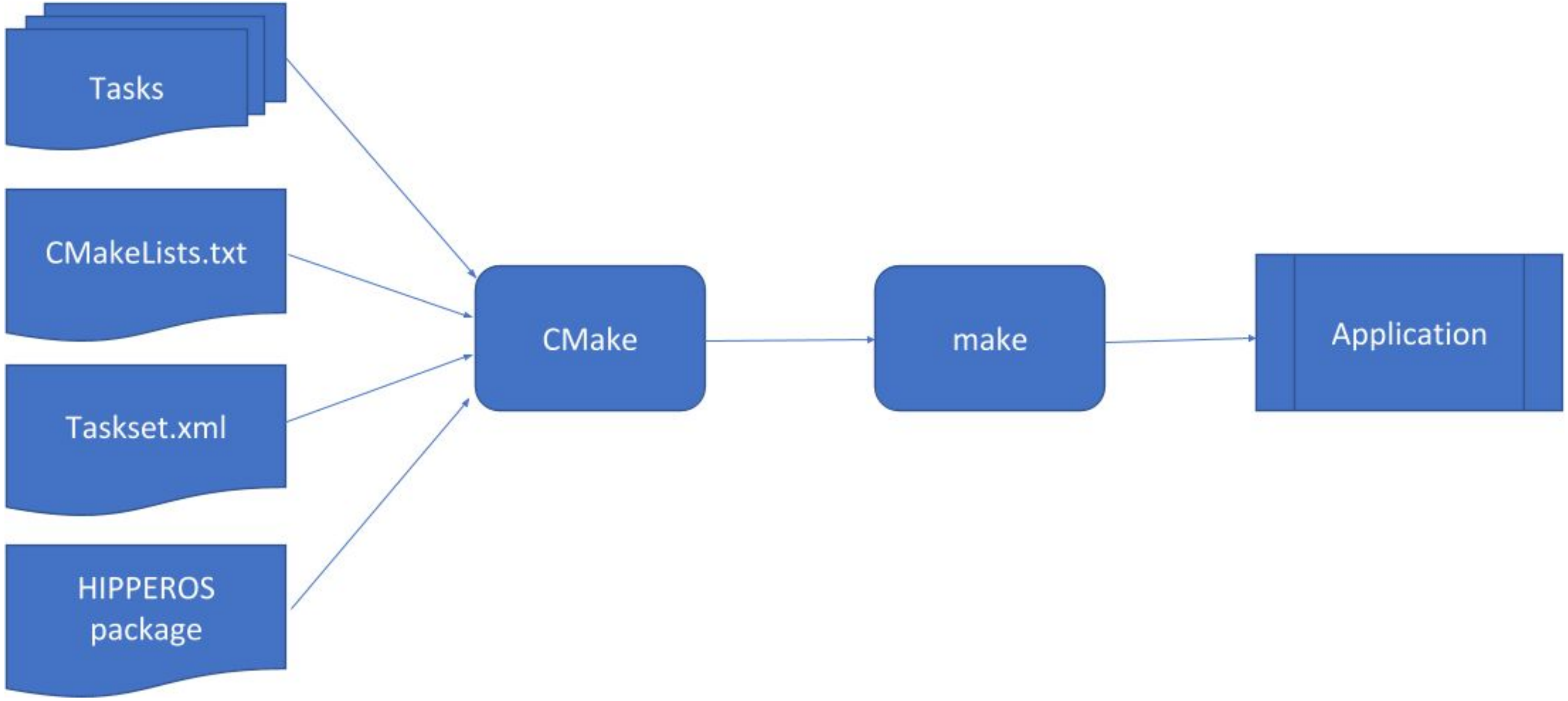
- Application = set of tasks
- 1 task → 1 process
- Process = task + OS structures
  - Process state (active/inactive/blocked)
  - Threads (active/inactive/processing/blocked)
  - Page Table
- Real-time properties enforced by the OS

Tasks & Processes & Threads

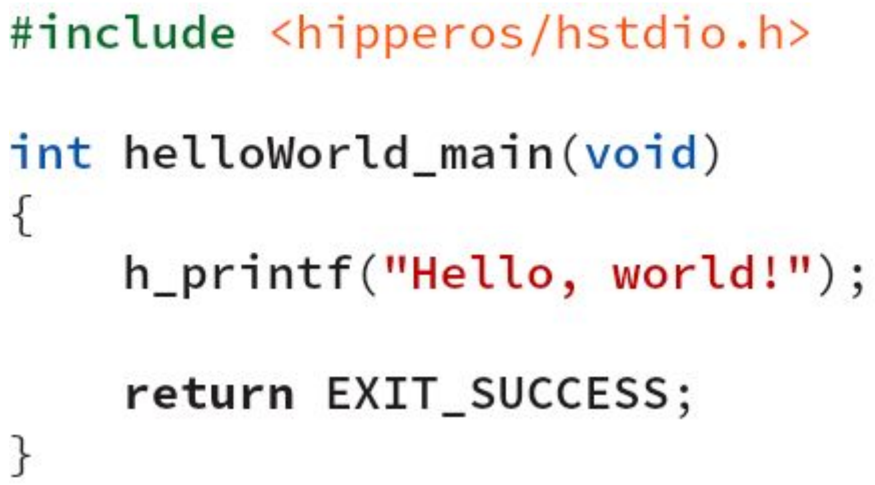
- Thread = The schedulable entity
- 1 process → N threads
- 1 thread → 1 context
- Inherit real-time properties from the process
- Fine-grained scheduling state

Create an application

In practice: build an application

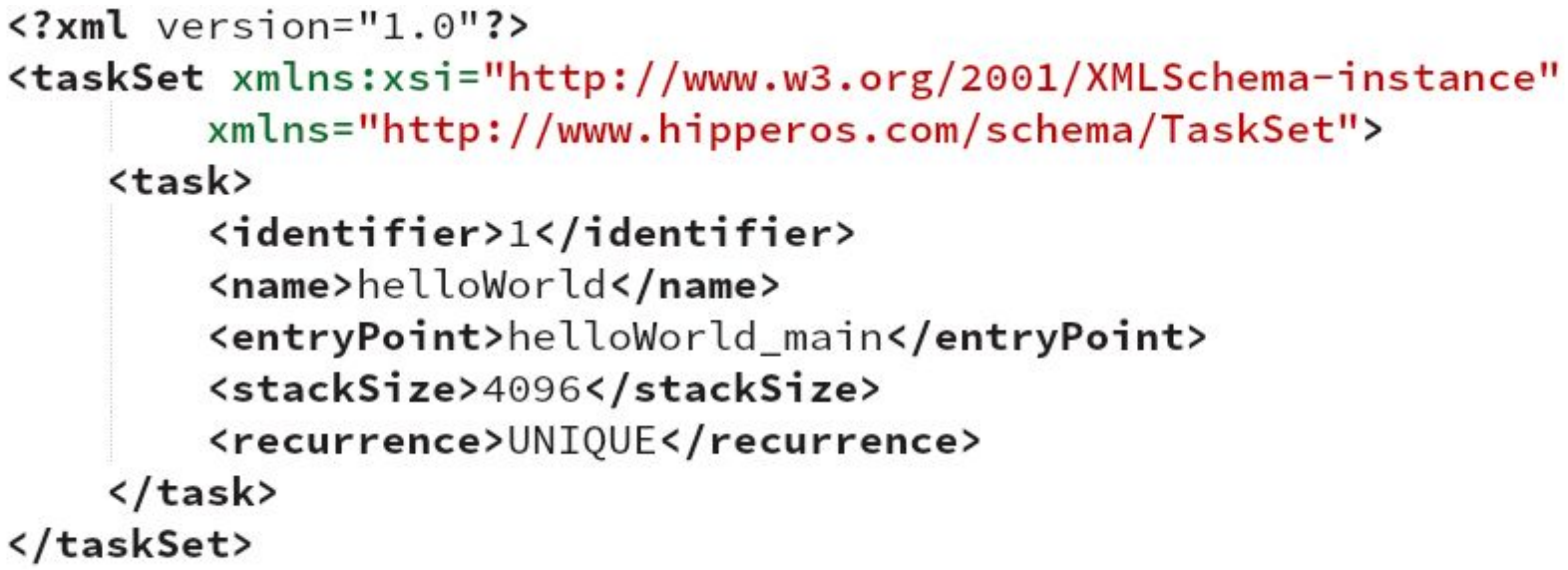


Tasks = C/C++ programs



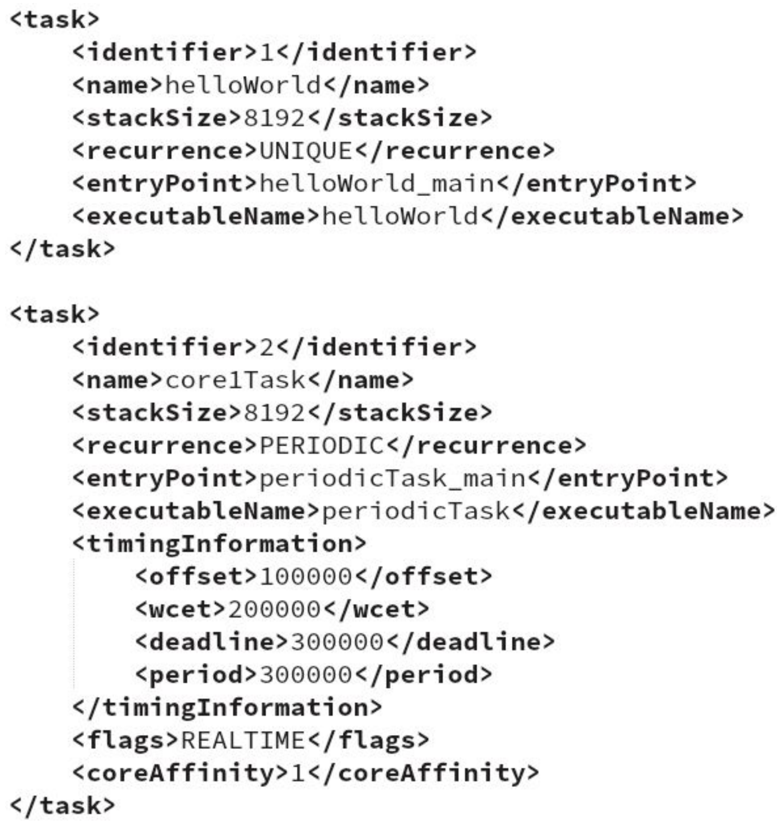
```
#include <hipperos/hstdio.h>

int helloWorld_main(void)
{
    h_printf("Hello, world!");

    return EXIT_SUCCESS;
}
```

HIPPEROS application = set of pre-defined tasks



```xml
<?xml version="1.0"?>
<taskSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.hipperos.com/schema/TaskSet">
    <task>
        <identifier>1</identifier>
        <name>helloWorld</name>
        <entryPoint>helloWorld_main</entryPoint>
        <stackSize>4096</stackSize>
        <recurrence>UNIQUE</recurrence>
    </task>
</taskSet>
```

Task set file



```
<task>
    <identifier>1</identifier>
    <name>helloWorld</name>
    <stackSize>8192</stackSize>
    <recurrence>UNIQUE</recurrence>
    <entryPoint>helloWorld_main</entryPoint>
    <executableName>helloWorld</executableName>
</task>

<task>
    <identifier>2</identifier>
    <name>core1Task</name>
    <stackSize>8192</stackSize>
    <recurrence>PERIODIC</recurrence>
    <entryPoint>periodicTask_main</entryPoint>
    <executableName>periodicTask</executableName>
    <timingInformation>
        <offset>100000</offset>
        <wcet>200000</wcet>
        <deadline>300000</deadline>
        <period>300000</period>
    </timingInformation>
    <flags>REALTIME</flags>
    <coreAffinity>1</coreAffinity>
</task>
```

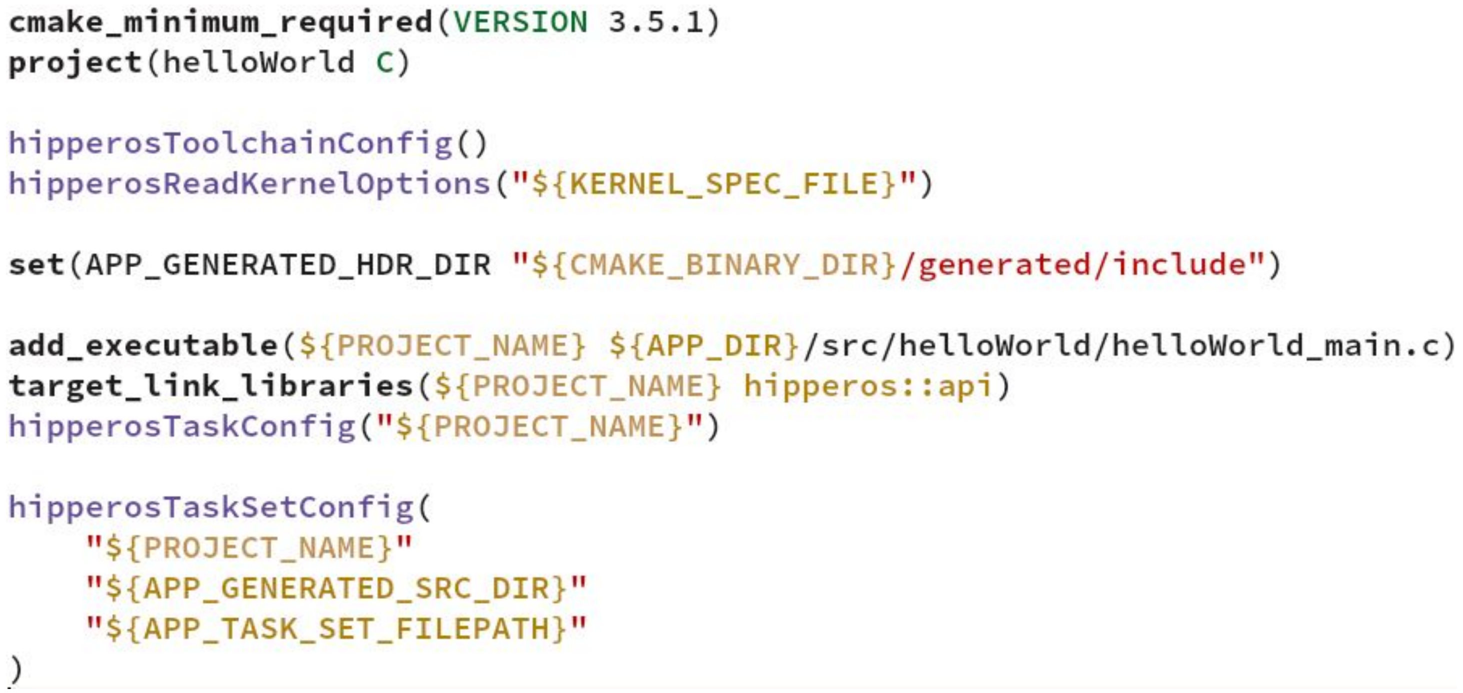
- Timing parameters
- Periodicity
- Code
- Core affinities

Task set file

```
<task>
    <identifier>1</identifier>
    <name>helloWorld</name>
    <stackSize>8192</stackSize>
    <recurrence>UNIQUE</recurrence>
    <entryPoint>helloWorld_main</entryPoint>
    <executableName>helloWorld</executableName>
</task>

<task>
    <identifier>2</identifier>
    <name>core1Task</name>
    <stackSize>8192</stackSize>
    <recurrence>PERIODIC</recurrence>
    <entryPoint>periodicTask_main</entryPoint>
    <executableName>periodicTask</executableName>
    <timingInformation>
        <offset>100000</offset>
        <wcet>200000</wcet>
        <deadline>300000</deadline>
        <period>300000</period>
    </timingInformation>
    <flags>REALTIME</flags>
    <coreAffinity>1</coreAffinity>
</task>
```

- Timing parameters
- Periodicity
- Code
- Core affinities

Configure the build with CMake



```
cmake_minimum_required(VERSION 3.5.1)
project(helloWorld C)

hipperosToolchainConfig()
hipperosReadKernelOptions("${KERNEL_SPEC_FILE}")

set(APP_GENERATED_HDR_DIR "${CMAKE_BINARY_DIR}/generated/include")

add_executable(${PROJECT_NAME} ${APP_DIR}/src/helloWorld/helloWorld_main.c)
target_link_libraries(${PROJECT_NAME} hipperos::api)
hipperosTaskConfig("${PROJECT_NAME}")

hipperosTaskSetConfig(
    "${PROJECT_NAME}"
    "${APP_GENERATED_SRC_DIR}"
    "${APP_TASK_SET_FILEPATH}"
)
```

Miscellaneous

- Processes at run-time: 1-1 tasks in the task set
  - No other tasks!
    → very predictable system
  - Dynamicity could be implemented
- Device drivers have a special flag: **SYSTEM**

Build environment

1. Customized kernel
2. Application build definition
3. Developer environment

Custom kernel for use case requirements

Each use case has its own purpose

- Requires some modules
- Do not use some other ones

Modules that are not needed may be removed at compile-time

- This decreases memory consumption

Application build environment provided

HIPPEROS is distributed with a CMake-based build environment

- Easy definition of the system
- Fast development loop with provided scripts

Controlled development environment (SDK)

- **Docker container** image with the complete build environment
- No issue due to interaction with other pre-installed tools
- Build environments for different versions can co-exist on the same system
- Native performance
- Used both on continuous build servers and on developers' systems

Agenda

1. The human story behind the RTOS
2. HIPPEROS high level features
3. Run-time model & build environment
4. **Architecture overview and main design choices**
5. Research results achieved
6. Conclusions and bright future

OS requirements

Requirements

- Multitasking
- Hardware abstractions
- Software abstractions (ex: libraries)

Challenges:

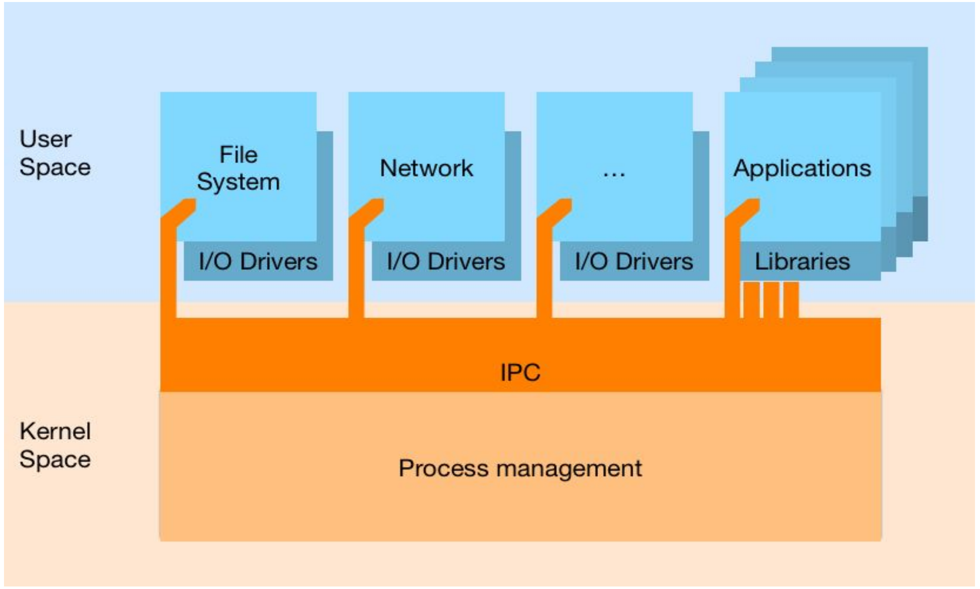
- **Reliability** of large software systems
- **Predictability** on complex hardware architectures
- **Performance** for demanding applications
- **Security** in a connected world

HIPPEROS features for **reliability** (1)

Micro-kernel based OS

(Nearly) everything runs in user-space
→ where the complexity lies
→ kernel stays simple

Very little code has the potential to crash the whole system at run-time

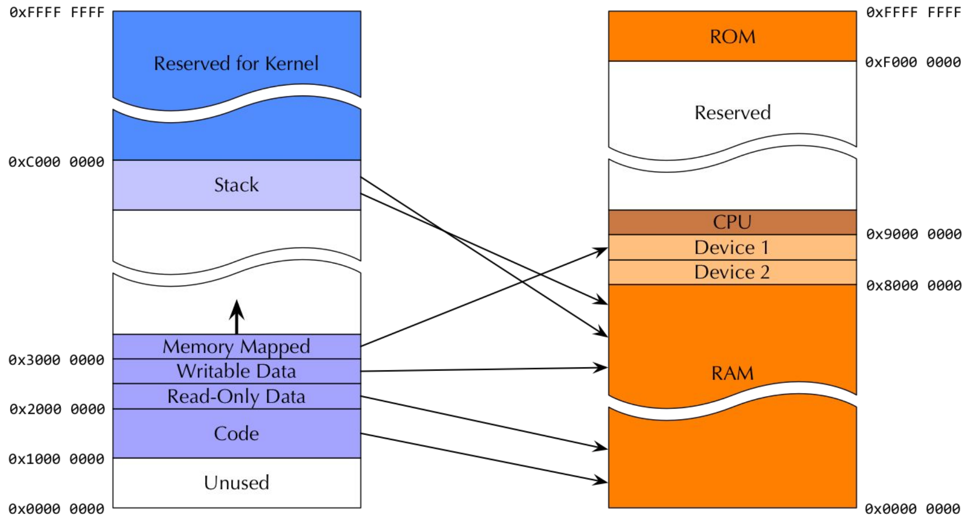


HIPPEROS features for **reliability** (2)

Memory virtualisation

- Full memory virtualisation
  → Isolation of user processes

- Access rights system (R/W/E)



HIPPEROS features for **predictability**

Hard Real-Time schedulers

Support for multicore HRT schedulers

Time Guards

Real-time monitors of execution time and deadlines

Master-Slave architecture

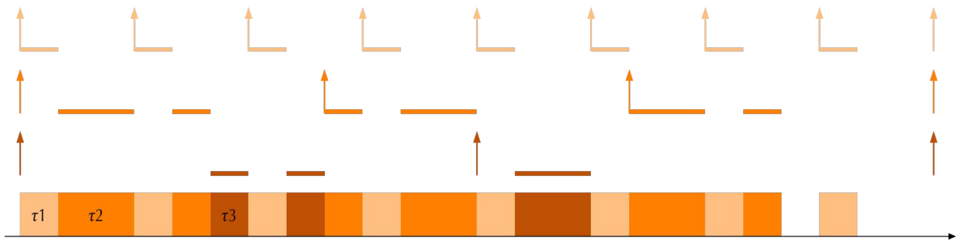
Efficient and predictable, avoids issues such as kernel locks
The master core computes the threads to schedule and fire scheduling order to slave cores

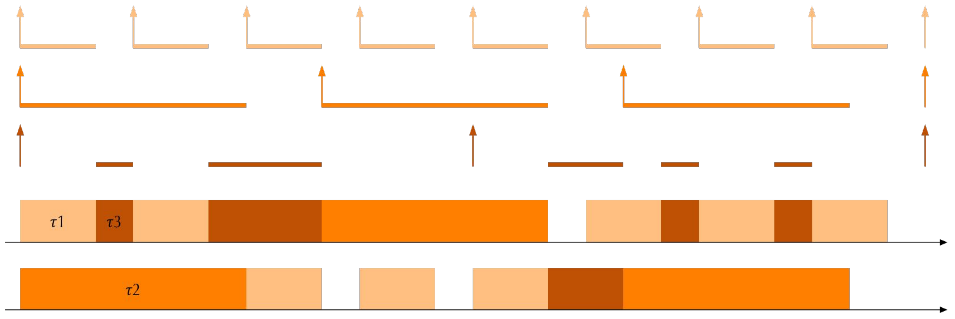
Support of multiple scheduler models

Rate Monotonic (RM) and Earliest Deadline First (EDF)
Partitioned, global, U-EDF (experimental)

Priority inheritance / priority ceiling

Posix compliant API (also for SCOPE_*)





HIPPEROS features for **performance**

- **Performant schedulers**
  High utilisation systems are guaranteed to meet deadlines

- **Fast communication**
  Safe zero-copy IPC channels

- **Fast user-space abstractions**
  - Lightweight Multithreading and support for good users-space API (OpenMP, OpenCV)
  - Fast user-space locks
  - User-level I/O mapping
  - FPGA reconfiguration support
  - Power Management module for energy efficiency

HIPPEROS features for **security**

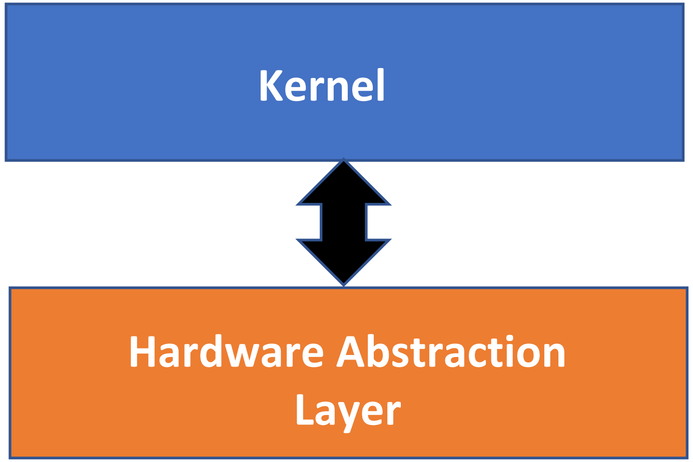
- **The system is designed at compile-time**
  - All processes known at design-time in the task set
  - All processes are selected by the user
- **No system tasks**
  - No hidden processes with special access rights
- **I/Os (ex: network) are disabled by default**
  - The user only needs to prevent unwanted access on I/Os he is actively using
- **Isolated processes through MMU**
  - Least privilege principle

More details on the **kernel software architecture**

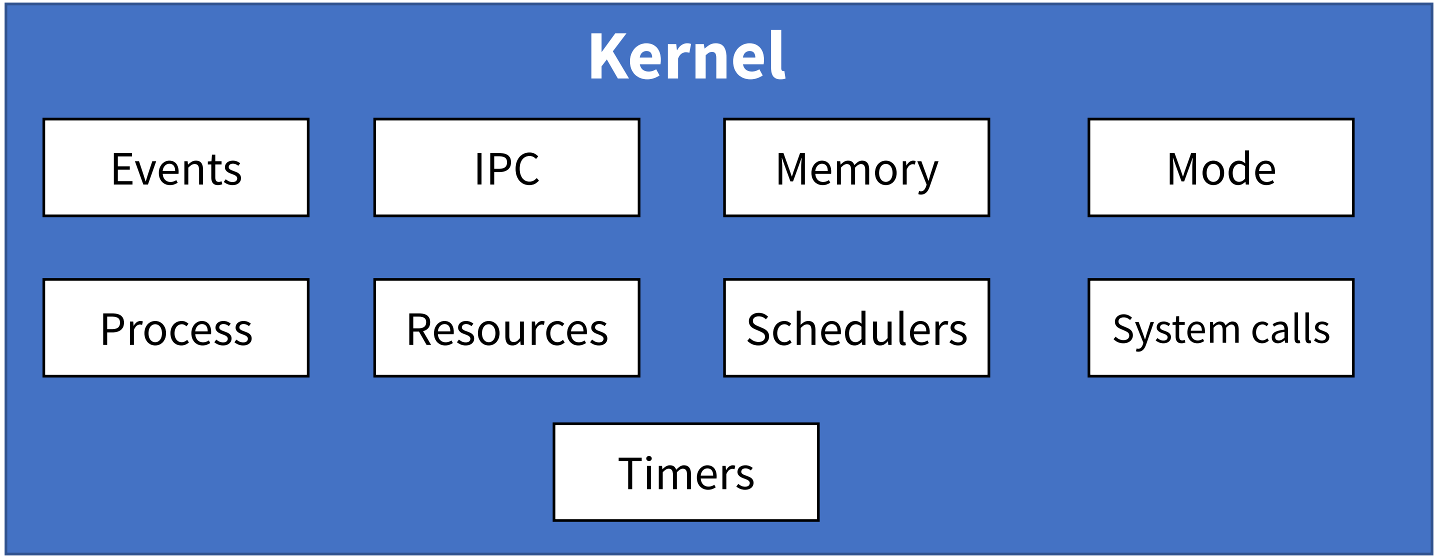
Well designed API between modules

- Allows to replace/choose components that implement the same API transparently

- Ease the design and transfer of information among developers

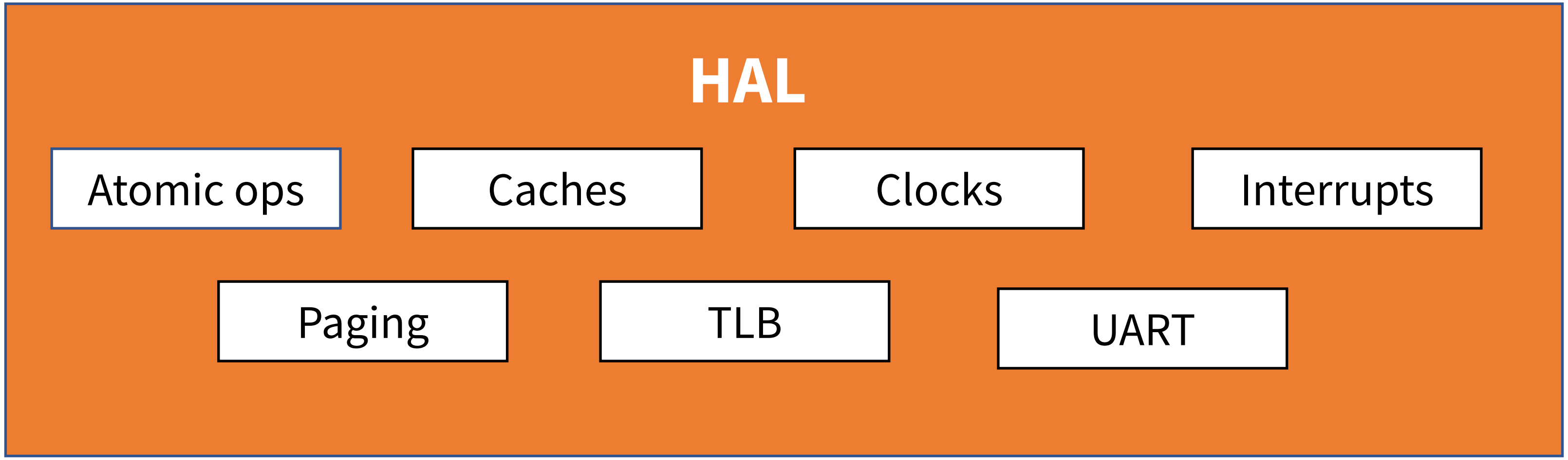


Kernel modules



- Written in C
- Several variants of different modules (e.g. schedulers, resources)

HAL modules



- Written in C & Assembly
- The HAL must be written from scratch for each architecture; some modules are platform-specific (e.g. UART)

HAL main features

The HAL provides:

- Entry points
    - Boot
    - System calls
    - Exceptions
    - Interrupts

- Inter-Processor Interrupts (IPIs)

Advantages of splitting Kernel and HAL

Easily port to new boards/architectures (only need to re-write the HAL)

Test the kernel without compiling the HAL (~ unit tests)

Test the HAL without compiling the kernel (~ platform tests)

Testing everything is still possible (~ integration tests and user tests)

Asymmetric kernel design [2]

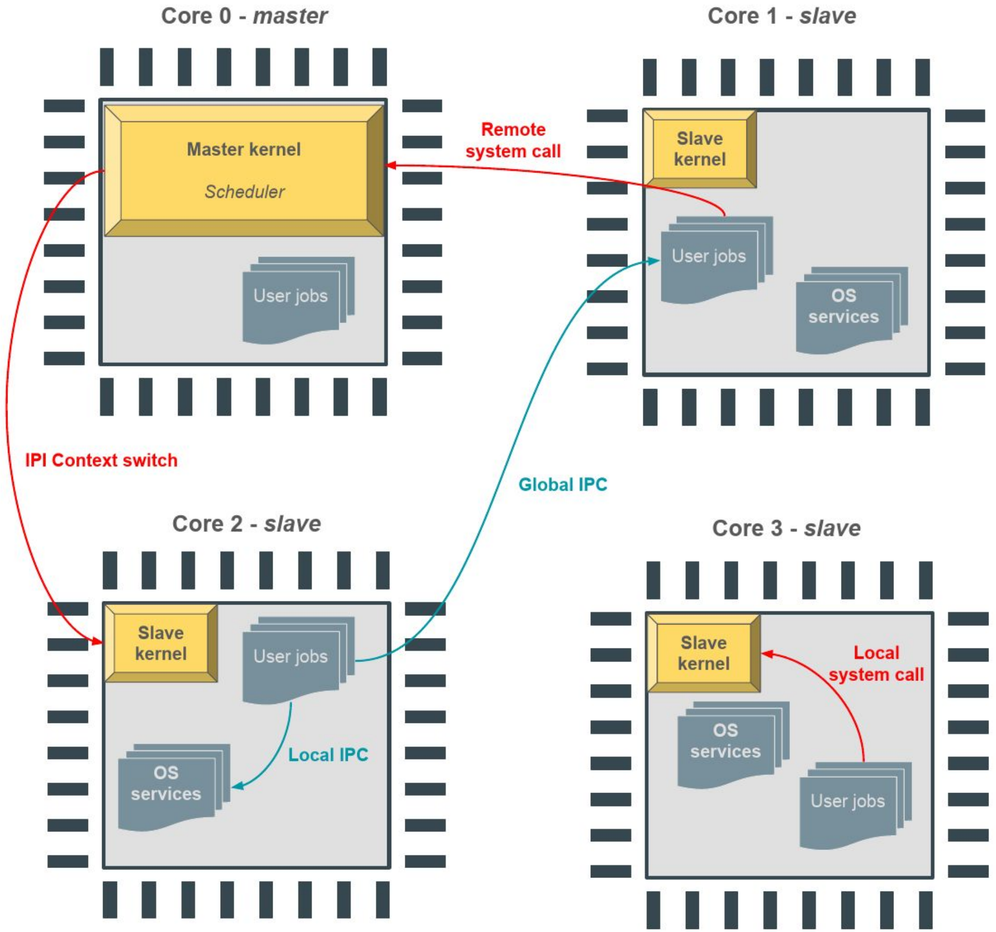
Master kernel

- Handles system events
- Takes scheduling decisions
- Changes thread states
- Executes remote system calls

Slave kernel (really small)

- Executes master decision (context switches)
- Executes local system calls
- Forwards remote system calls to the master

[2] Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joel Goossens, Ben Rodriguez. *A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms.* ECRTS, OSPERT, Lund, Sweden, July 2015.


Core 0 - *master*
Master kernel
Scheduler
User jobs
Core 1 - *slave*
Slave kernel
User jobs
OS services
Remote system call
IPI Context switch
Global IPC
Core 2 - *slave*
Slave kernel
User jobs
OS services
Local IPC
Core 3 - *slave*
Slave kernel
OS services
User jobs
Local system call

Advantages of asymmetric design

Ease of design

Facilitate onboarding, debugging, etc.

No complex network of kernel spinlocks to manage

Scalable solution [Cerqueira, 2014]

“Message passing” approach, less cache bouncing and less peak contention

Fine-grained IPC management and clustered approach to scale up (>10 cores)

[Cerqueira, 2014] F. Cerqueira, M. Vanga, and B. Brandenburg. *Scaling Global Scheduling with Message Passing.* RTAS 2O14, Berlin, Germany.

Kernel execution

Kernel code is executed:

- Boot-time
- Interrupts/system calls → ISR handler

The kernel is

- Interruptible
- **Not** preemptible

Memory models

3 memory models

- **No MMU** (internal use)
- **Single Page Table**: one page table for the whole system
    - Used when the kernel and the user tasks are statically linked together
- **Independent**: one page table per process (full isolation)

Inter-Process Communication

3 types of process-level communication

- Synchronous IPC
  - Shared-memory page
  - The kernel protects the page from unwanted access during writes
  - HIPPEROS native API
  - 0 copy
  - Ideal for 1-to-1 communication (large data transfer, e.g. SDIO drivers and file systems)
- Kernel buffers
  - “Channels” (= local network ports) for the server to bind and waits
  - Inspired by UDP communication
  - Ideal for small control messages and many-to-one communication

User-space device drivers

- Access device physical memory
  - **`mmap()`** system call (extended)
  - Granted to tasks marked as “system” (task set)

- Can be packaged as a library (as opposed to isolated tasks) for small systems

- Interrupt registration
  - **`irq_wait(size_t id)`** system call (native HIPPEROS API)
  - Queuing of threads to processor interrupt lines
  - Thread is blocked and rescheduled when the IRQ is fired
  - IRQ is masked until next `irq_wait()` call is issued
  - Thread must disable the interrupt source (e.g. driver)

Agenda

1. The human story behind the RTOS
2. HIPPEROS high level features
3. Run-time model & build environment
4. Architecture overview and main design choices
5. **Research results achieved**
6. Conclusions and bright future

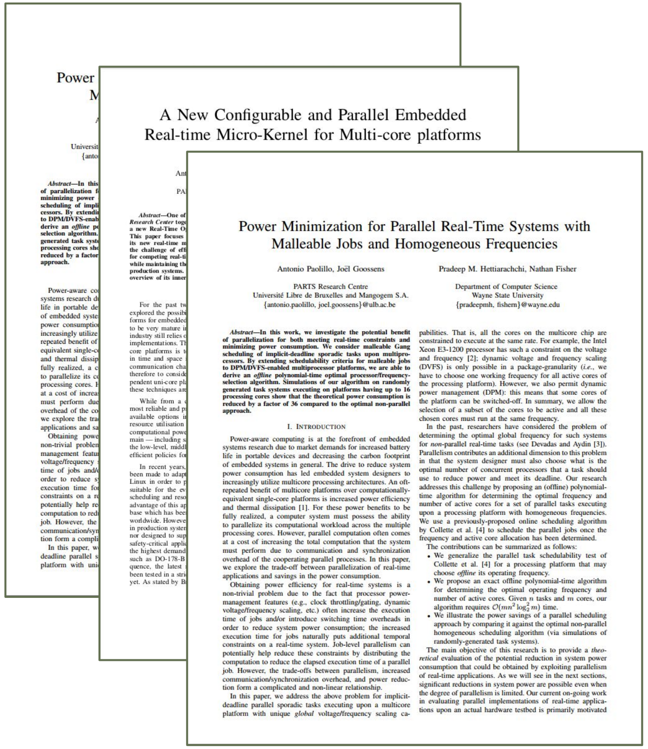
Research and Development

Importance of publishing results

Presence in academia

Results on:

- power management
- mixed-criticality scheduling
- memory centric scheduling
- ...
- Some latest developments



A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms

Power Minimization for Parallel Real-Time Systems with Malleable Jobs and Homogeneous Frequencies

Antonio Paolillo, Joël Goossens
PARTS Research Centre
Université Libre de Bruxelles and Mangogem S.A.
{antonio.paolillo, joel.goossens}@ulb.ac.be

Pradeep M. Hettiarachchi, Nathan Fisher
Department of Computer Science
Wayne State University
{pradeepmh, fishern}@wayne.edu

***Abstract*—In this work, we investigate the potential benefit of parallelization for both meeting real-time constraints and minimizing power consumption. We consider malleable Gang scheduling of implicit-deadline sporadic tasks upon multiprocessors. By extending schedulability criteria for malleable jobs to DPM/DVFS-enabled multiprocessor platforms, we are able to derive an *offline* polynomial-time optimal processor/frequency-selection algorithm. Simulations of our algorithm on randomly generated task systems executing on platforms having up to 16 processing cores show that the theoretical power consumption is reduced by a factor of 36 compared to the optimal non-parallel approach.**

I. INTRODUCTION

Power-aware computing is at the forefront of embedded systems research due to market demands for increased battery life in portable devices and decreasing the carbon footprint of embedded systems in general. The drive to reduce system power consumption has led embedded system designers to increasingly utilize multicore processing architectures. An oft-repeated benefit of multicore platforms over computationally-equivalent single-core platforms is increased power efficiency and thermal dissipation [1]. For these power benefits to be fully realized, a computer system must possess the ability to parallelize its computational workload across the multiple processing cores. However, parallel computation often comes at a cost of increasing the total computation that the system must perform due to communication and synchronization overhead of the cooperating parallel processes. In this paper, we explore the trade-off between parallelization of real-time applications and savings in the power consumption.

Obtaining power efficiency for real-time systems is a non-trivial problem due to the fact that processor power-management features (e.g., clock throttling/gating, dynamic voltage/frequency scaling, etc.) often increase the execution time of jobs and/or introduce switching time overheads in order to reduce system power consumption; the increased execution time for jobs naturally puts additional temporal constraints on a real-time system. Job-level parallelism can potentially help reduce these constraints by distributing the computation to reduce the elapsed execution time of a parallel job. However, the trade-offs between parallelism, increased communication/synchronization overhead, and power reduction form a complicated and non-linear relationship.

In this paper, we address the above problem for implicit-deadline parallel sporadic tasks executing upon a multicore platform with unique *global* voltage/frequency scaling capabilities. That is, all the cores on the multicore chip are constrained to execute at the same rate. For example, the Intel Xeon E3-1200 processor has such a constraint on the voltage and frequency [2]; dynamic voltage and frequency scaling (DVFS) is only possible in a package-granularity (*i.e.*, we have to choose one working frequency for all active cores of the processing platform). However, we also permit dynamic power management (DPM): this means that some cores of the platform can be switched-off. In summary, we allow the selection of a subset of the cores to be active and all these chosen cores must run at the same frequency.

In the past, researchers have considered the problem of determining the optimal global frequency for such systems for *non*-parallel real-time tasks (see Devadas and Aydin [3]). Parallelism contributes an additional dimension to this problem in that the system designer must also choose what is the optimal number of concurrent processors that a task should use to reduce power and meet its deadline. Our research addresses this challenge by proposing an (offline) polynomial-time algorithm for determining the optimal frequency and number of active cores for a set of parallel tasks executing upon a processing platform with homogeneous frequencies. We use a previously-proposed online scheduling algorithm by Collette et al. [4] to schedule the parallel jobs once the frequency and active core allocation has been determined.

The contributions can be summarized as follows:

- We generalize the parallel task schedulability test of Collette et al. [4] for a processing platform that may choose *offline* its operating frequency.
- We propose an exact offline polynomial-time algorithm for determining the optimal operating frequency and number of active cores. Given n tasks and m cores, our algorithm requires $\mathcal{O}(mn^2 \log_2^2 m)$ time.
- We illustrate the power savings of a parallel scheduling approach by comparing it against the optimal non-parallel homogeneous scheduling algorithm (via simulations of randomly-generated task systems).

The main objective of this research is to provide a *theoretical* evaluation of the potential reduction in system power consumption that could be obtained by exploiting parallelism of real-time applications. As we will see in the next sections, significant reductions in system power are possible even when the degree of parallelism is limited. Our current on-going work in evaluating parallel implementations of real-time applications upon an actual hardware testbed is primarily motivated

Power-aware and parallel scheduling [3]

Combining DVFS/DPM with multi-threading in real-time systems to save more energy

Applications written with OpenMP

Approach validating with a full stack real-world embedded stack

- a MPSoC embedded board
- HIPPEROS RTOS with OpenMP library and real-time scheduler
- Oscilloscope probe to measure energy requirements

[3] Antonio Paolillo, Paul Rodriguez, Nikita Veshchikov, Joël Goossens and Ben Rodriguez. *Quantifying Energy Consumption for Practical Fork-Join Parallelism on an Embedded Real-Time Operating System.* RTNS, Brest, France, October 2016.

Experimental setup

Mixed-criticality scheduling [4]

Process-based (no per-thread criticality)

Mode changes when a WCET overrun occurs

Configurable per-process mode change policy

- Change period, kill, suspend...

“time guards” similar to seL4 temporal exceptions

- Modified WCET overrun mechanism

[4] Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joel Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugère, Philippe Bonnot. *Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system.* WMC, RTSS, Paris, December 2017.

Avoiding memory interferences [5]

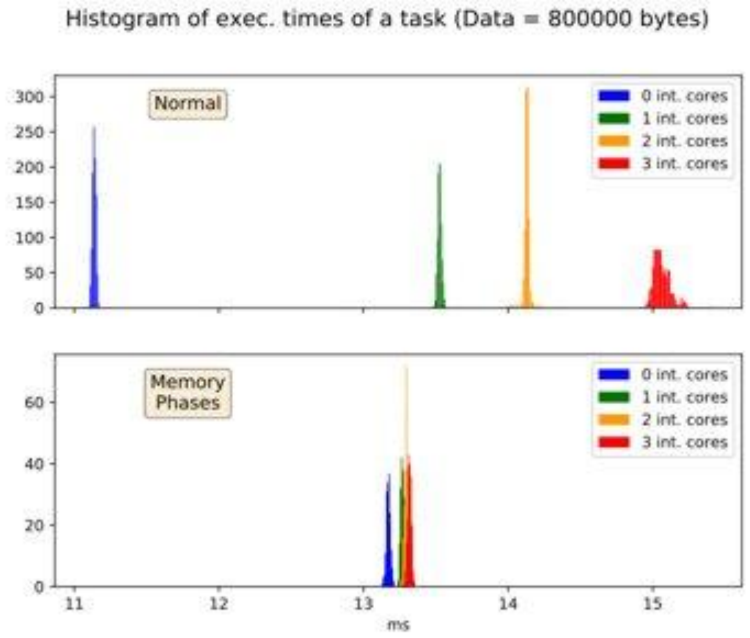
Shield execution time from interferences in the memory sub-system

- **Last-Level Shared Cache:** partition among cores

- **Memory Bus:** concept of exclusive memory phases to prefetch/write-back data. Execution then does not suffer from LLC cache misses → no opportunities for contention in memory bus

[5] Juan M Rivas, Joël Goossens, Xavier Poczekajlo and Antonio Paolillo. *Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems.* ECRTS, Stuttgart, Germany, July 2019.

Experiments on Cortex-A9 Quad Core 1MB shared L2 cache [5]



- Overheads due to memory phases are lower than inflation due to interferences

- Execution time not as affected by number of interfering cores

[5] Juan M Rivas, Joël Goossens, Xavier Poczekajlo and Antonio Paolillo. *Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems.* ECRTS, Stuttgart, Germany, July 2019.

Fast user-space locks

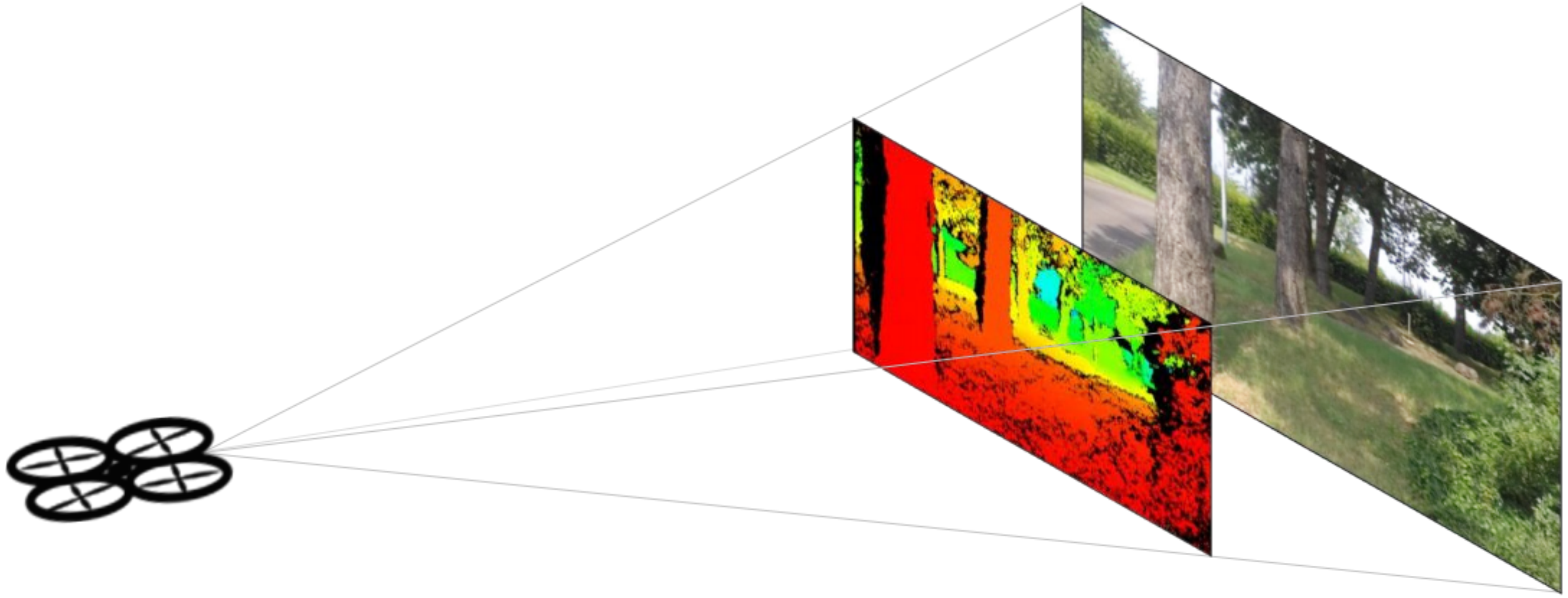
Futex-based synchronisation primitives

No system calls if no contention

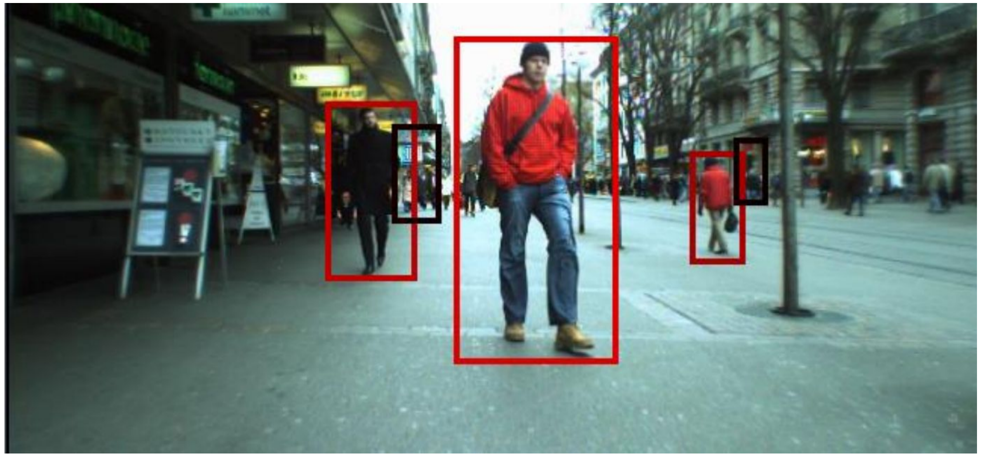
→ greatly reduces the locks overhead

All user locking primitives are built on top of futexes

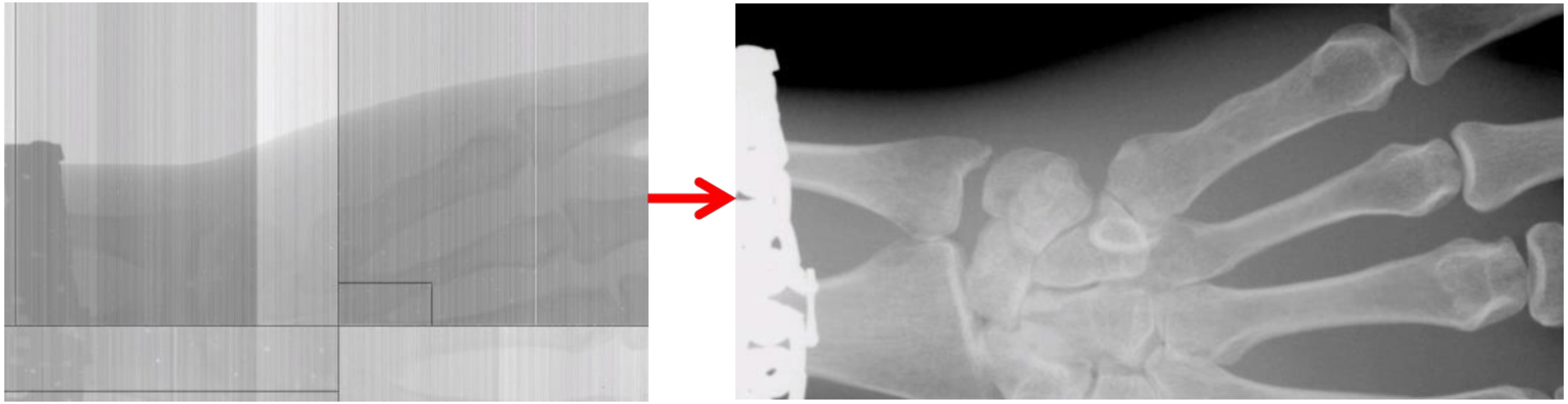
Application use cases - UAV



Application use cases - automotive

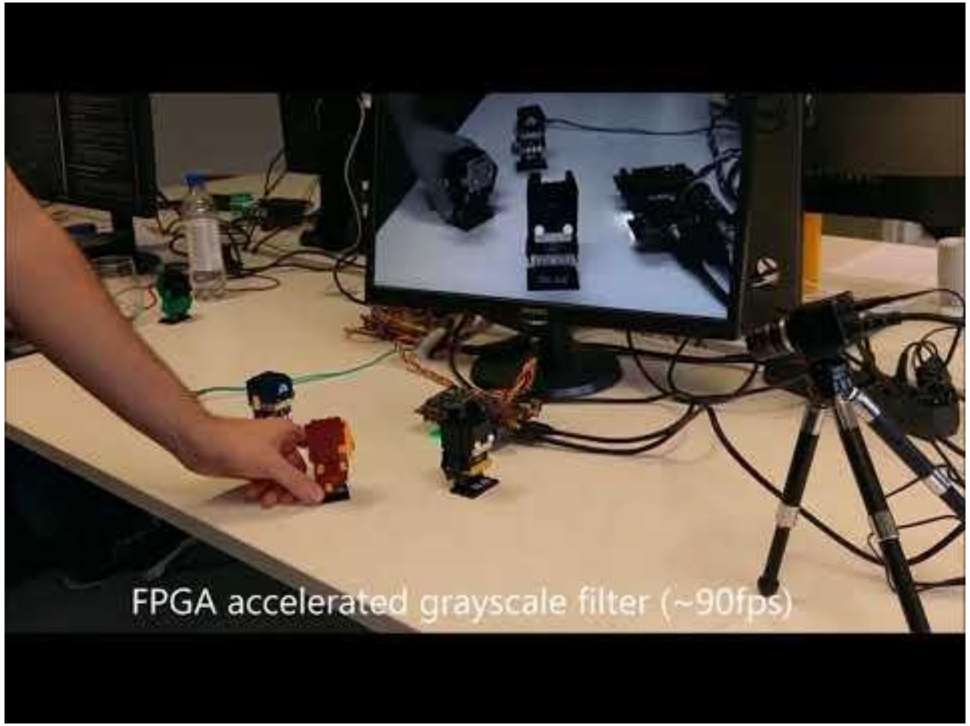


Application use cases - medical



Application use cases - FPGA filters

https://youtu.be/734qxpzKi4A



Agenda

1. The human story behind the RTOS
2. HIPPEROS high level features
3. Run-time model & build environment
4. Architecture overview and main design choices
5. Research results achieved
6. **Conclusions and bright future**

Summary: the HIPPEROS RTOS

HIPPEROS is a RTOS targeting modern high-end embedded platforms, exhibiting heterogeneous parallelism (multi-core, FPGA, etc.)

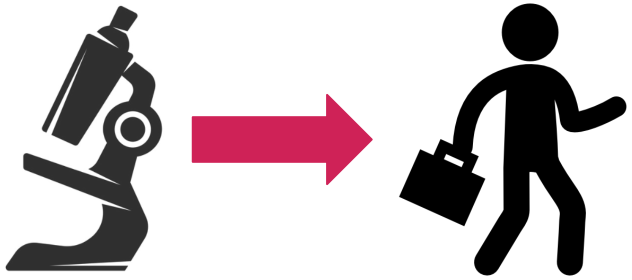
The RTOS is configurable and tailorable to system designers and application developers requirements (e.g. choose of scheduling policy, memory model, etc.)

Moreover, it provides predictability and reliability to the application layer with strict real-time scheduling and associated monitoring, space isolation through virtual memory

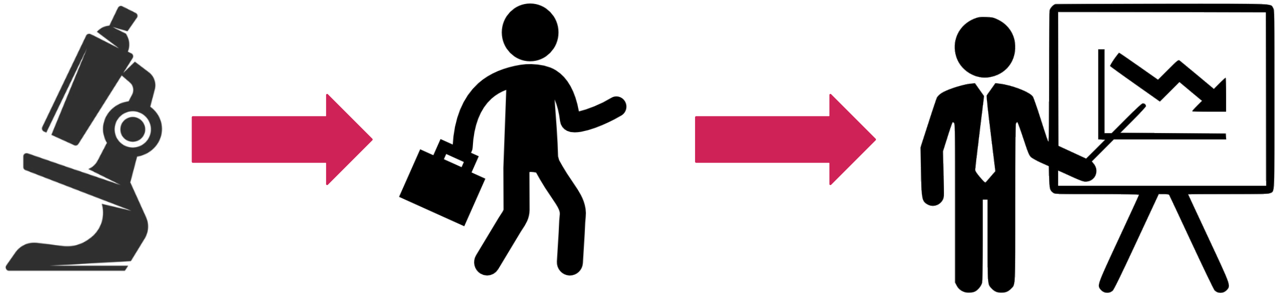
It may fits the requirements of an “Adaptive AUTOSAR” RTOS

A side goal is to validate research results and stay in touch with academia

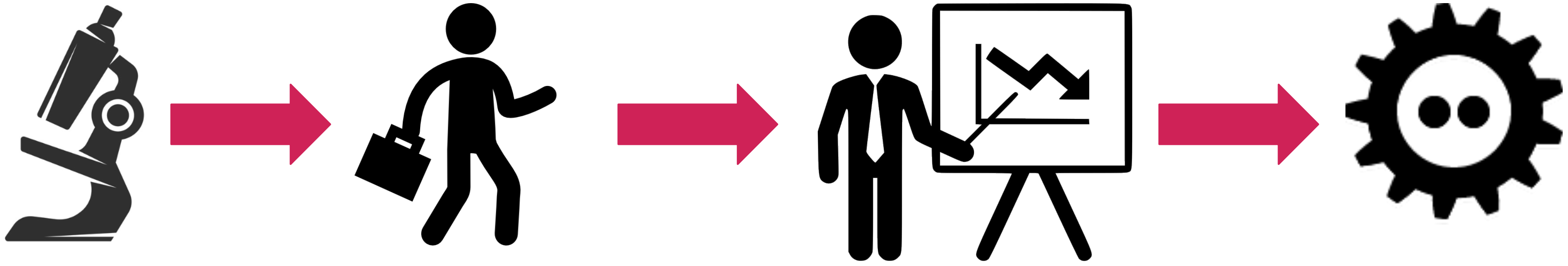
Initially intended fully proprietary...



Company landed in 2019 for lack of funding



New opportunity: open source



Future work

- The kernel & OS, initially proprietary, will be opened
- Dual licensing scheme:
  - Free & open-source for academic & non-commercial use
  - Licensed for commercial use, with MangoGem company
- Playground for experiments, new idea or algorithms / policies / ...
  - Real-time community (alternative to linux)
  - Others (e.g. micro-kernel design space exploration, SMP, power-management, ...)

Open questions

- What license?
  - I’m no IP expert
  - Open to idea / suggestion / discussion in the room

- What model for external contributions?
  - Will likely be *only* external

- Agenda
  - Code base cleanup
  - Define the license & the contribution model
  - Create a landing page
  - Open the project on Github… and celebrate :-)

Resources

[1] My Ph.D. thesis, a lot of HIPPEROS trivia are explained, publicly available
https://antonio.paolillo.be/research.html
https://antonio.paolillo.be/publications.html

[2] An introductory paper to HIPPEROS micro-kernel design (~3 pages)
https://people.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p25.pdf

[3] A paper about low-power scheduling on HIPPEROS (~12 pages)
http://dx.doi.org/10.1145/2997465.2997473

[4] A paper about using mixed-criticality scheduling in a real RTOS (~6 pages)
https://github.com/CPS-research-group/WMC2017/raw/master/papers/1.pdf

[5] Recent paper on predictable real-time memory scheduling (ECRTS'19),
using HIPPEROS RTOS as a validation platform (~23 pages)
http://drops.dagstuhl.de/opus/volltexte/2019/10744/

Thanks!

Q?