Getting Started with the Quest RTOS & Quest-V Partitioning Hypervisor

Dr. Richard West
Professor, Boston University
Chief Software Architect, Drako Motors

Shriram Raja
Boston University

Zhiyuan Ruan
Boston University

Rafiuddin Syed
Senior Kernel Developer
Drako Motors

Goals

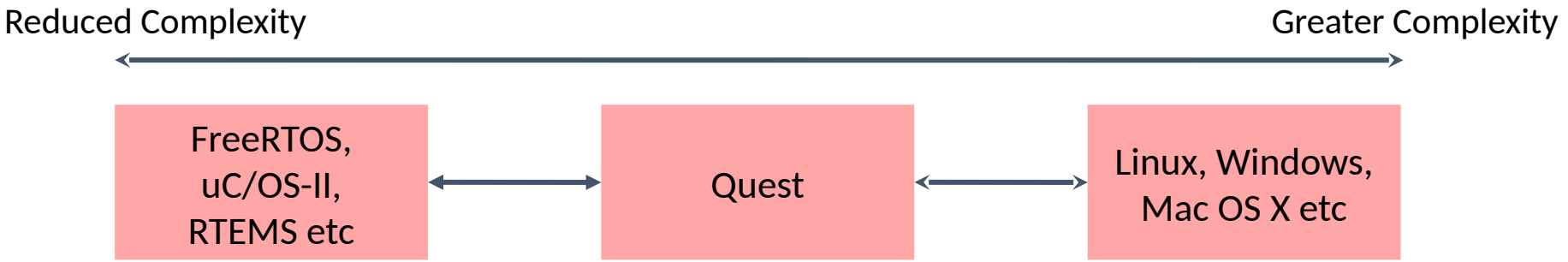
- High-confidence (embedded) systems
  - **Mixed criticalities** – timeliness and safety
  - **Predictable** – real-time support
  - **Secure** – resistant to component failures & malicious attacks
  - **Fault tolerant** – online recovery from soft errors and timing violations

Target Applications

- Healthcare
- Avionics
- Automotive
- Factory automation
- Robotics
- Space exploration
- Internet-of-Things (IoT)
- Industry 4.0 “smart factories”

In the Beginning…Quest

- Open source (GPL v3), GRUB bootable either with legacy or EFI firmware
- Initially a “small” RTOS
- ~30KB ROM image for uniprocessor version
- Page-based address spaces
- Kernel threads (simple POSIX implementation)
- Dual-mode kernel-user separation
- Real-time Virtual CPU (VCPU) task and interrupt scheduling
- Later SMP support (defaults up to 8 cores, expandable to 256 or higher)
- LAPIC timing
- Semaphores and spinlocks
- Tuned pipes
- Real-time USB 2 (EHCI) and 3.x (xHCI) stack

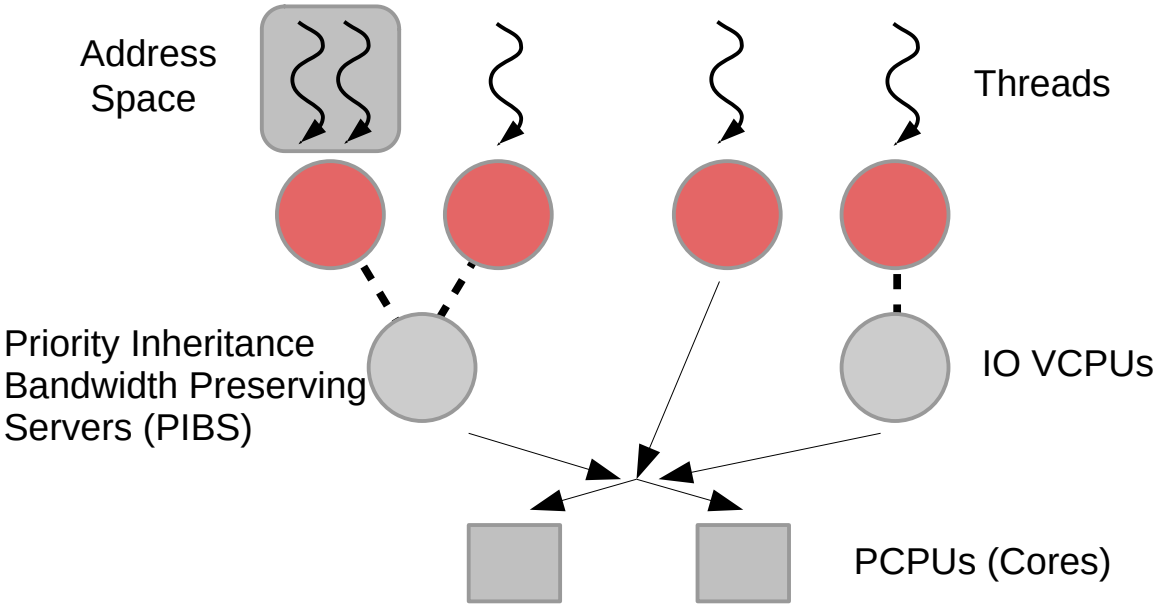


Quest Virtual CPUs (RTAS'11)

VCPUs are first-class entities within the RTOS

- Budgeted real-time execution of threads and interrupts

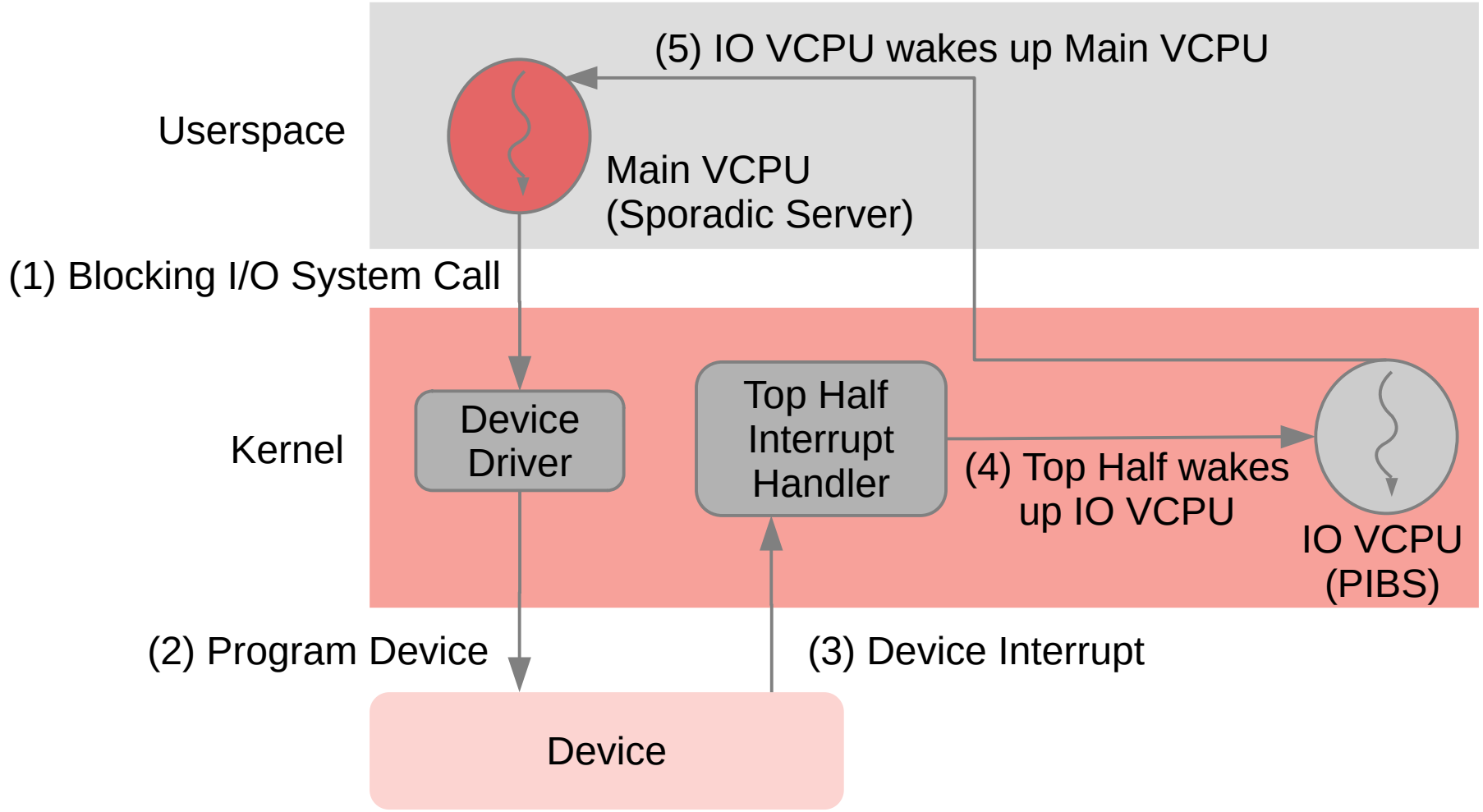
- Tasks → Main VCPUs (Sporadic servers: budget & period)
- Interrupts → IO VCPUs (PIBS: derive budget & period from Main VCPU)



Real-time IO

IOVCPU_CLASS_USB
IOVCPU_CLASS_NET
IOVCPU_CLASS_GPIO
...

VCPU Control Flow



Sporadic Server (SS) Scheduling

- Model periodic tasks
- Each SS has a pair (C,T) s.t. a server is guaranteed C CPU cycles every period of T cycles when runnable
  - Guarantee applied at foreground priority
  - background priority when budget depleted
- Rate-Monotonic Scheduling theory applies

PIBS Scheduling

- IO VCPUs have utilization factor, $U_{V,IO}$
- IO VCPUs inherit priorities of tasks (or Main VCPUs) associated with IO events
  - Priorities are based on periods of corresponding Main VCPU
  - IO VCPU budget is limited to:
    - $T_{V,Main}$ x $U_{V,IO}$ for period $T_{V,Main}$

PIBS Scheduling

- IO VCPUs have eligibility times, when they can execute
- $t_e = t + C_{actual} / U_{V,IO}$
  - t = start of latest execution
  - t >= previous eligibility time

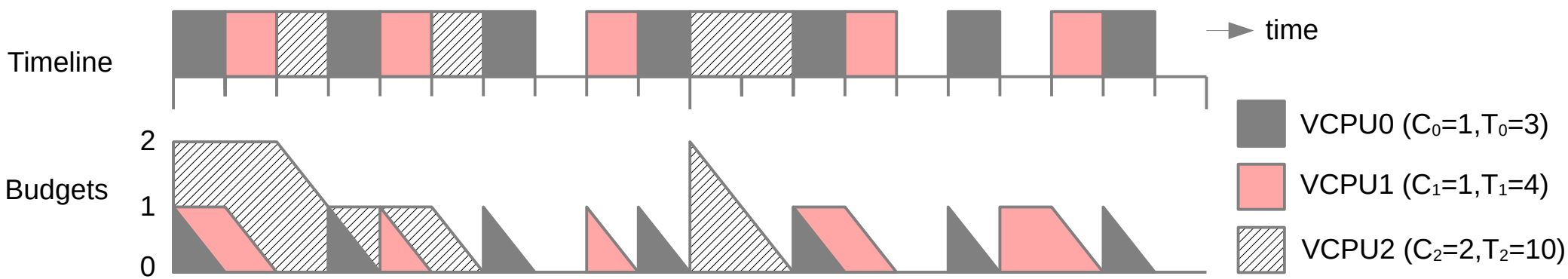
VCPU Scheduling

- Sandbox with 1 PCPU, n Main VCPUs (SS), and m IO VCPUs (PIBS)
- Ci = Budget Capacity of Main VCPU, Vi
- Ti = Replenishment Period of Vi
- Uj = Utilization factor for I/O VCPU, Vj

- Utilization bound feasibility test (with rate-monotonic scheduling of VCPUs):

$$\sum_{i=0}^{n-1} \frac{Ci}{Ti} + \sum_{j=0}^{m-1} (2 - Uj) \cdot Uj \leq n \cdot (\sqrt[n]{2} - 1)$$

Example VCPU Schedule



Quest USB Stack

- Supports EHCI, xHCI
  - Supports xDBC for host-to-host communication

- Real Time Capability
  - USB 2 (EHCI) & 3 (xHCI) Scheduling
  - Differentiated Service of Interrupts

From Quest to Quest-V

- Quest-V for multi-/many-core processors
  - Distributed system on a chip
  - Uses Intel VT-x capabilities found on PCs and SBCs/SoCs:
    - Galileo, MinnowBoard, Edison, Joule, Intel Aero, Up boards, Intel Automotive SoC (Malibou Lake),…
  - Separate sandbox kernels for system components
  - Memory isolation using hardware-assisted memory virtualization
  - Also CPU, I/O, cache partitioning
  - Focus on safety, efficiency, predictability + security
- Supports symbiotic union between Quest RTOS and other legacy systems such as Linux or Android

Quest-V Related Work

- Existing virtualized solutions for resource partitioning
  - Wind River Hypervisor, XtratuM, PikeOS
  - Mentor Graphics Hypervisor
  - Xen
  - Oracle PDOMs
  - IBM LPARs
  - Muen
  - (Siemens) Jailhouse
  - Bao Partitioning Hypervisor

- Quest-V statically allocates resources using a lightweight **partitioning hypervisor** into separate sandboxes
- Sandboxes are tightly coupled as one system of systems into a **separation kernel**

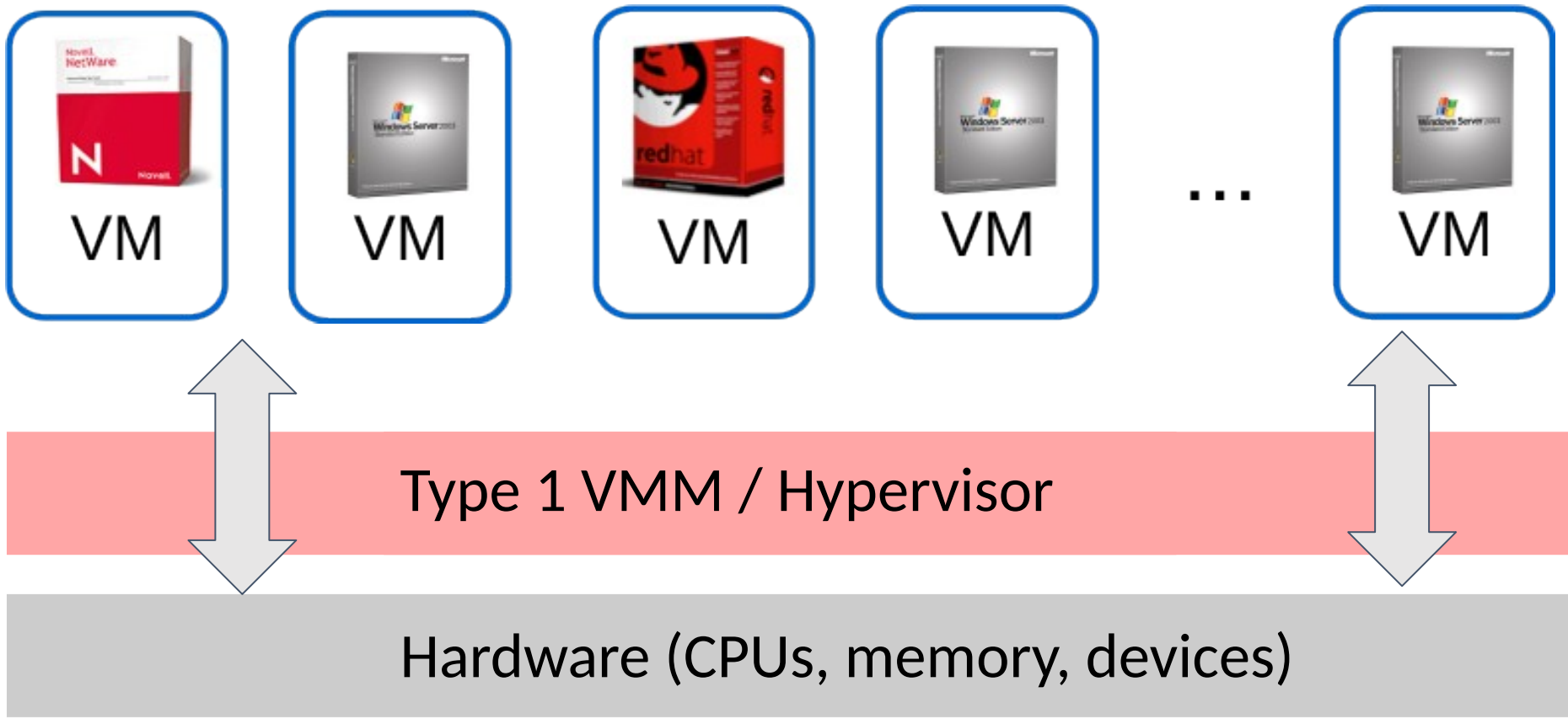
Problem

Traditional Virtual Machine approaches **too expensive**

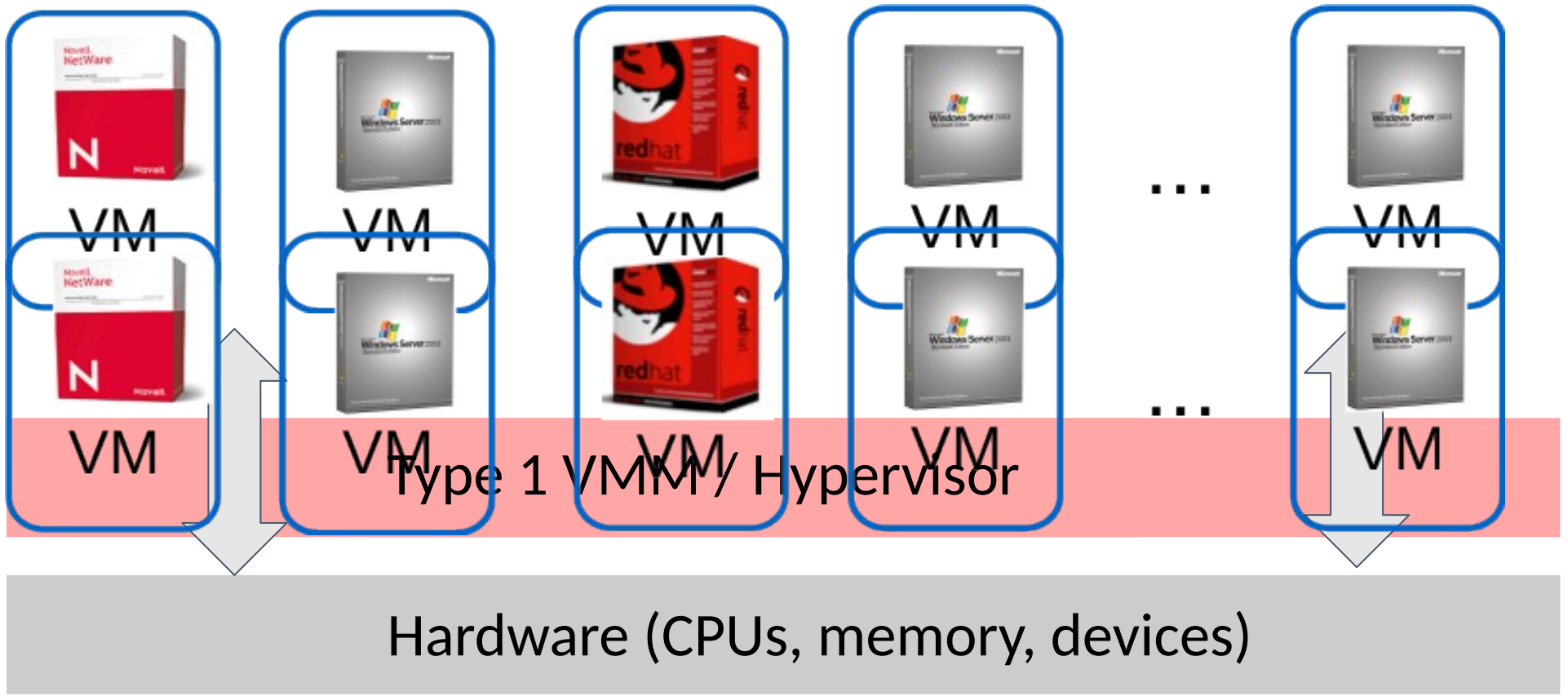
- Require traps to VMM (a.k.a. hypervisor) to mux & manage machine resources for multiple guests
- e.g., ~1500 clock cycles VM-Enter/Exit on Xeon E5506

Traditional Virtual Machine approaches **too memory intensive** for many real-time embedded systems

Traditional Approach (Type 1 VMM)

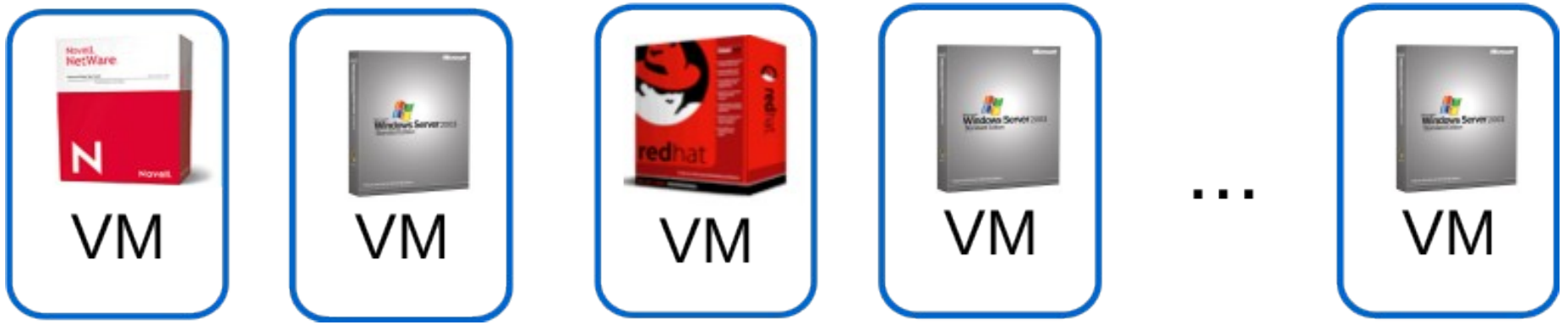


Quest-V Approach



Quest-V Approach

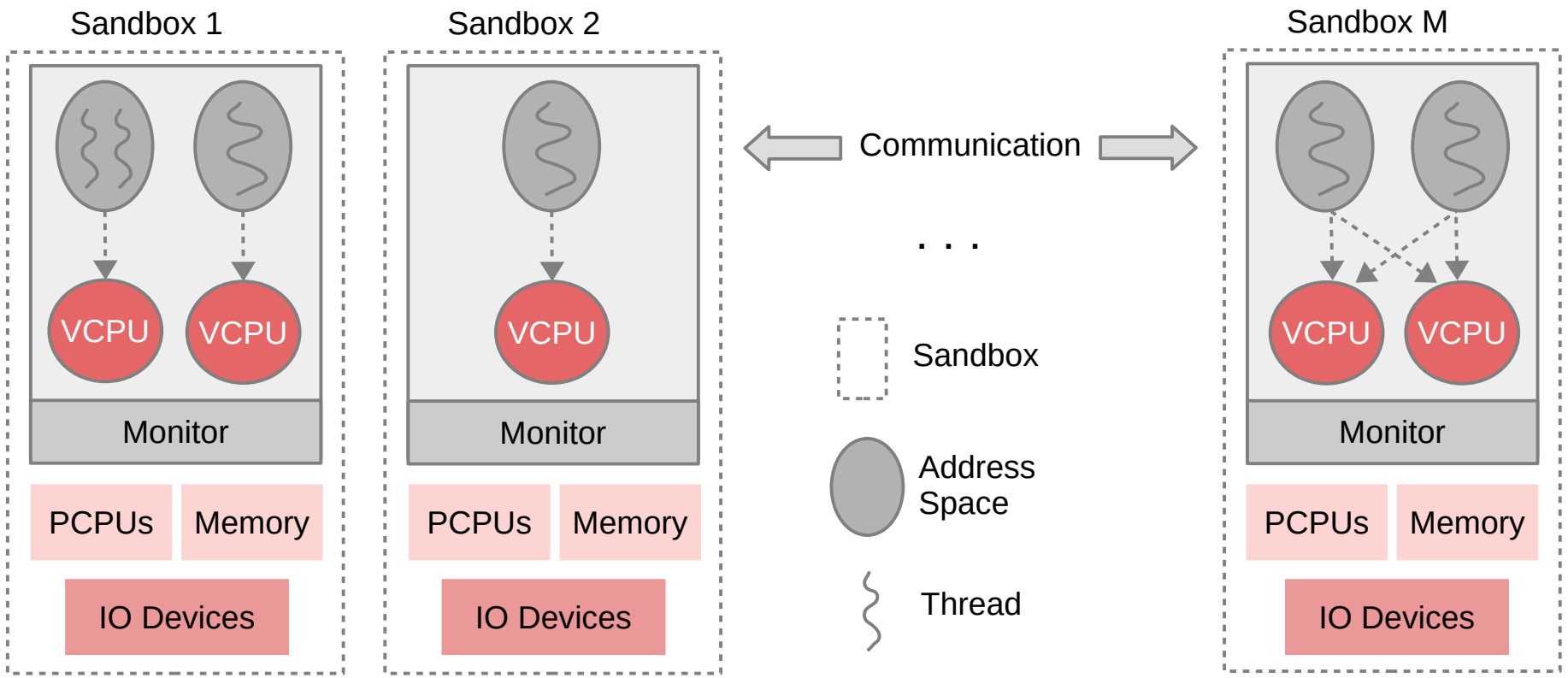
Eliminates hypervisor intervention during normal virtual machine operations



Hardware (CPUs, memory, devices)

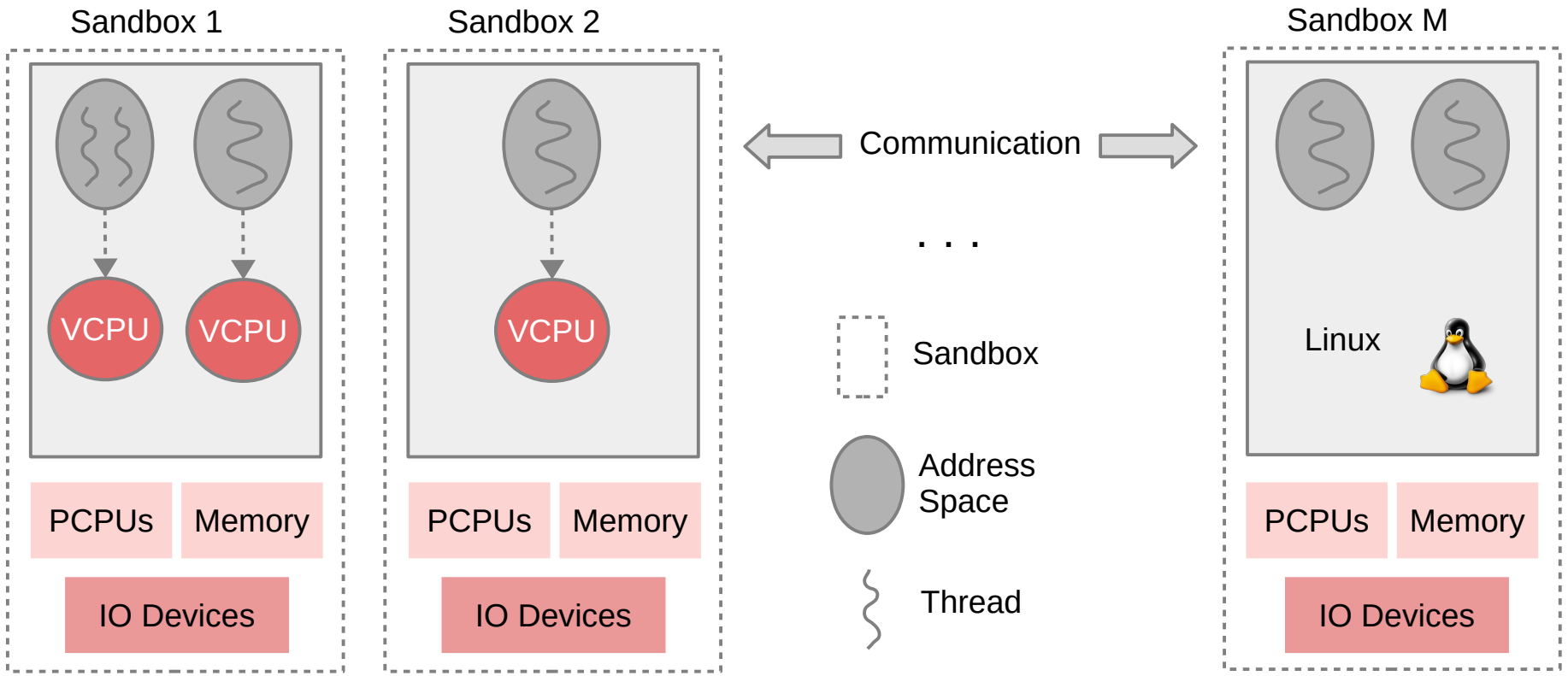
Quest-V Separation Kernel (VEE'14, ACM TOCS'16)

- Monitors partition CPU cores, RAM, I/O devices among sandboxed guests
- Monitors have small trusted compute base – no runtime resource management

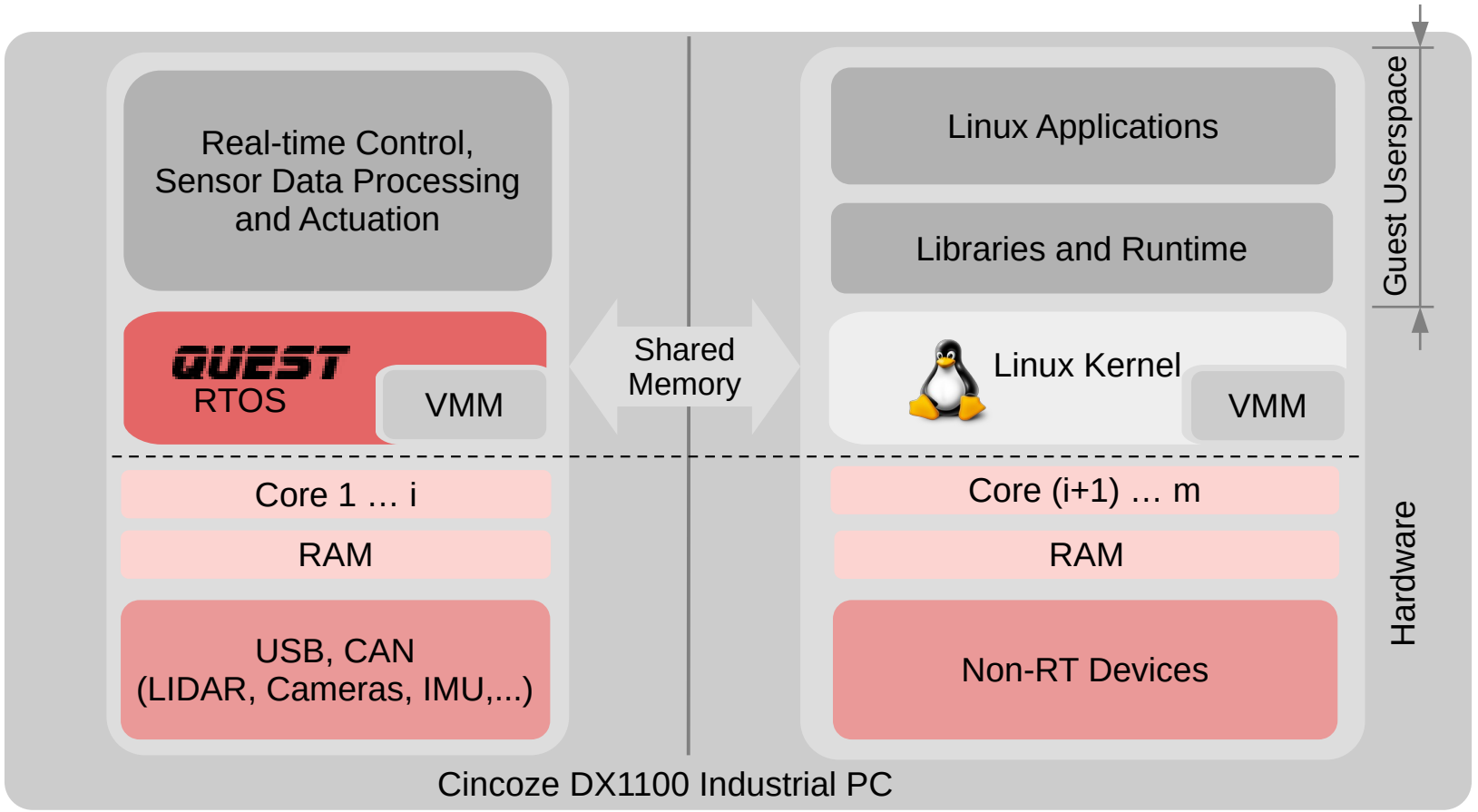


Quest-V Separation Kernel (VEE'14, ACM TOCS'16)

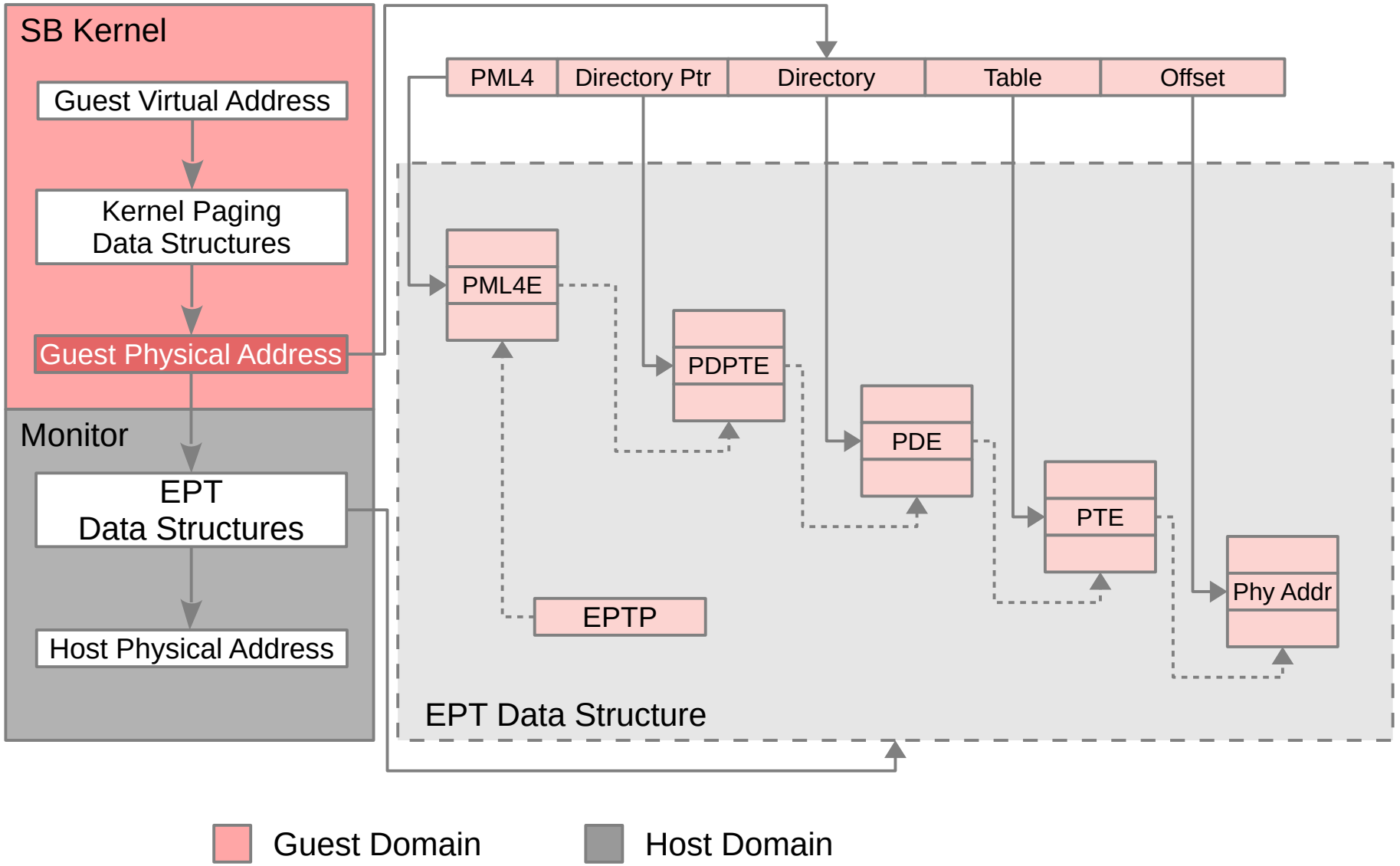
- Partitioning hypervisor – statically partitions resources
- Separation kernel – distributed collection of sandboxed components, indistinguishable from separate private machines for each component



Example: Quest-V for DriveOS

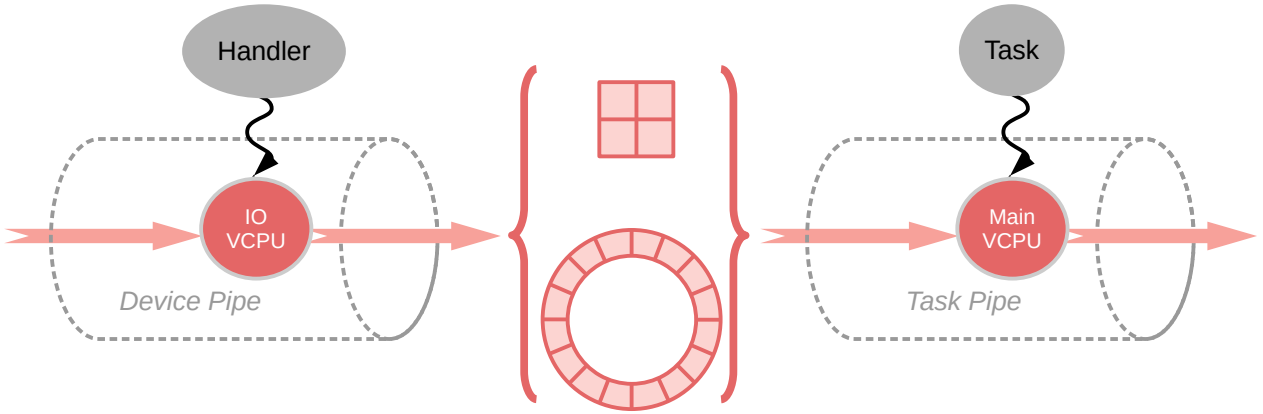


Memory Partitioning



Tuned Pipes (RTSS'18)

- Like POSIX pipes but guarantee throughput and delay on communication
- Simpson's 4-slot (asynchronous) & FIFO (synchronous) buffering

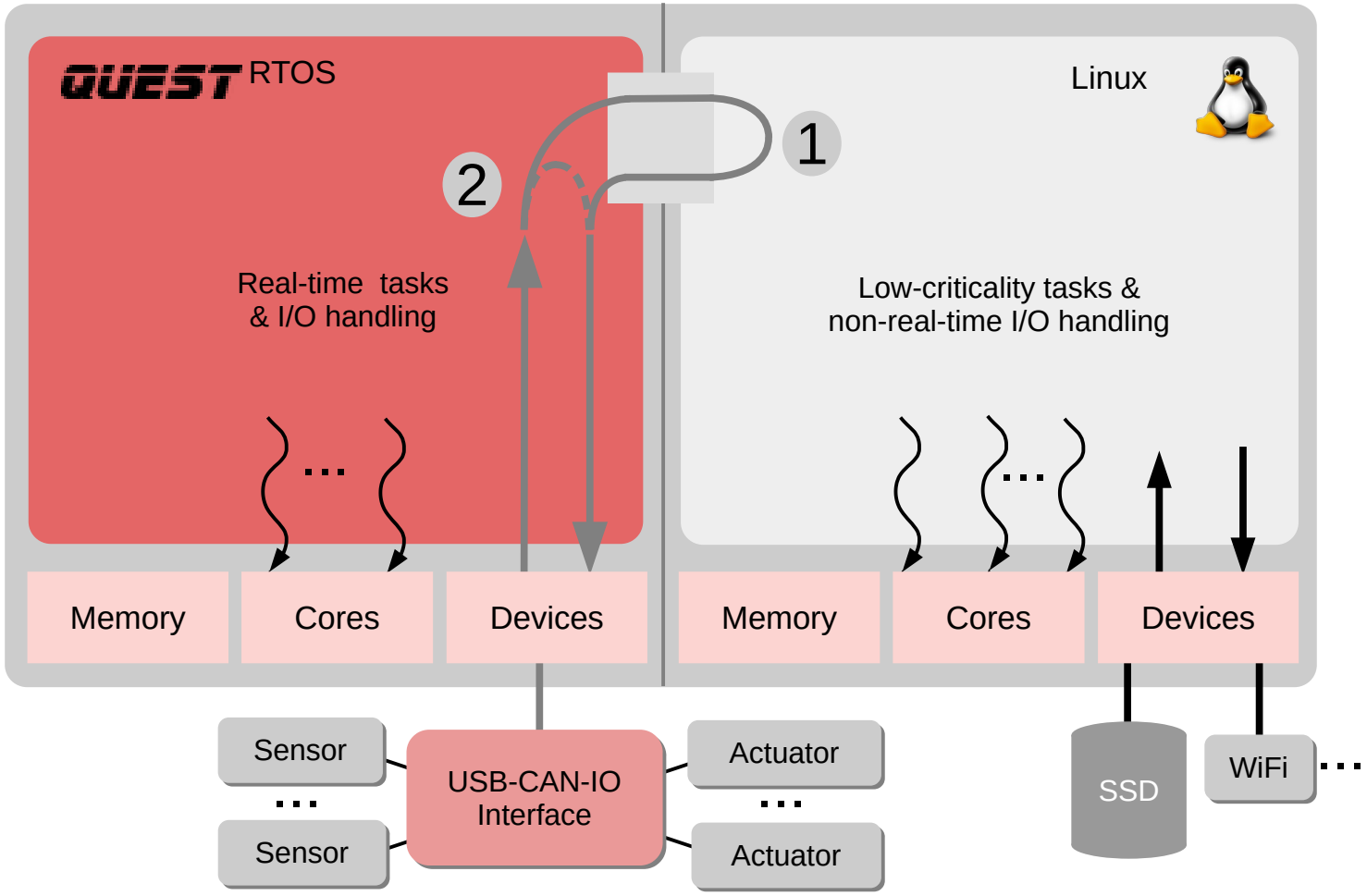


- **Boomerang** I/O subsystem in Quest-V supports real-time pipelines across Quest RTOS and legacy OSes
- Rate match tasks in pipeline to avoid blocking or missed data
- Quest appears as a **real-time virtual device interface** to Linux/Android

Boomerang Inter-OS Task Pipeline Example (RTAS'20)

Boomerang tuned pipe path (1) spans Quest + Linux + USB-CAN-IO

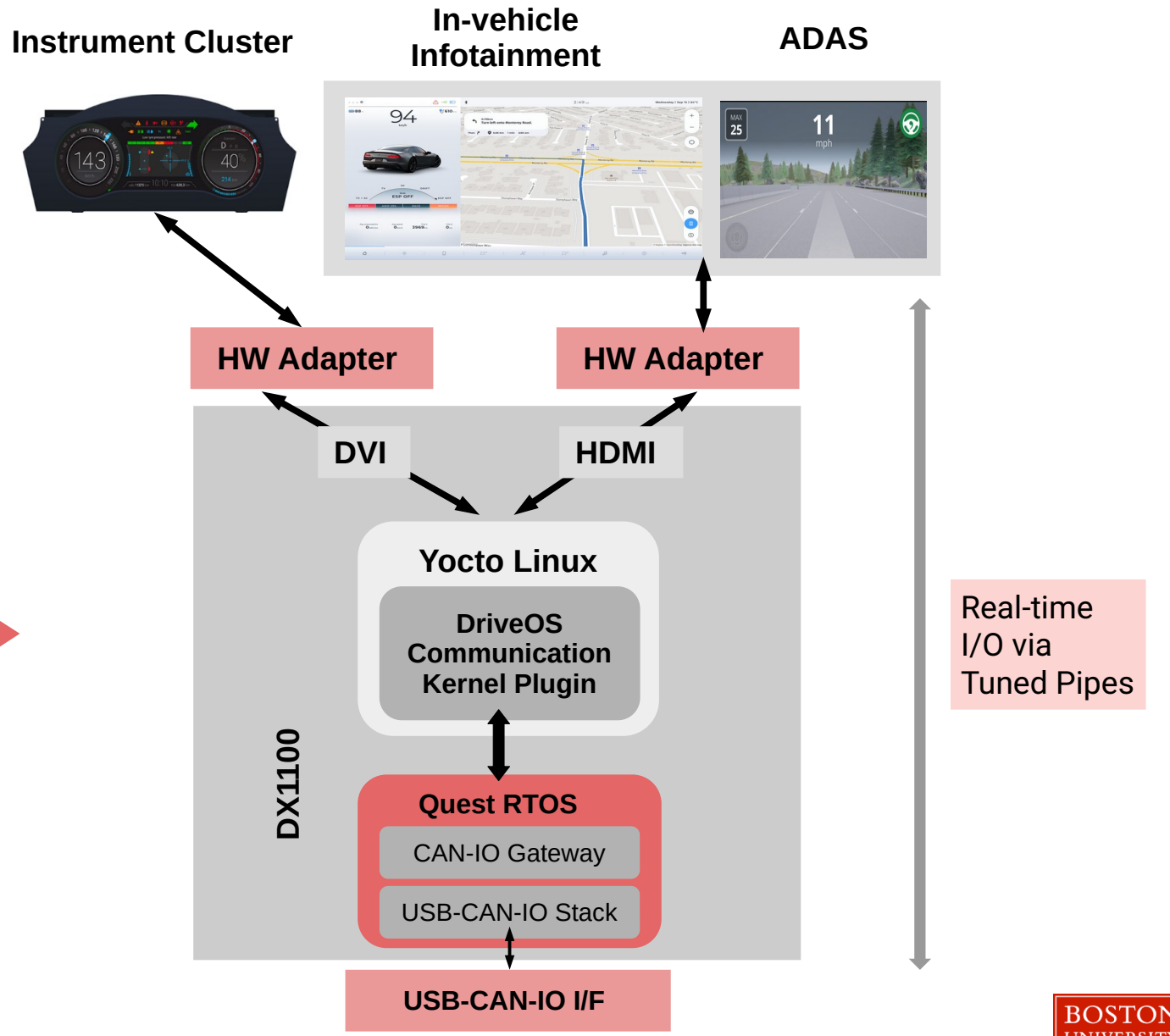
Boomerang tuned pipe path (2) spans Quest + USB-CAN-IO



DriveOS Example (EMSOFT'21)

Map all services to a
single industrial automotive PC

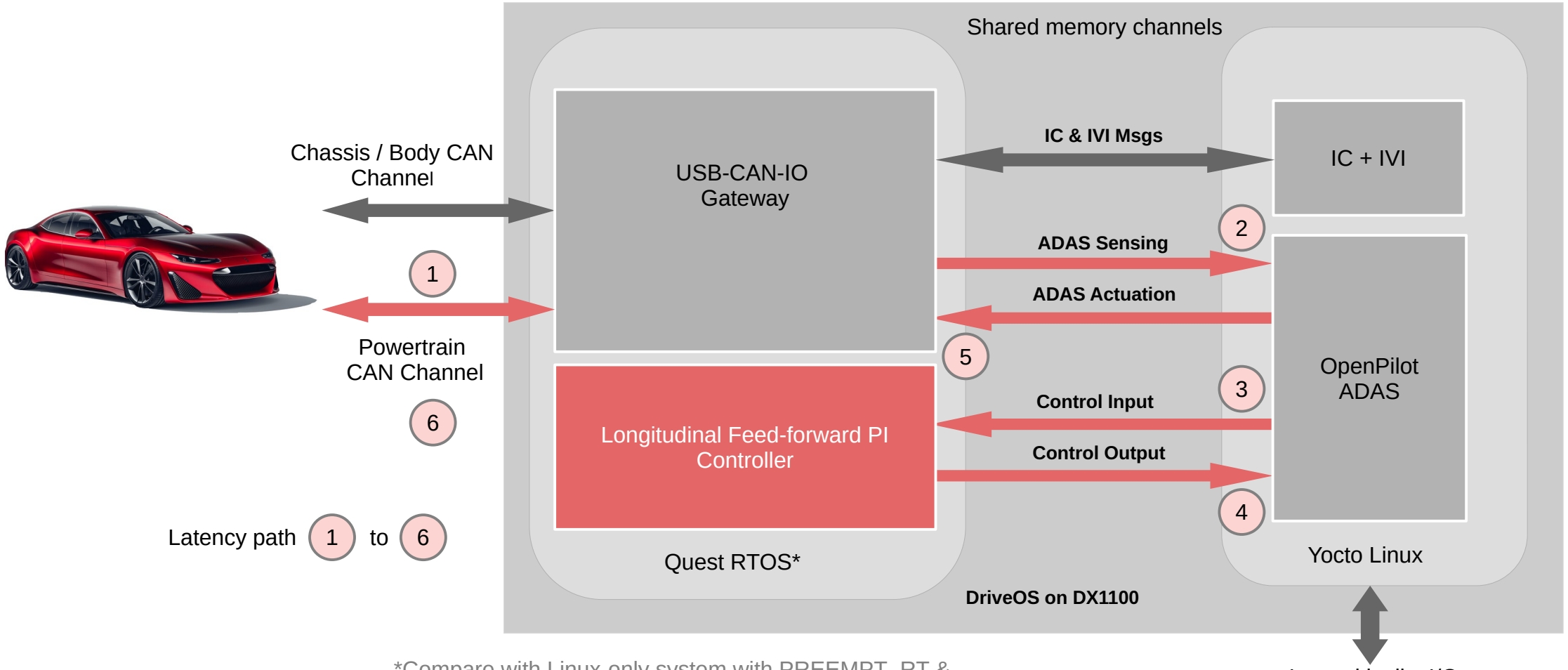
Cincoze DX1100



DRAKO

BOSTON UNIVERSITY

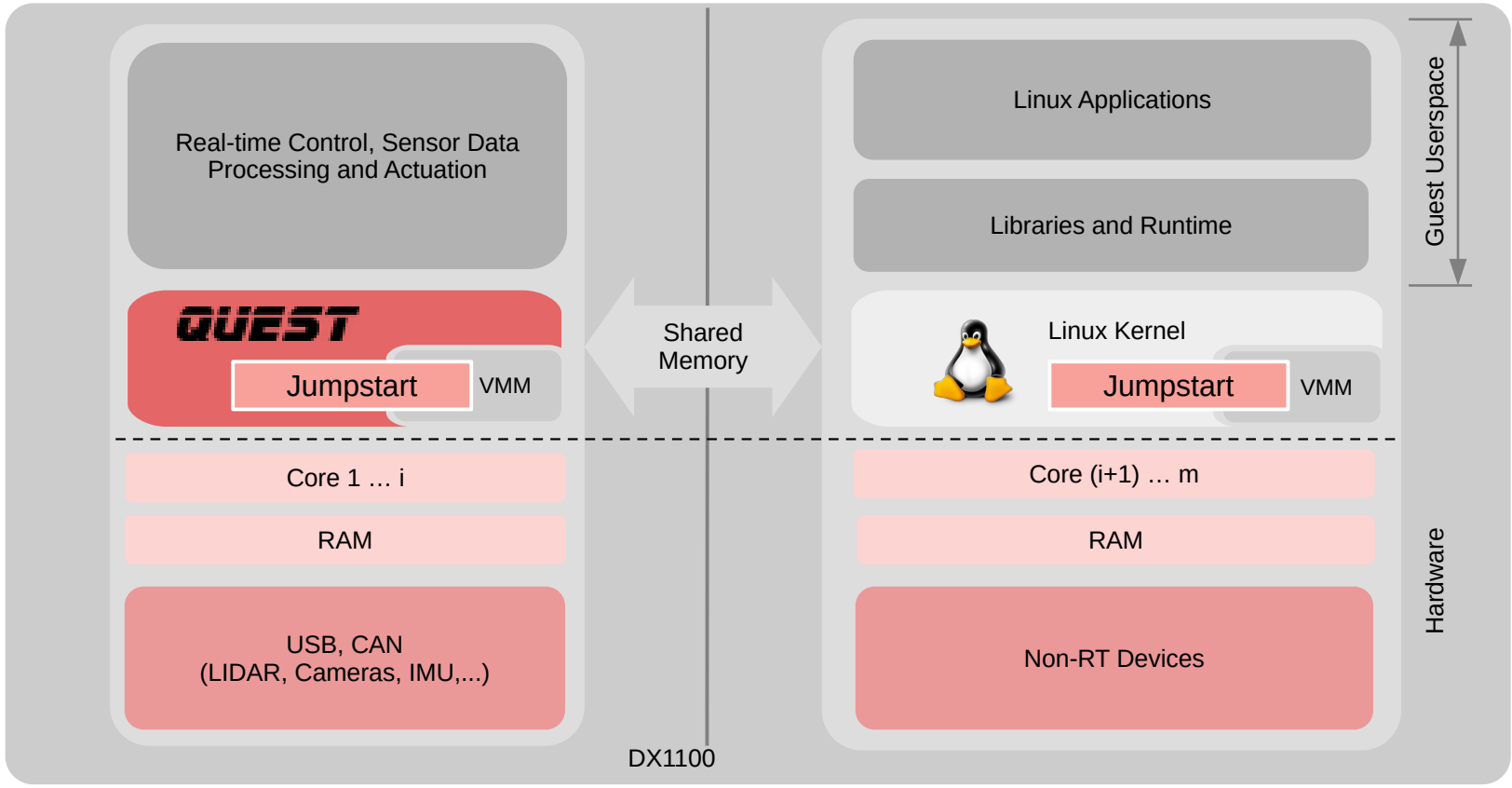
DriveOS: Example OpenPilot ADAS+IC+IVI (EMSOFT'21)



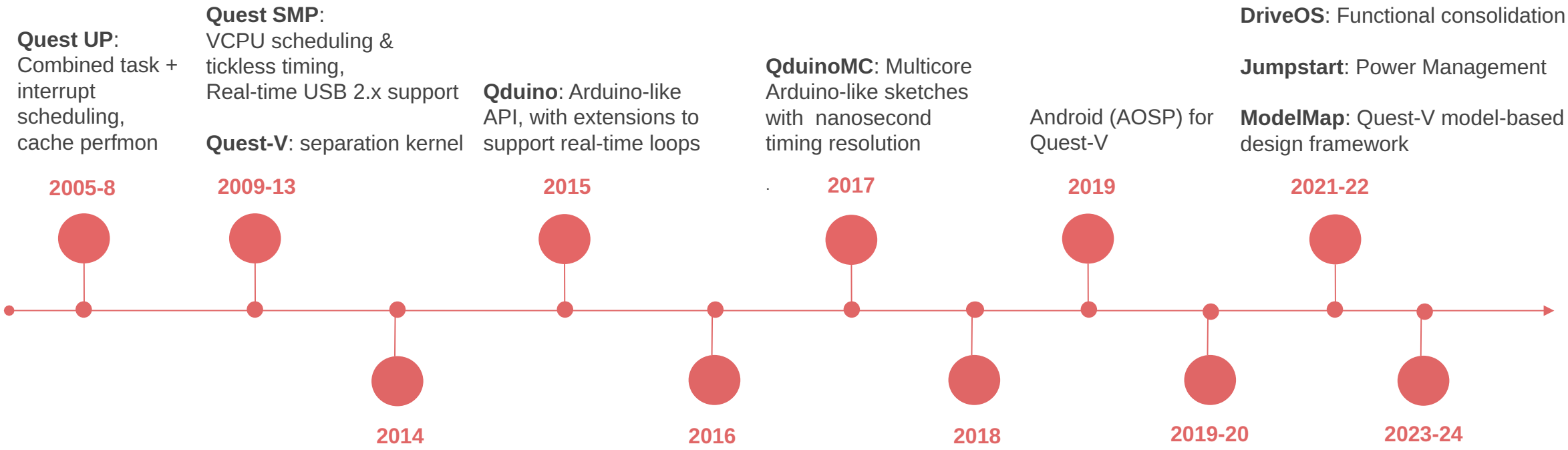
*Compare with Linux-only system with PREEMPT_RT & SCHED_DEADLINE tasks

Jumpstart Power Management (RTAS'22)

- PC hardware requires Firmware POST, bootloader, device & service initialization to boot OS
- DriveOS uses Jumpstart ACPI S3 suspend-to-RAM & resume-from-RAM for low latency restart of critical tasks (e.g., CAN gateway services)



Timeline (See History Markdown)



Quest-V Summary

- Separation kernel built from scratch
  - Distributed system on a chip
  - Uses hardware virtualization to partition resources into sandboxes
  - Secure communication channels b/w sandboxes

- Sandboxes can have different criticalities
  - Linux or Android front-end for less critical legacy services

- Sandboxes responsible for local resource management
  - Avoids monitor involvement

Getting Started with the SDK Source Tree

- The following Markdown topics are relevant to getting started
  - Source tree layout
  - Configuring Quest(-V)
  - Using the Quest Docker toolchain to build system images
    - Quest(-V) has its own Docker toolchain
    - Run use-docker to setup environment to build Quest or Quest-V, depending on settings in default-config.mk

Quest API

- VCPU - Virtual CPU scheduling (Markdown)
- Newlib - C library
- Tuned Pipes (Markdown)
- Shared Memory Communication – shmcomm (Markdown)
- USB
- See the Quest API Markdown for further information

SDK – Run Quest/Quest-V in QEMU

- See
  - Standalone Quest booted from a USB Markdown
  - Quest-V Markdown

Application Development in Quest

- The /boot directory in Quest is the default location for all userspace executables
- The directories containing user programs are specified in the QUEST_USER_PROGS_DIRS variable in the Makefile

Application Development in Quest

- Relevant directories:
  - sysprogs: as the name suggests, this directory holds system programs, such as the shell
  - tests: this directory is used for test programs, such as those created to evaluate new Quest features

Example Application Development in Quest

- hello_world – first code example
- fork_test – process creation
- vcpu_test – VCPU API example
- (a)sync_send/(a)sync_recv – shared memory communication example

- Additional examples (not covered in the tutorial)
  - arduino_lcd – Simple communication with Arduino
  - cdc_read – USB data transfer
  - can-gw – Teensy CAN bus data logger

Application Development in Quest: hello_world (Markdown)

- hello_world.c compiled to ELF binary using Quest docker toolchain
- To run:
  - enter name of executable in Quest shell, or
  - automatically start executable at bootup using user_init service

```c
#include <stdio.h>

int main (int argc, char* argv[]) {
 printf("Hello World!\n");
}
```

```
>hello_world
Hello World!
>
```

Other Examples: fork_test

```c
#include <stdio.h>

void main() {
 int pid;
 if ((pid = fork ())) {
   printf ("parent: my id is %d\n", getpid());
   waitpid (pid);
   _exit (0);
 } else {
   printf ("child: my id is %d\n", getpid());
   _exit (0);
 }
}
```

```
>fork_test
child: my id is 10
parent: my id is 9
```

Other Examples: vcpu_test

There are several args, but here we specify that we want a Main VCPU with a budget of 20 us allocated every period of 100 us

```
#include <stdio.h>
#include <vcpu.h>

int main(int argc, char* argv[]) {
 struct sched_param s_params = {.type = MAIN_VCPU, .C = 20, .T = 100};
 struct sched_param s_params2;
 memset (&s_params2, 0, sizeof(s_params2));
 int new_vcpu = vcpu_create (&s_params);
 if (new_vcpu < 0) {
   printf ("Failed to create vcpu\n");
   exit (1);
 }
```

Other Examples: vcpu_test

```
 vcpu_getparams (&s_params2);
 printf ("Before vcpu bind task: C = %d, T = %d\n", s_params2.C, s_params2.T);
 vcpu_bind_task (new_vcpu);
 usleep (1000000);
 vcpu_getparams (&s_params2);
 printf ("After vcpu bind task: C = %d, T = %d\n", s_params2.C, s_params2.T);
}
```

```
>vcpu_test
Before vcpu bind task: C = 5000, T = 50000
After vcpu bind task: C = 20, T = 100
>
```

Parameters of the default (Best-Effort) VCPU

Parameters of the newly created VCPU

shmcomm Example: async_send (Quest)

Arguments to open a channel to send are:

- shm key - unique ID
- Flags - new channel/connect to existing channel; async/synch
- Marshall function - callback function that is called before data is written to the channel
- Size of each packet
- Buffer Length - only valid in the case of synch communication
- Pointer to the channel that can be used for further operations

```
#include <stdlib.h>
#include <stdio.h>
#include <shmcomm_user.h>
#include <shmcomm_internal.h>

int main() {
 printf ("Sender\n");
 shmcomm_channel_t comm_channel;
 int res = shmcomm_open_send (153, (SHMCOMM_CREATE_CH | SHMCOMM_ASYNC_CH),
                              NULL, sizeof(int), -1, &comm_channel);
 if (res < 0) {
   printf("Error\n"); return 0;
 }

 int X = 10;
 printf ("Sending X\n");
 shmcomm_write (comm_channel, &X, sizeof(int));
 printf ("Sent X=%d\n", X);

 if (shmcomm_close (comm_channel) < 0) {
   printf ("Error in closing channel.\n"); return -1;
 }
 printf ("Channel successfully closed.\n");
 return 0;
}
```

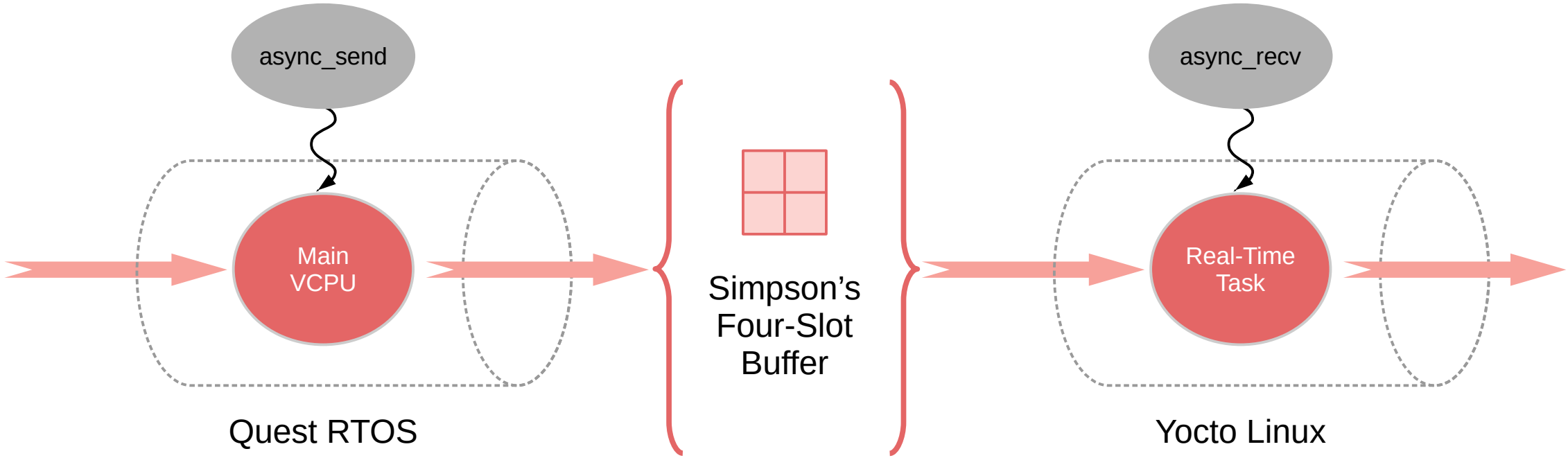
shmcomm Example: async_recv (Linux)

- The receiver has to be aware of the shm key to connect to the right channel
- Since we are connecting to an existing channel, type of channel is unnecessary; it can be inferred by the shmcomm functions
- The unmarshall function is called just after reading the channel, and just before returning the data to the application
- The other args are the same as the sender, except for one key difference: in Linux, the program uses a file descriptor instead of a shmcomm channel pointer
- get_vshm_key() allows a Linux program to obtain the shm key from a file descriptor

```c
#include <stdlib.h>
#include <stdio.h>
#include <shmcomm_user.h>

int main() {
 printf ("Receiver\n");

 int fd = shmcomm_open_receive (153, SHMCOMM_CONNECT_CH,
                                NULL, sizeof(int), -1);
 printf ("Value of res: %d\n", fd);
 if (fd < 0) {
   printf("Error\n"); return 0;
 }

 printf ("Recv Channel Key: %u\n", get_vshm_key(fd));

 int X;
 shmcomm_read (fd, &X, sizeof(int));
 printf ("Received X=%d\n", X);
 shmcomm_close (fd);

 return 0;
}
```

shmcomm Example: asynchronous communication

- In the given example, we use default scheduling parameters in both sandboxes
- If a custom Main VCPU were used in Quest, and the Linux task was configured to use SCHED_DEADLINE, we can tune their parameters to setup predictable asynchronous (or synchronous) communication across sandboxes in Quest-V



User Interaction: shell

- View tasks or VCPUs currently in the system: ps <-v|-t> [vcpuid | taskid]
- Terminate tasks:
  - specific task: kill -t <taskid>
  - latest task created by the shell: kill -c
- Execute tasks with the shell active in the background: nowait <filename> <args>
- Move the shell to the background: wait

User Interaction: shell

Ensures the shell remains active

```
>nowait thread
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #1!
Main thread!
Child thread 1!
Hello World! It's me, thread #2!
Child thread 2!
ps -t

4 tasks
 ID                        Name Parent VCPU PCPU          Status Thread
  8             /boot/user_init      8    0    0         running      1
  9                /boot/thread      9    0    0        sleeping      3
 10                             9    0    0        sleeping      1
 11                             9    0    0        sleeping      2
ps -v

1 VCPUs
VCPU PCPU Type          C          T  num_tasks
   0    0 main       5000      50000          4
```

List task and VCPU status with ps -t and ps -v

All tasks are assigned to the Best-Effort VCPU by default

User Interaction: shell

The parent has been marked for termination…

```
Child thread 1!
Child thread 2!
Main thread!
kill -c

kill: task 9 marked for termination
ps -v

1 VCPUs
VCPU PCPU Type          C          T  num_tasks
   0    0 main       5000      50000          4
ps -t

4 tasks
 ID                          Name Parent VCPU PCPU            Status Thread
  8                /boot/user_init      8    0    0           running      1
  9                   /boot/thread      9    0    0 sleeping/terminal      3
 10                                     9    0    0          sleeping      1
 11                                     9    0    0          sleeping      2
```

… and is hence terminal.
But since it is sleeping, it has not been terminated.

User Interaction: shell

```
Child thread 1!
Child thread 2!
Main thread!
ps -t

4 tasks
 ID                          Name Parent VCPU PCPU           Status Thread
  8               /boot/user_init      8    0    0          running      1
  9                  /boot/thread      9    0    0  waiting/terminal     3
 10                                    9    0    0 sleeping/terminal     1
 11                                    9    0    0 sleeping/terminal     2

Child thread 1!
Child thread 2!
ps -t

1 tasks
 ID                          Name Parent VCPU PCPU           Status Thread
  8               /boot/user_init      8    0    0          running      1
shell: task 9 has exited
>
```

When the parent thread wakes up, it marks the child threads for termination and then waits

Finally when the child tasks next wake up, they are terminated, also terminating the parent

User Interaction: quesh

- quesh /dev/qSB<sandbox-number>
- Sending a ctrl-c/kill command from Linux via shared memory to terminate foreground task in Quest
  - echo $'\cc' > /dev/qSB<sandbox-number>
  - echo "kill -c" > /dev/qSB<sandbox-number>

Debugging Quest and Quest-V

- Both Quest and Quest-V can be debugged with GDB attached to QEMU
- See Markdown

Drako Motors

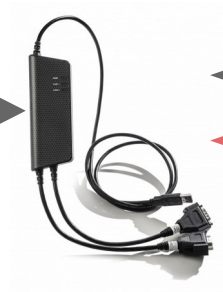
- Drako Motors: Silicon Valley company building luxury electric supercars with software-defined technology
- Tech Stack: Quest RTOS + Quest-V hypervisor + Yocto Linux for real-time, safe, and secure operation
- DriveOS: Drako's own software stack with custom components powering its vehicle platform

Drako Motors

- DX1100 (Quest RTOS + Linux Yocto) receives raw CAN data, parses it on Quest, and shares structured messages via shmcomm for real-time IVI/IC rendering
- Custom Yocto layer (meta-idc-minimal) provides core services, including a CAN parser and the Qt/C++/QML-based Instrument Cluster application
- Development setup uses laptop + Kvaser + Teensy + HDMI display; simulated CAN signals from canutils canplayer flow through to DX1100 for live visualization

DriveOS: Example Drako Instrument Cluster Functionality

canplayer
playback
of CAN logs



**Kvaser
USBcan
Light 2x HS**

Chassis / Body
CAN Channel

Powertrain
CAN Channel

**Teensy
4.1**

USB

USB-CAN-IO
Gateway

Quest RTOS

Shared Memory
Channels

**Instrument
Cluster Msgs**

Yocto Linux

QEMU SDK

DRAKO

BOSTON
UNIVERSITY

Questions?

- If you're interested in joining the Quest group, we're always keen to work with driver writers, library and application developers, and anyone with knowledge of computer architecture and operating systems

- Consider being a developer and join our Quest for knowledge!
- See https://www.questos.org for further information

Quest, still searching...