Interrupt response times on Arduino and Raspberry Pi

Tomaž Šolc
tomaz.solc@ijs.si

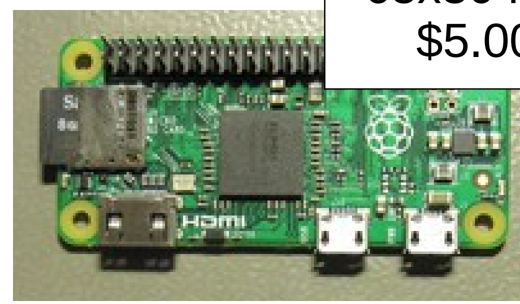
MEDNARODNA
PODIPLOMSKA ŠOLA
JOŽEFA STEFANA

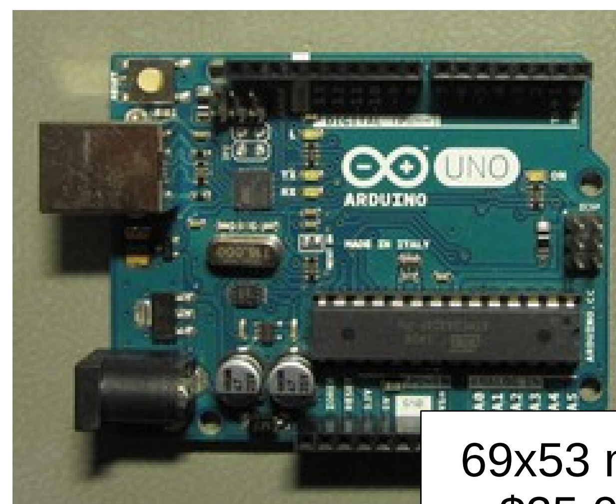
JOŽEF STEFAN
INTERNATIONAL
POSTGRADUATE SCHOOL

Introduction

- Full-featured Linux-based systems are replacing microcontrollers in some embedded applications
  - for low volumes, difference in BOM price is insignificant
  - very little difference in physical size
  - cheaper software development (proprietaty toolchains, asm vs. gcc, Python, Javascript, ...)
  - simpler debugging (JTAG vs. shell access, gdb)

65x30 mm
$5.00





69x53 mm
$25.00

A significantly different approach to real-time tasks

- general-purpose OS
  - pre-emptive kernel, priority scheduling
- application-profile CPU core
  - SMP, cache, pipelining, parallelism, MMU...
- 1 GHz clock, 1 GB RAM, 10 GB storage

gross overprovisioning

- No OS or simple RT-OS
  - hw. interrupt priority, low system overhead
- microcontroller-profile CPU core
  - well-defined instruction lengths, access times...
- 10 MHz clock, 10 kB RAM, 100 kB storage

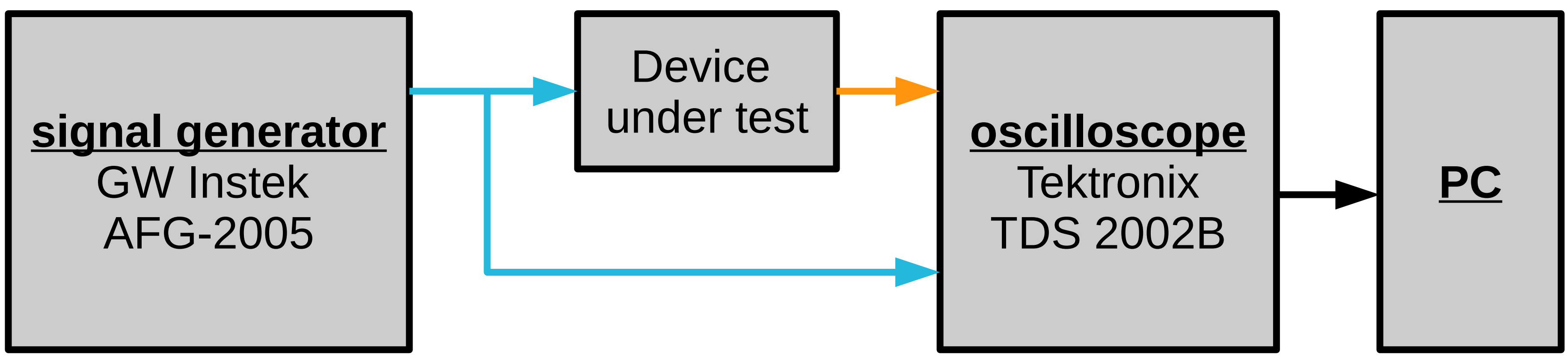
predictability

Motivation for experiment

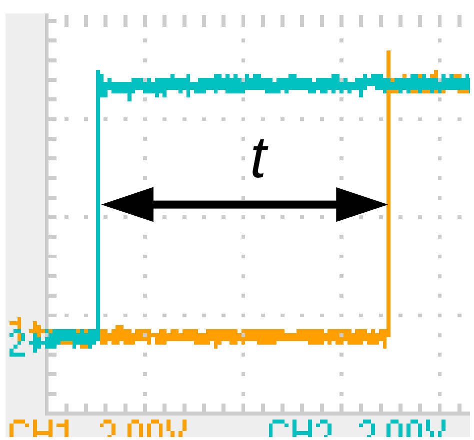
- What kind of interrupt response times can be expected from these systems out-of-the-box?
- Test most straightforward implementations
  - Examples from manuals, first results on web search, ...
  - First implementation typically also the last (cheaper to go with higher-performance hardware than study and optimize prototype, lack of time, expertise, ...)
- Two platforms commonly used as starting points
  - e.g. hardware startups making IoT devices, SMEs (not specialized industries with large existing teams)

Experiment setup

Task: on PIN rising edge raise POUT



signal generator outputs a square wave signal with period >> *t*

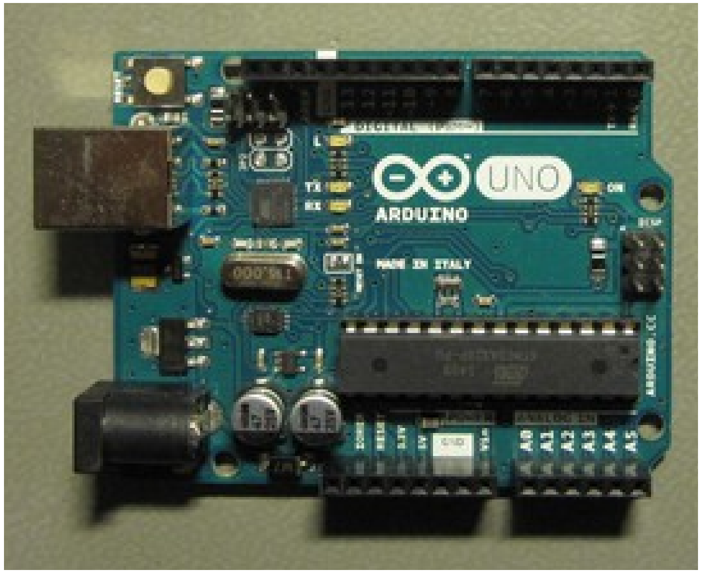


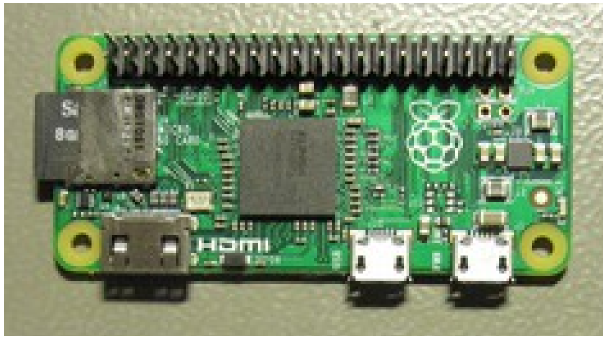
oscilloscope measures time between rising edges on PIN and POUT lines

PC stores results for later analysis

Hardware

- Arduino Uno
  - Atmel ATmega328p,
    AVR architecture, 16 MHz clock
  - Arduino IDE 1.0.5 (C++)
- Raspberry Pi Zero
  - Broadcom BCM2835,
    ARM11 architecture, 1 GHz clock
  - Raspbian Jessie OS
    (2016-03-18 image, Linux 4.1.19+)
  - Python 2.7.9, RPi.GPIO 0.6.2

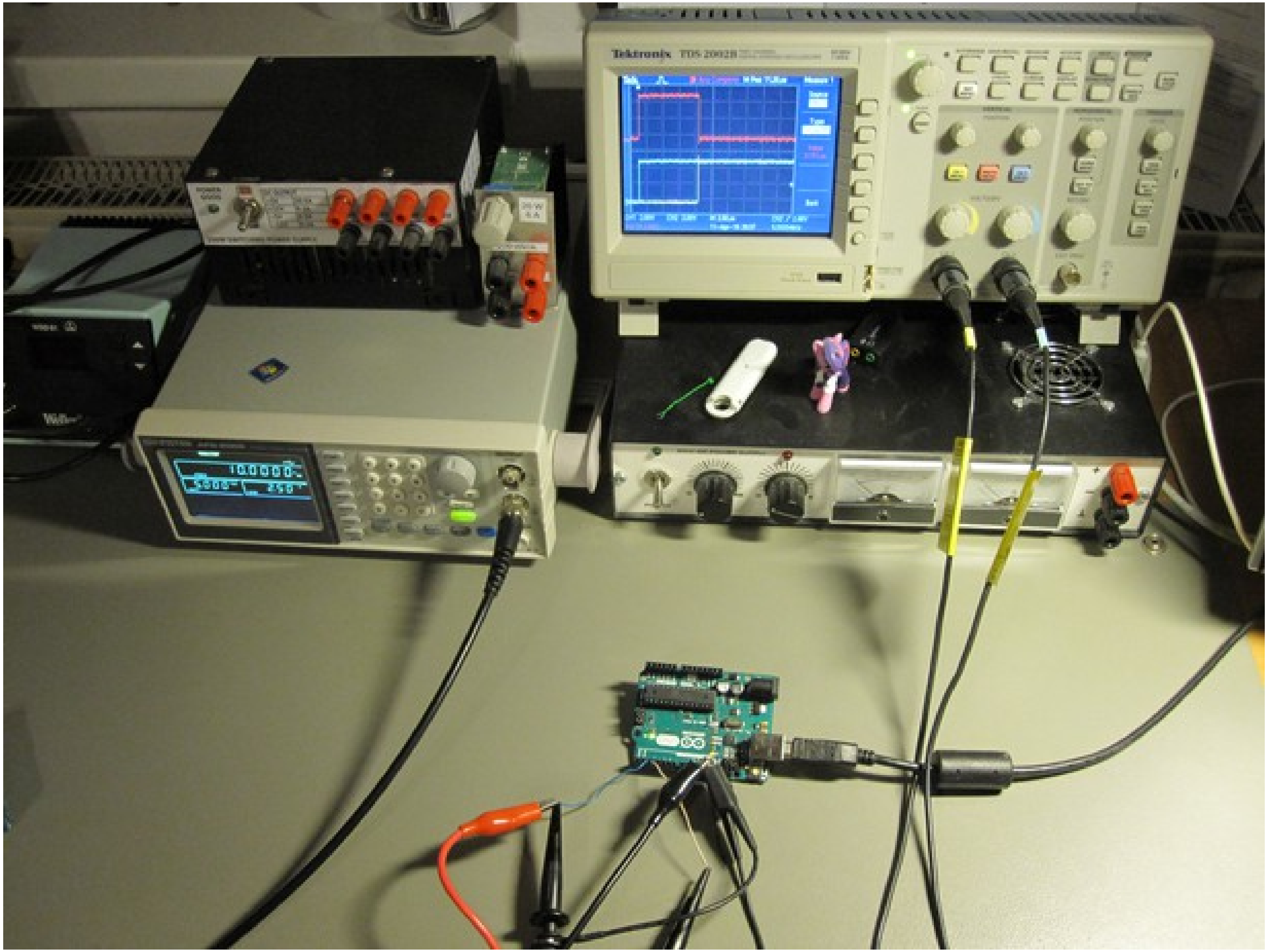




Software implementations

- Arduino Uno
  - interrupt service routine on GPIO interrupt using *attachInterrupt()* standard library call.
  - busy polling loop using *digitalRead()* std. library call.
- Raspberry Pi Zero
  - **native, kernel space** - Linux .ko module using »GPIO consumer« interface.
  - **native, user space** - Compiled C program using *sysfs* interface for GPIO.
  - **interpreted Python script** using RPi.GPIO library.

Experiment setup



Results

- Arduino, IRQ

  t_{min} = 8.9476 µs

  t_{avg} = 9.0673 µs

  t_{max} = 14.0163 µs

- Arduino, polling

  t_{min} = 6.6675 µs

  t_{avg} = 8.6581 µs

  t_{max} = 14.8937 µs

- R-Pi, native, kernel space

  t_{min} = 6.0367 µs

  t_{avg} = 12.6761 µs

  t_{max} = 43.7788 µs

- R-Pi, native, user space

  t_{min} = 179.9882 µs

  t_{avg} = 280.5045 µs

  t_{max} = 511.2023 µs

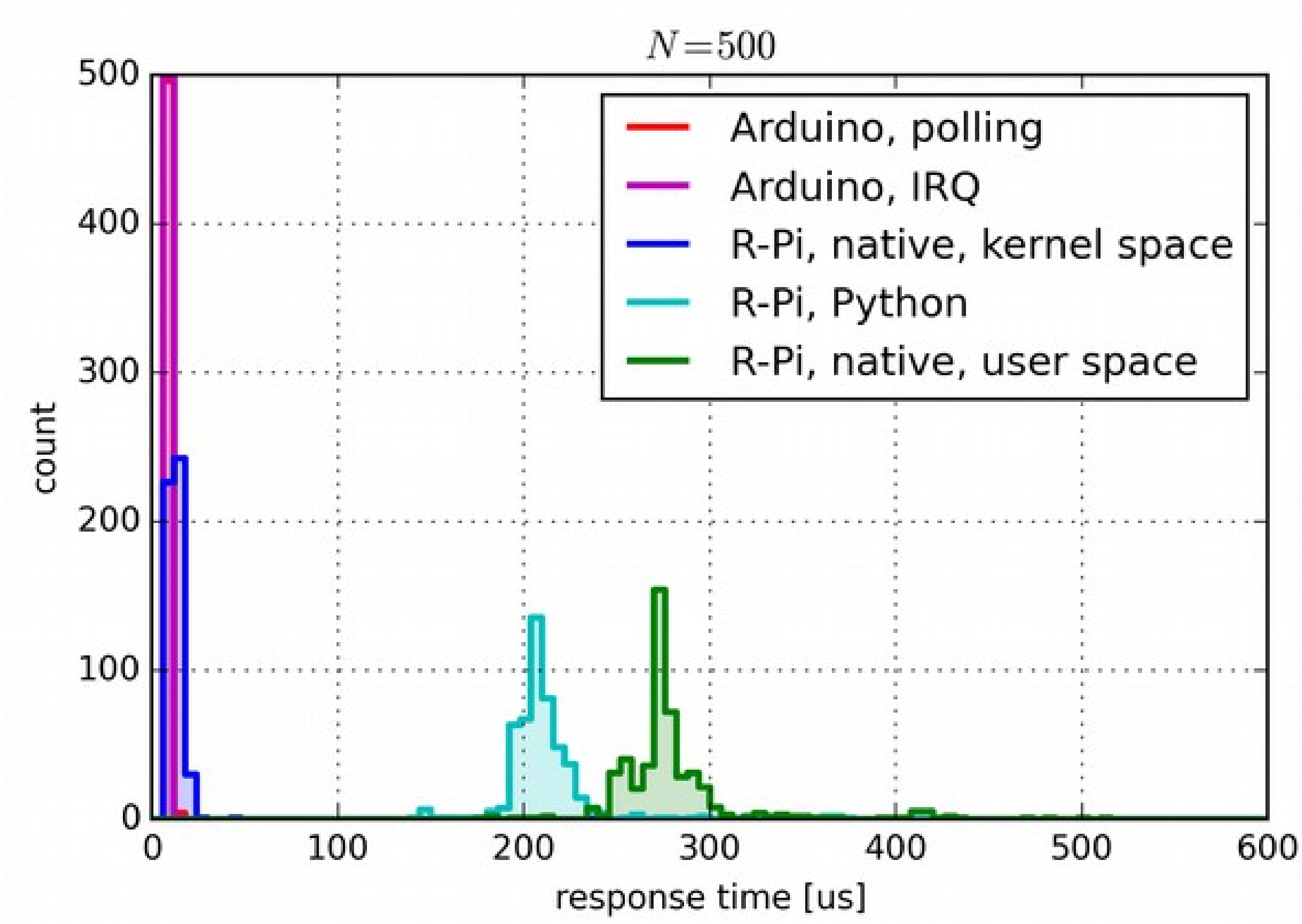
- R-Pi, Python

  t_{min} = 143.1988 µs

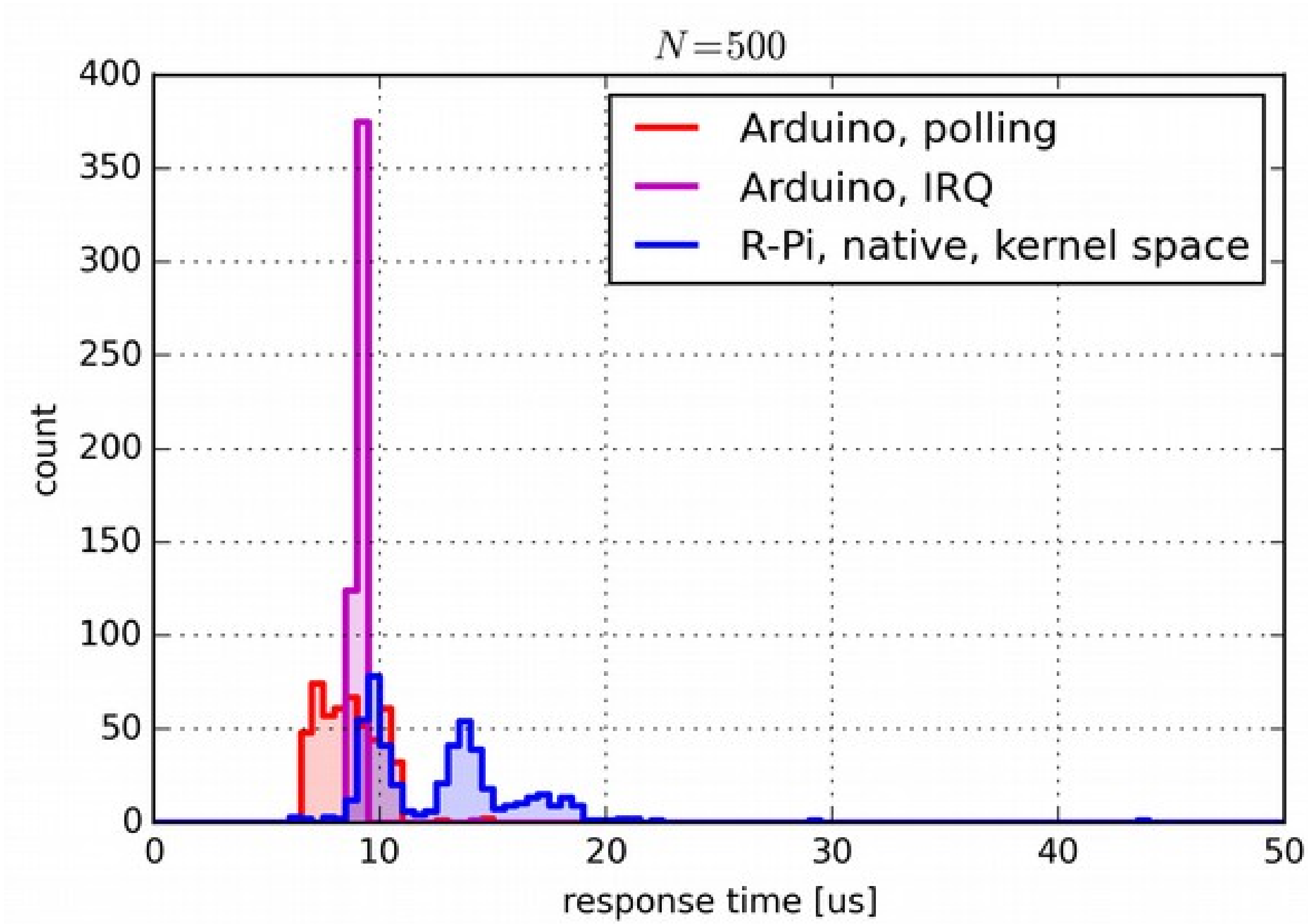
  t_{avg} = 212.9129 µs

  t_{max} = 377.4056 µs

Results (histogram 0-600 µs)



Results (histogram 0-50 μs)



Discussion of results

- Arduino response times unexpectedly inconsistent
  - Measured >5 μs spread,
    should be 0,25 μs given CPU clock, instruction lengths
- R-Pi kernel mode code on average close to Arduino
  - R-Pi has >60x faster CPU clock
  - Less consistent (kernel has many other ISRs to serve)
- Interpreted Python implementation faster than native userspace code?
- Tested on an otherwise idle system on R-Pi.

Cause of Arduino inconsistency

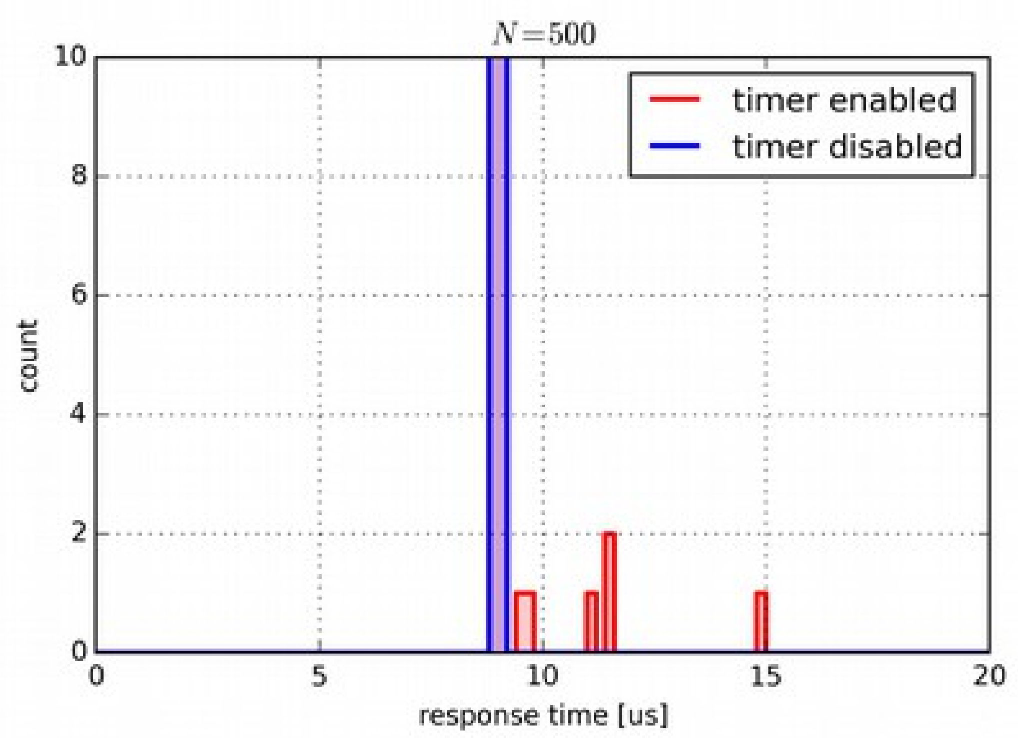
- Standard library keeps timer0 enabled.
  - Timer overflow interrupt competes with GPIO interrupt.
  - If timer0 is disabled, measurements fit theory.

t_{min} = 8.9485 μs

t_{avg} = 9.0526 μs

t_{max} = 9.1986 μs

Δt = 0.2501 μs

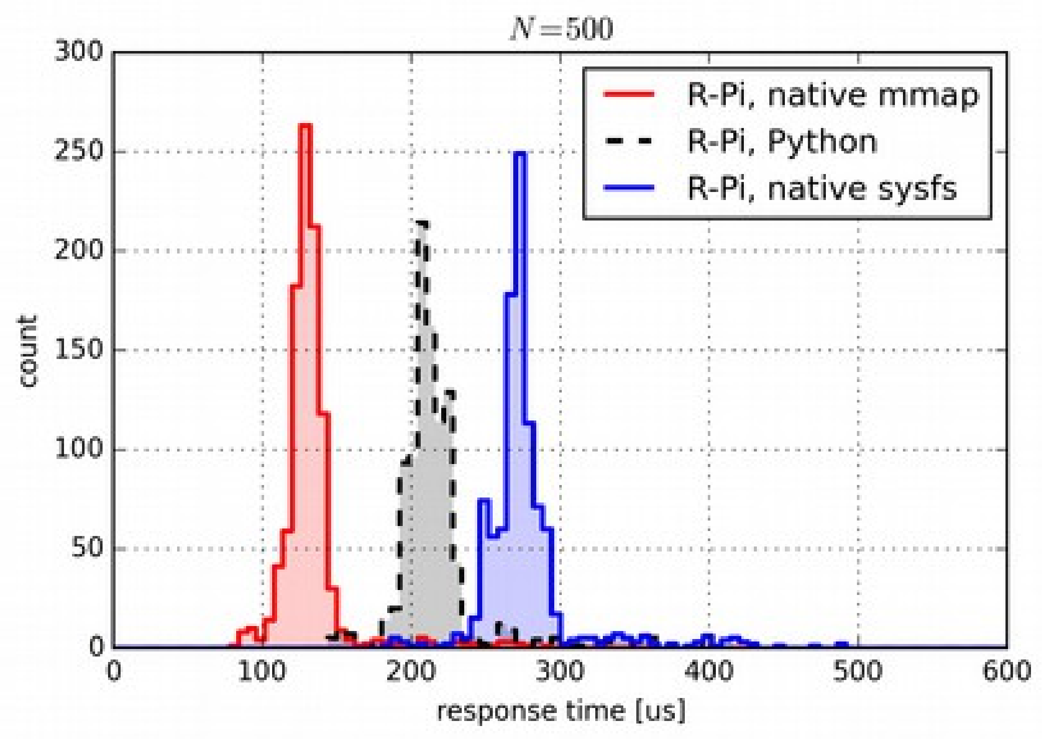


Why is Python faster than C?

- Python RPi.GPIO library mmaps GPIO registers to its own process address space
  - removes need for syscalls when changing POUT state.
  - syscall still needed when waiting for PIN edge – interrupt vectors not directly accessible from userspace.
  - syscalls are slow (context switch into kernelspace)
- Initial native userspace implementation used *sysfs*
  - GPIO lines exposed as special files in */sys* filesystem.
  - 3 POSIX syscalls per POUT state change:
    open(), write(), close()

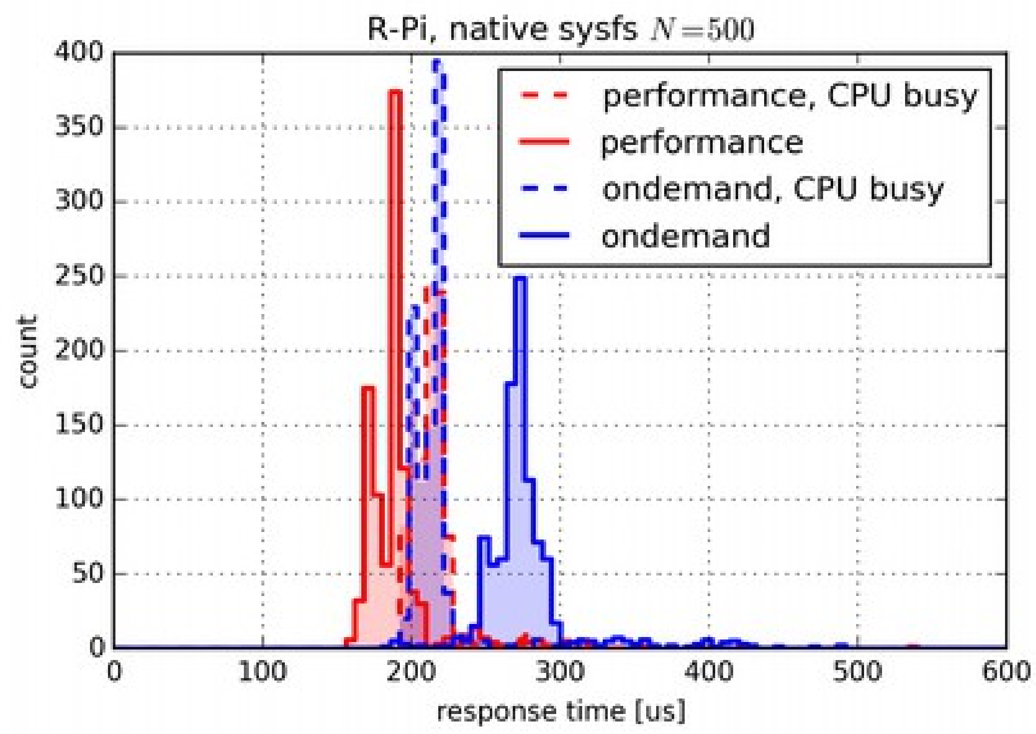
Why is Python faster than C?

- A native userspace implementation using mmap approach is faster than Python.



Effect of CPU load on R-Pi

- With default setup, R-Pi has *lower response times* when CPU is loaded compared to idle!
- Power saving feature – *ondemand* governor lowers CPU clock from 1 GHz to 700 MHz when CPU is idle.
- Changing governor to *performance* produces expected results



Conclusions

- Complex systems can be counterintuitive
  - simpler is not always faster
  - profiling and measurements are important
- R-Pi can be »good enough« for some real-time tasks
  - average times comparable to microcontrollers when using kernel driver – but upper bound is not predictable
  - expectations for reliability of consumer devices are decreasing - often features are more important.

    »it's good enough if it works 90% of the time«

Questions?

Tomaž Šolc
tomaz.solc@ijs.si

source code and raw data at
https://github.com/avian2/interrupt-response-times