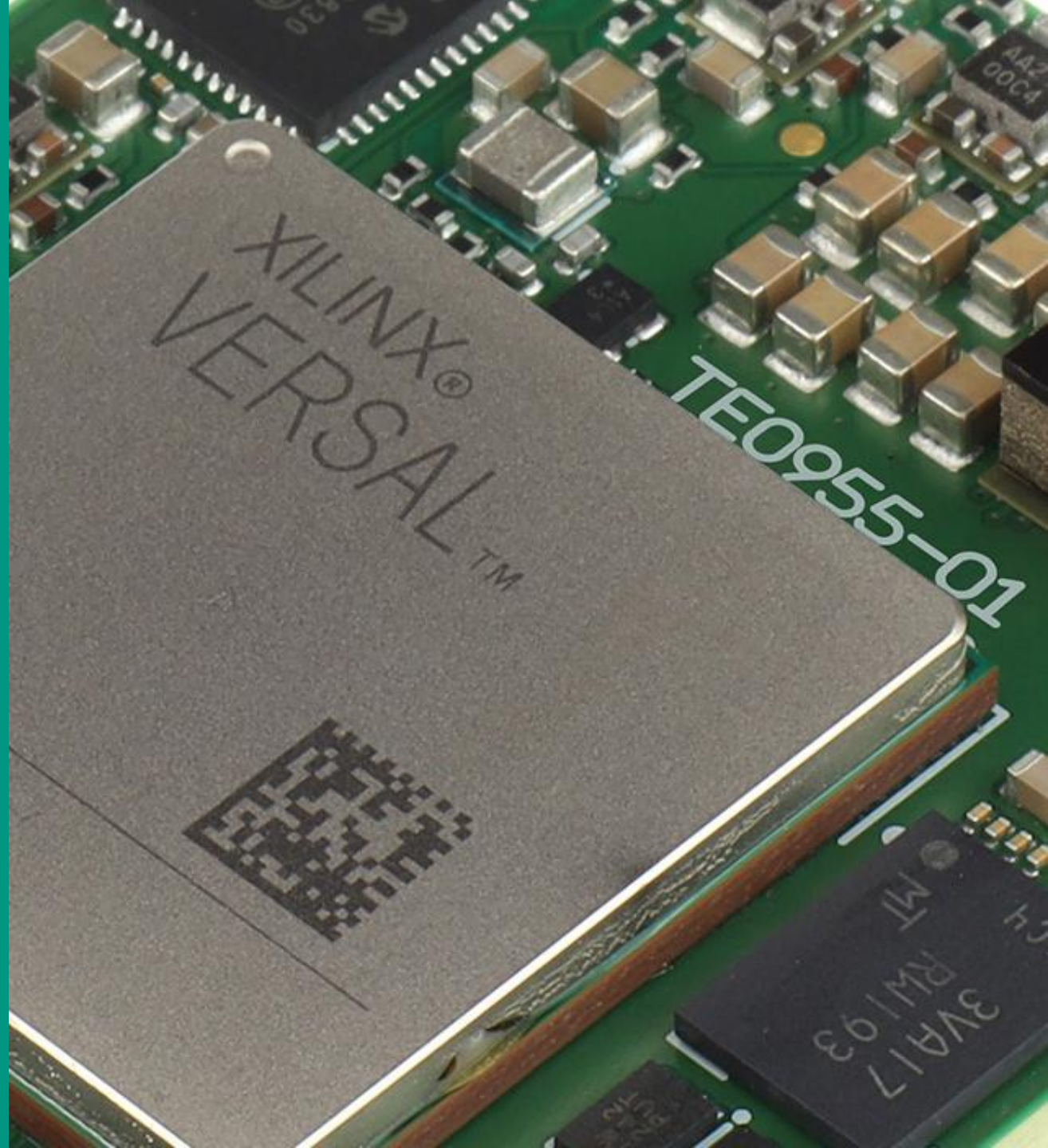
SoCks

A Build Framework for Heterogeneous High Performance System-on-Chips

Marvin Fuchs, Lukas Scheller,
Timo Muscheid, Oliver Sander,
& Luis Ardila-Perez

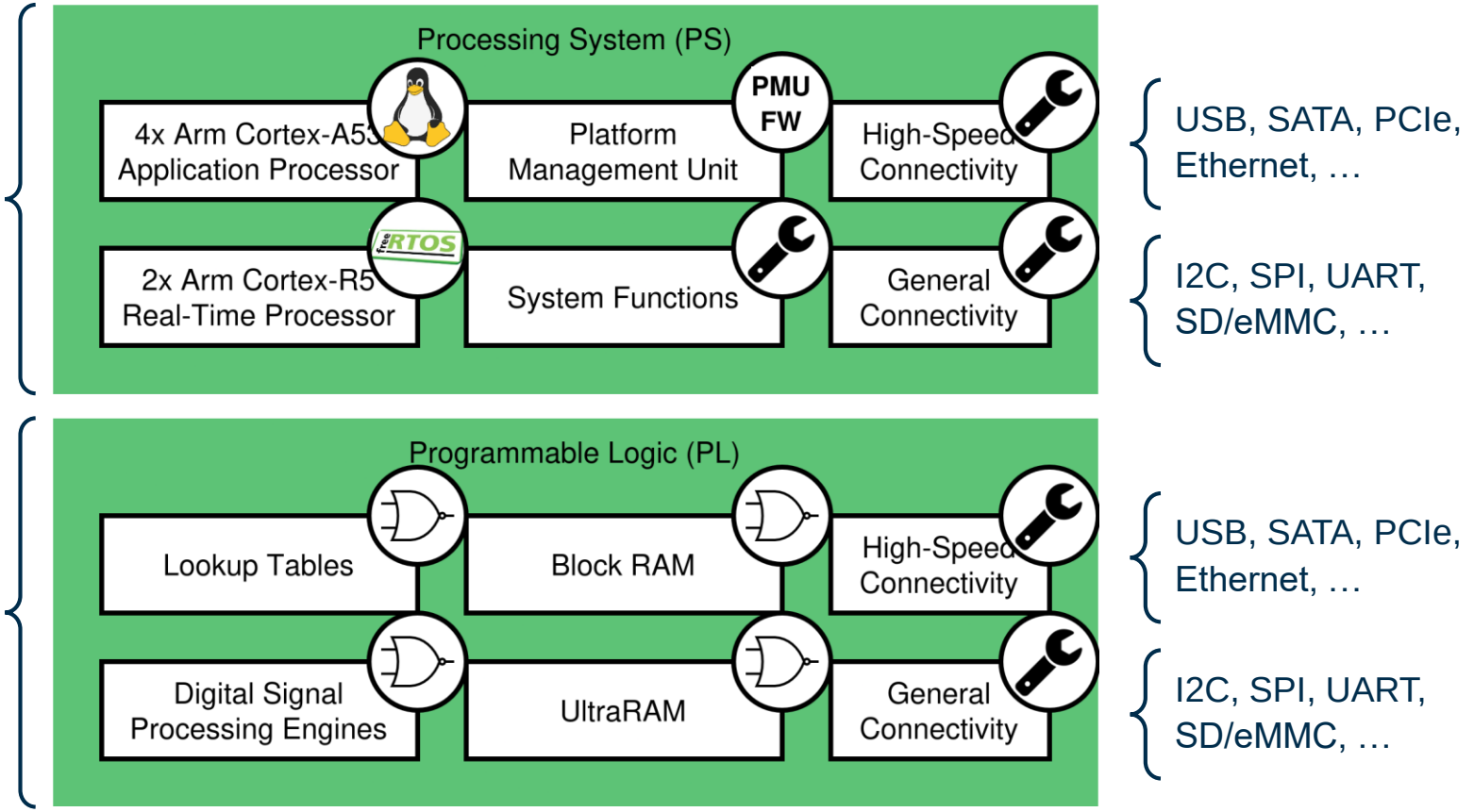
4th CERN SoC Workshop



Heterogeneous High Performance System-on-Chips

Processors
e.g. to provide
a user interface

FPGA Fabric
e.g. for parallel real-
time data processing



AMD Zynq UltraScale+ MPSoC

How to build an image for an AMD ZynqMP device?

Vivado:

- FPGA firmware development
- Low-level SoC configuration

Vitis:

- High-level development environment
- Bare metal software, HLS, AI, …

PetaLinux:

- Entry level Linux OS development
- Based on Yocto

Yocto:

- Advanced Linux OS development
- Recommended for production

AMD Vivado

AMD Vitis

AMD PetaLinux Tools

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842250/PetaLinux

yocto PROJECT

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841883/Yocto

How to build an image for an AMD ZynqMP device?

Vivado:

- FPGA firmware development
- Low-level SoC configuration

Vitis:

- High-level development environment
- Bare metal software, HLS, AI, …

PetaLinux:

- En… ols
- Based on Yocto

PetaLinux is being discontinued, and AMD will instead focus on supporting the Yocto workflow!
(https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2907766785/Yocto+Project+Machine+Configuration+Support#PetaLinux-Relationship-to-Yocto-Project)

Yocto:

- Advanced Linux OS development
- Recommended for production

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841883/Yocto

How to build images for such SoCs in general?

Two widely used frameworks with similar advantages and disadvantages

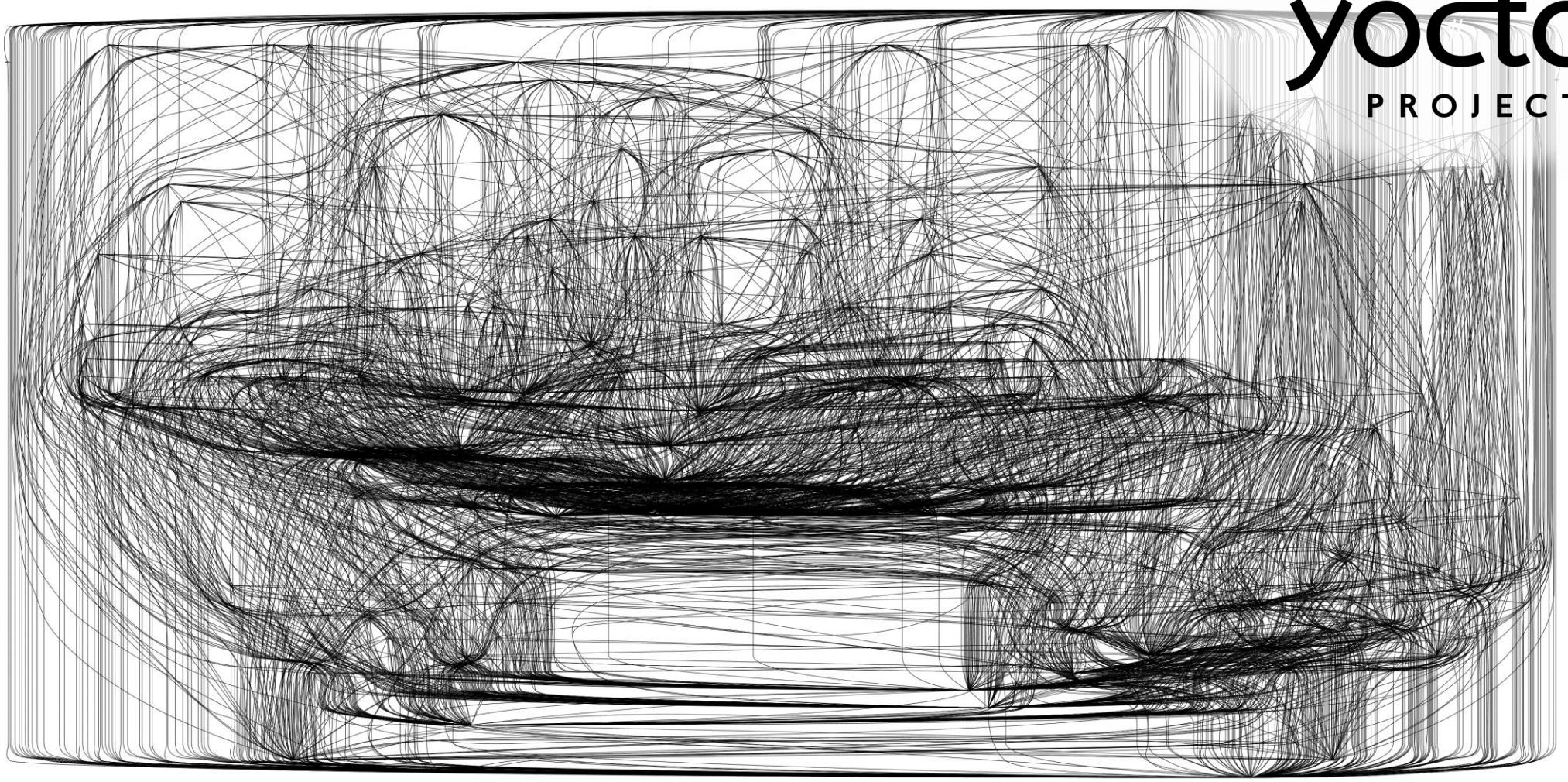
Advantages:

- Well established frameworks
- Supported by manufacturer

Disadvantages:

- Frameworks for building Linux distribution
- Flat hierarchy of hundreds of components
- Training is time consuming
- Hard to manage over time

The Downside of the Yocto Approach

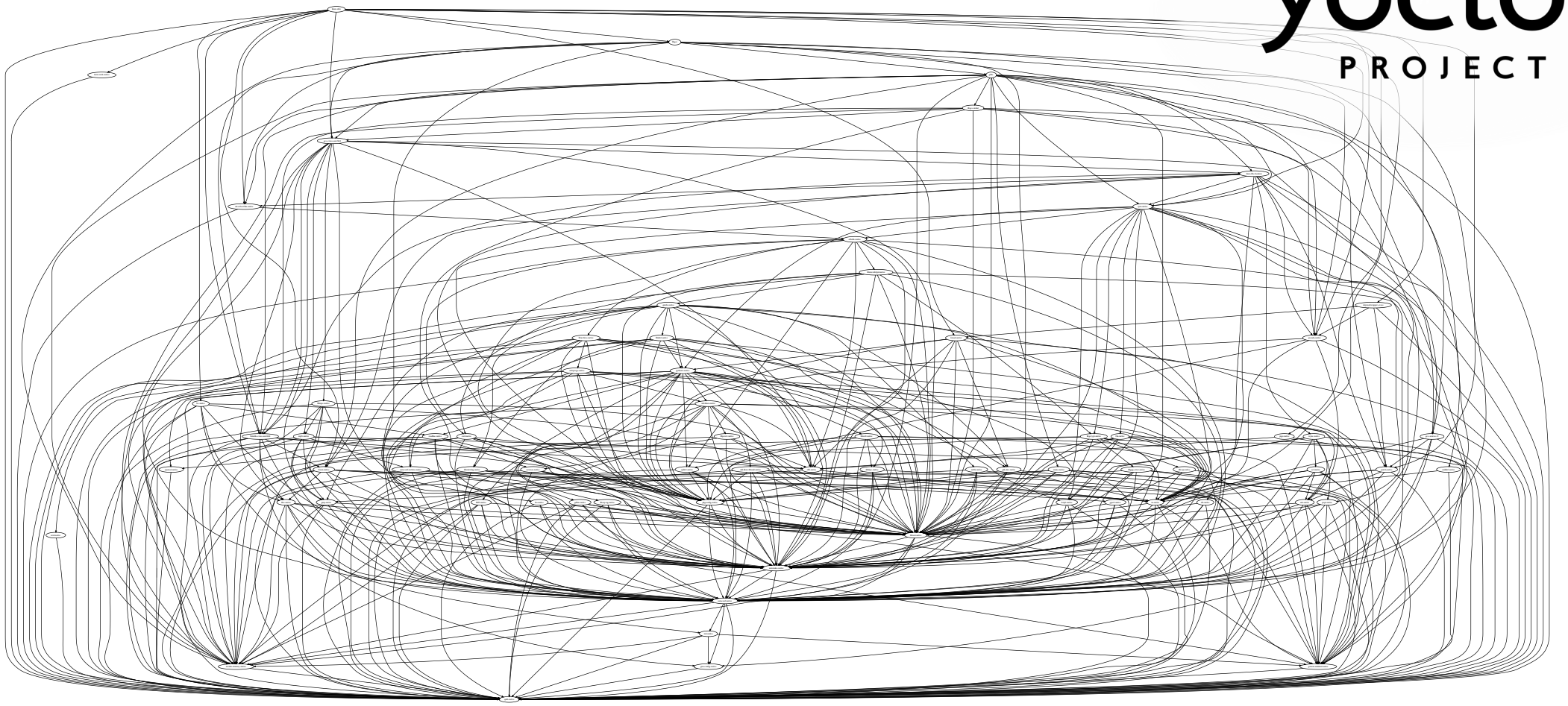


Dependencies of a complete AMD Zynq US+ image in Yocto

The Downside of the Yocto Approach

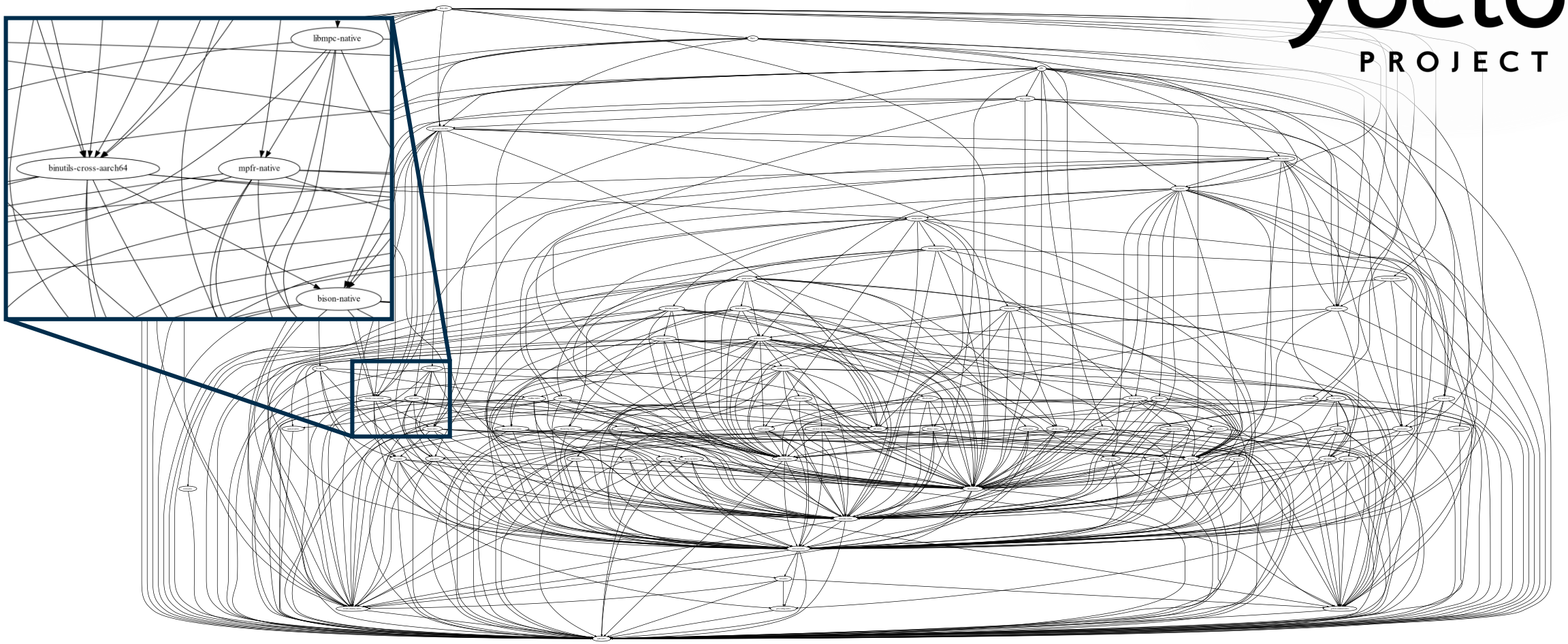
Dependencies of a complete AMD Zynq US+ image in Yocto

The Downside of the Yocto Approach



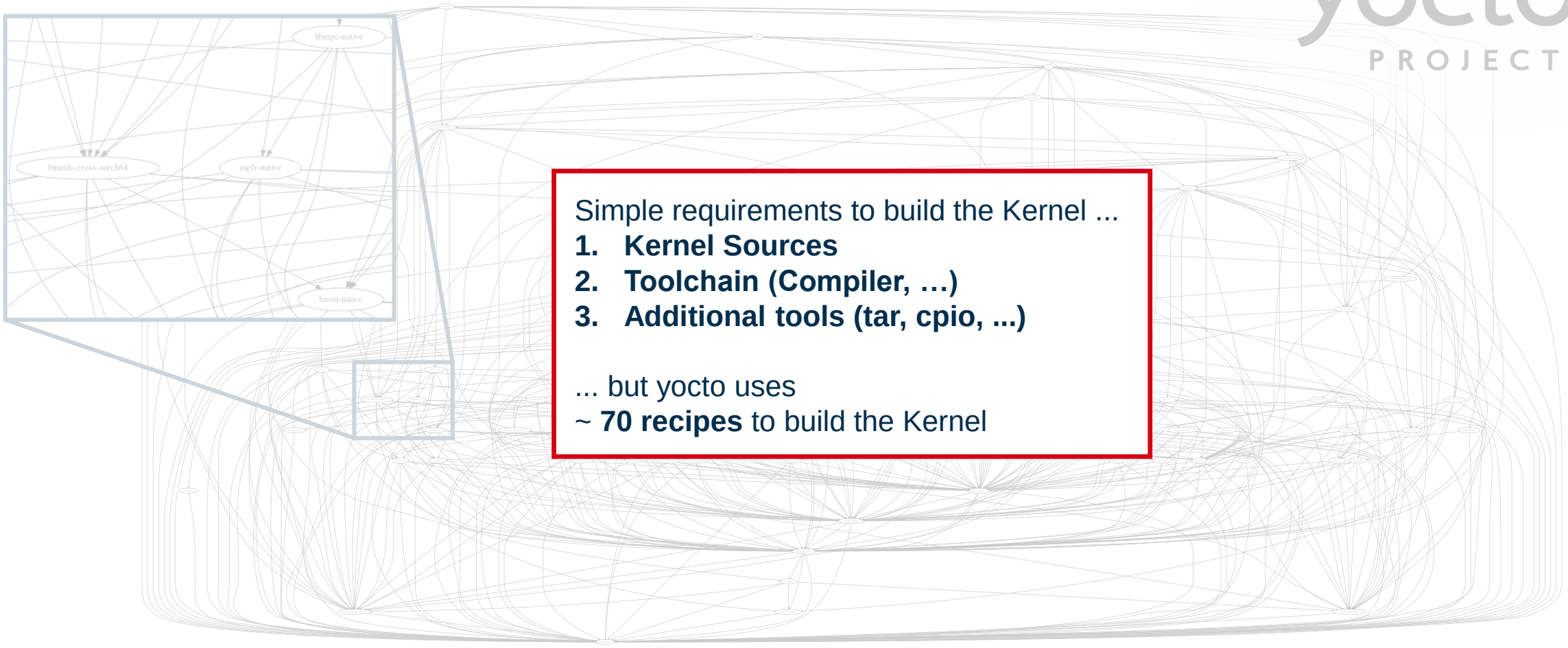
Dependencies of the Linux Kernel recipe in yocto

The Downside of the Yocto Approach



Dependencies of the Linux Kernel recipe in yocto

The Downside of the Yocto Approach



Simple requirements to build the Kernel ...

1. **Kernel Sources**
2. **Toolchain (Compiler, …)**
3. **Additional tools (tar, cpio, ...)**

... but yocto uses
~ **70 recipes** to build the Kernel

Dependencies of the Linux Kernel recipe in yocto

The Downside of the Yocto Approach

Simple requirements to build the Kernel ...

1. **Kernel Sources**
2. **Toolchain (Compiler, …)**
3. **Additional tools (tar, cpio, ...)**

... but yocto uses
~ **70 recipes** to build the Kernel

How could a simpler approach look like?

Dependencies of the Linux Kernel recipe in yocto

SoCks (SoC Blocks)

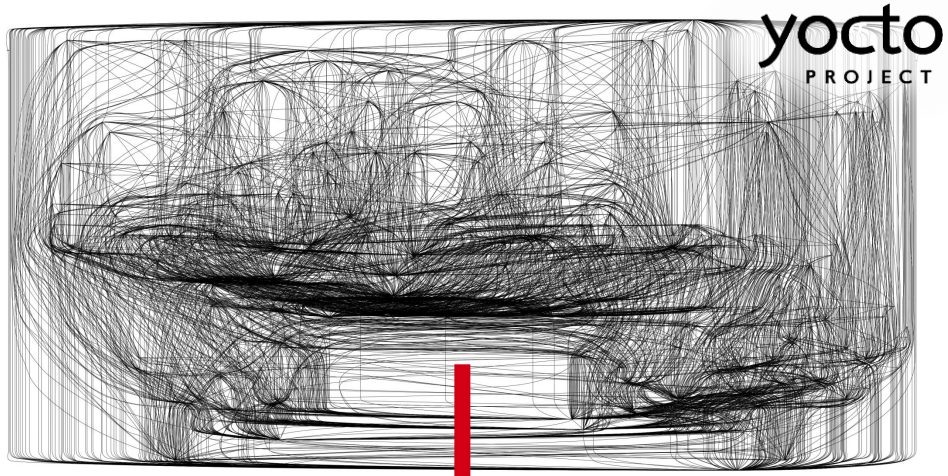
Objectives:

- Reduced complexity
- Easy to learn and use
- Fast and reliable
- Simple debugging of projects
- Full compatibility with GitLab CI/CD
- Encourage reuse of software and firmware

Ideas:

- Divide and conquer
- Introduce a new abstraction layer (blocks)
- Use existing Linux distributions

SoCks (SoC Blocks)

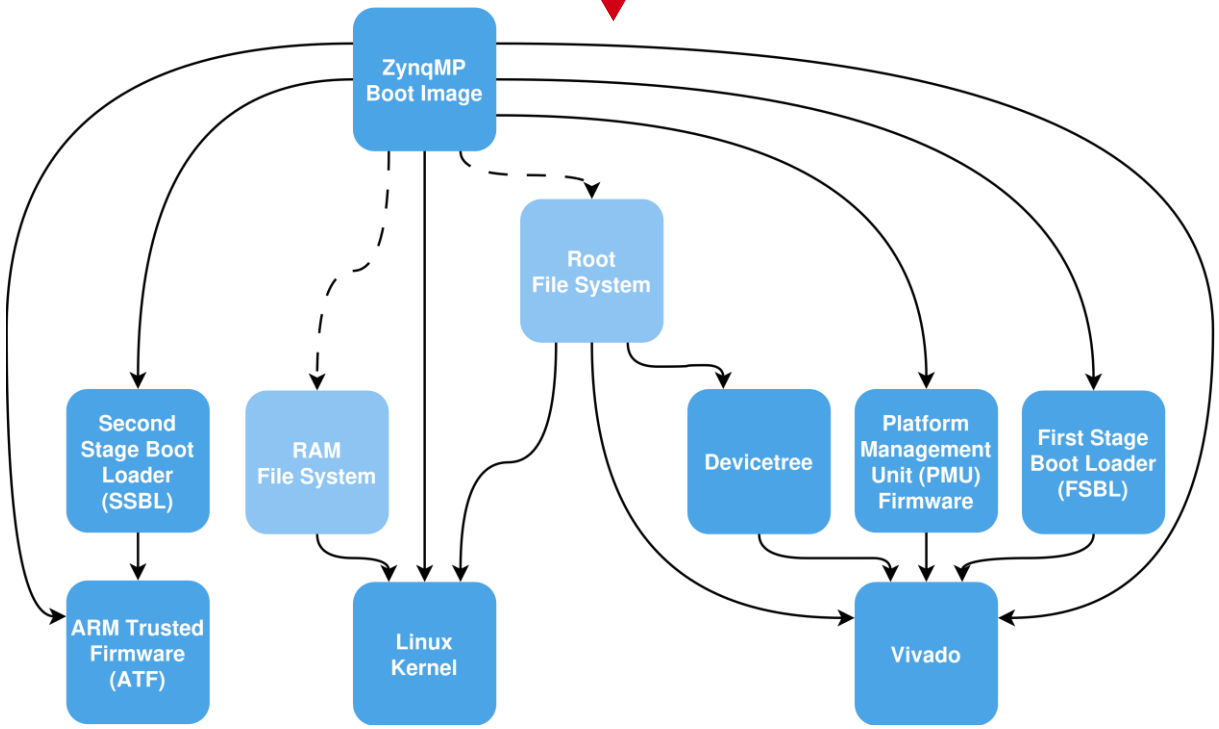


Objectives:

- Reduced complexity
- Easy to learn and use
- Fast and reliable
- Simple debugging of projects
- Full compatibility with GitLab CI/CD
- Encourage reuse of software and firmware

Ideas:

- Divide and conquer
- Introduce a new abstraction layer (blocks)
- Use existing Linux distributions



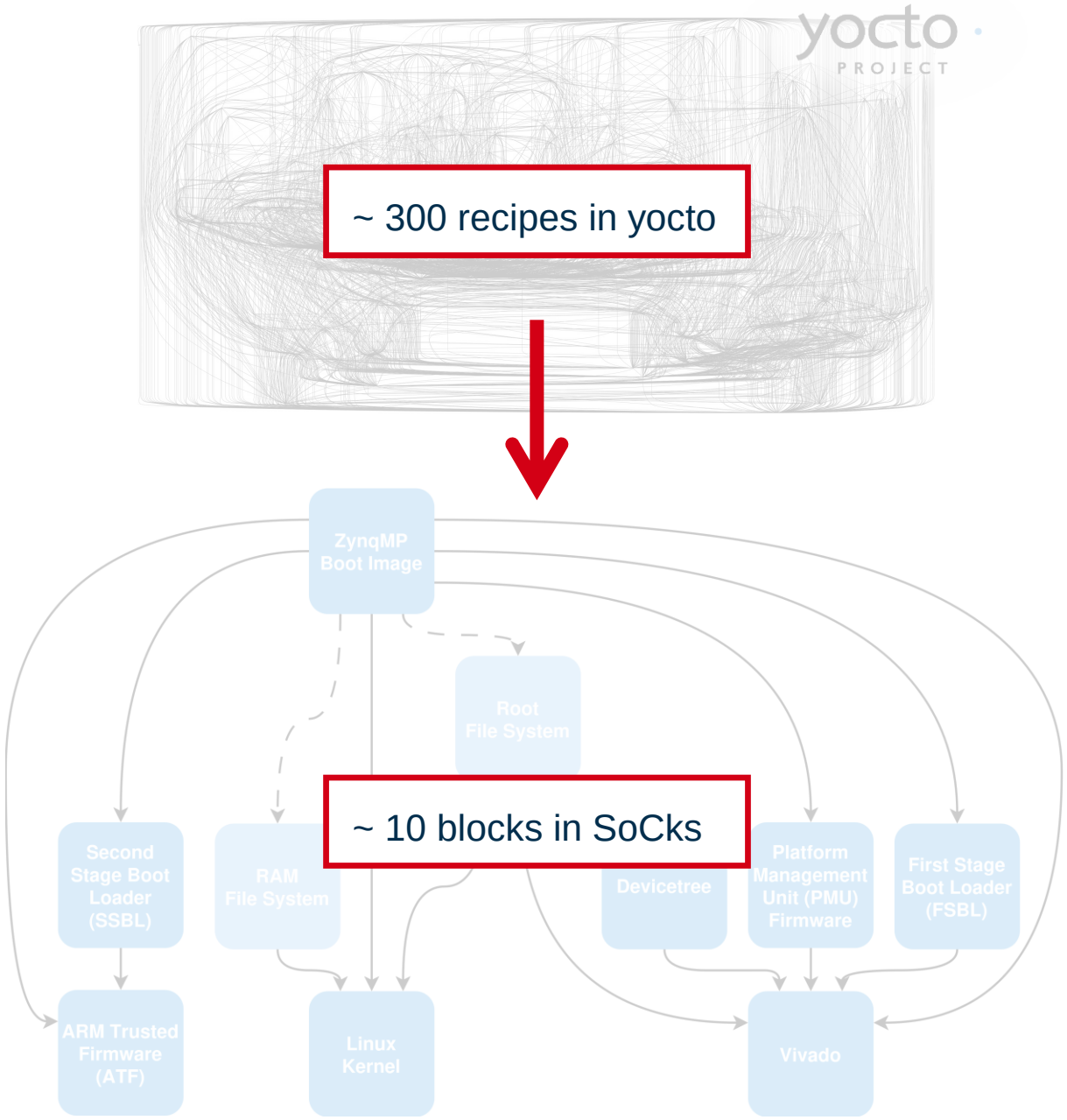
SoCks (SoC Blocks)

Objectives:

- Reduced complexity
- Easy to learn and use
- Fast and reliable
- Simple debugging of projects
- Full compatibility with GitLab CI/CD
- Encourage reuse of software and firmware

Ideas:

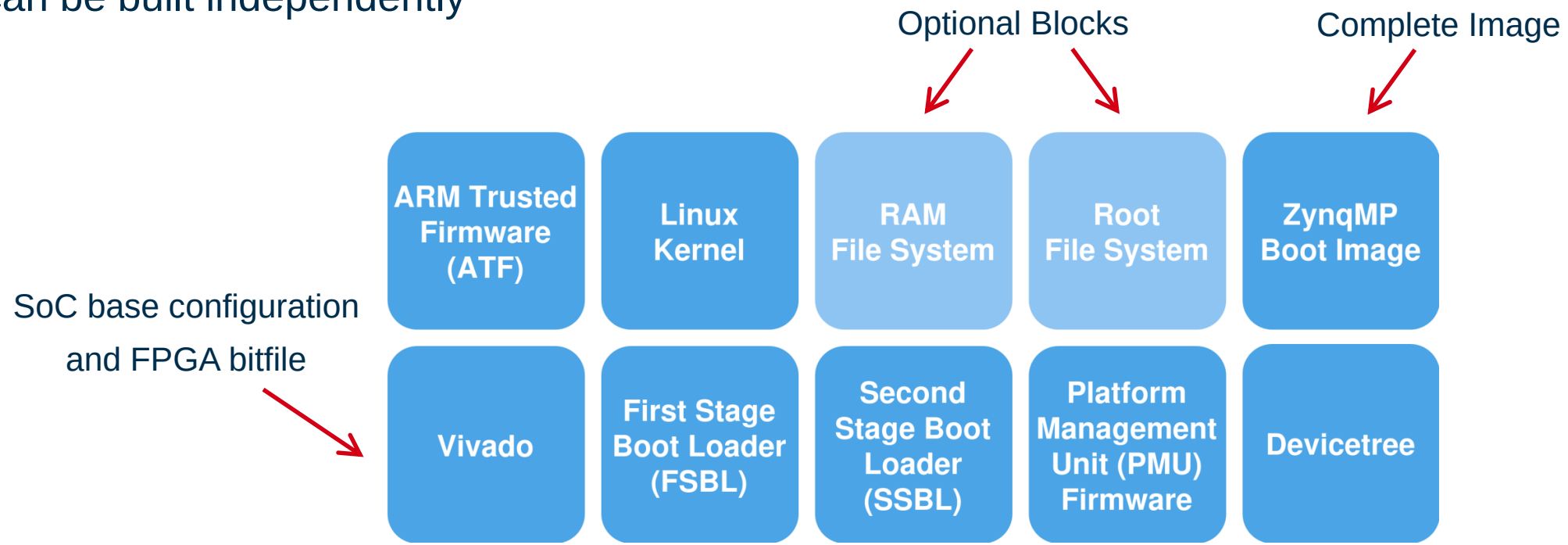
- Divide and conquer
- Introduce a new abstraction layer (blocks)
- Use existing Linux distributions



SoCks Software and Firmware Image

Image:

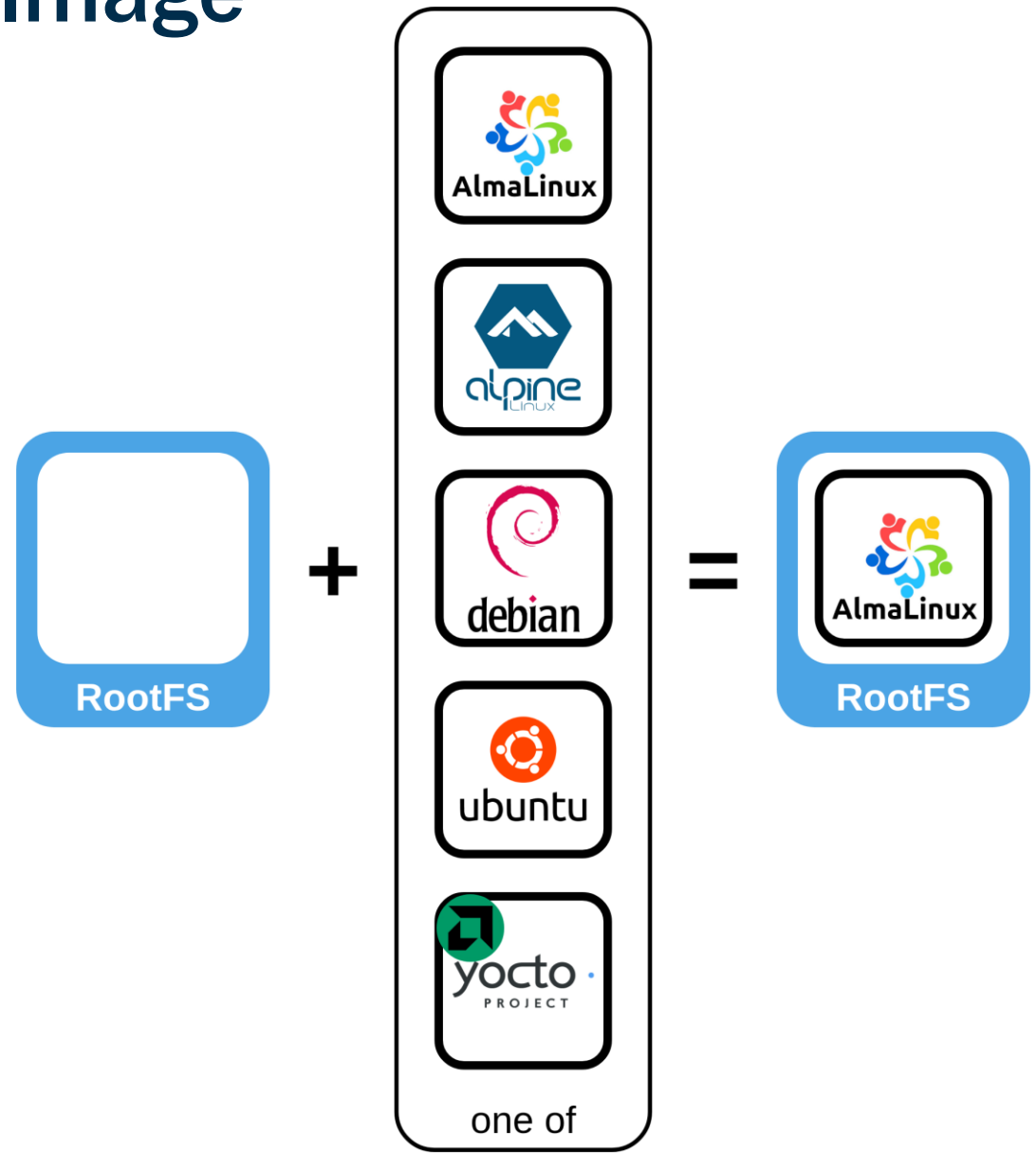
- Composed of reusable blocks
- Clear interfaces between blocks
- Blocks can be built independently



SoCks Software and Firmware Image

Builder:

- Builds one version of the block
- Multiple builders per block available
- Select one builder per block



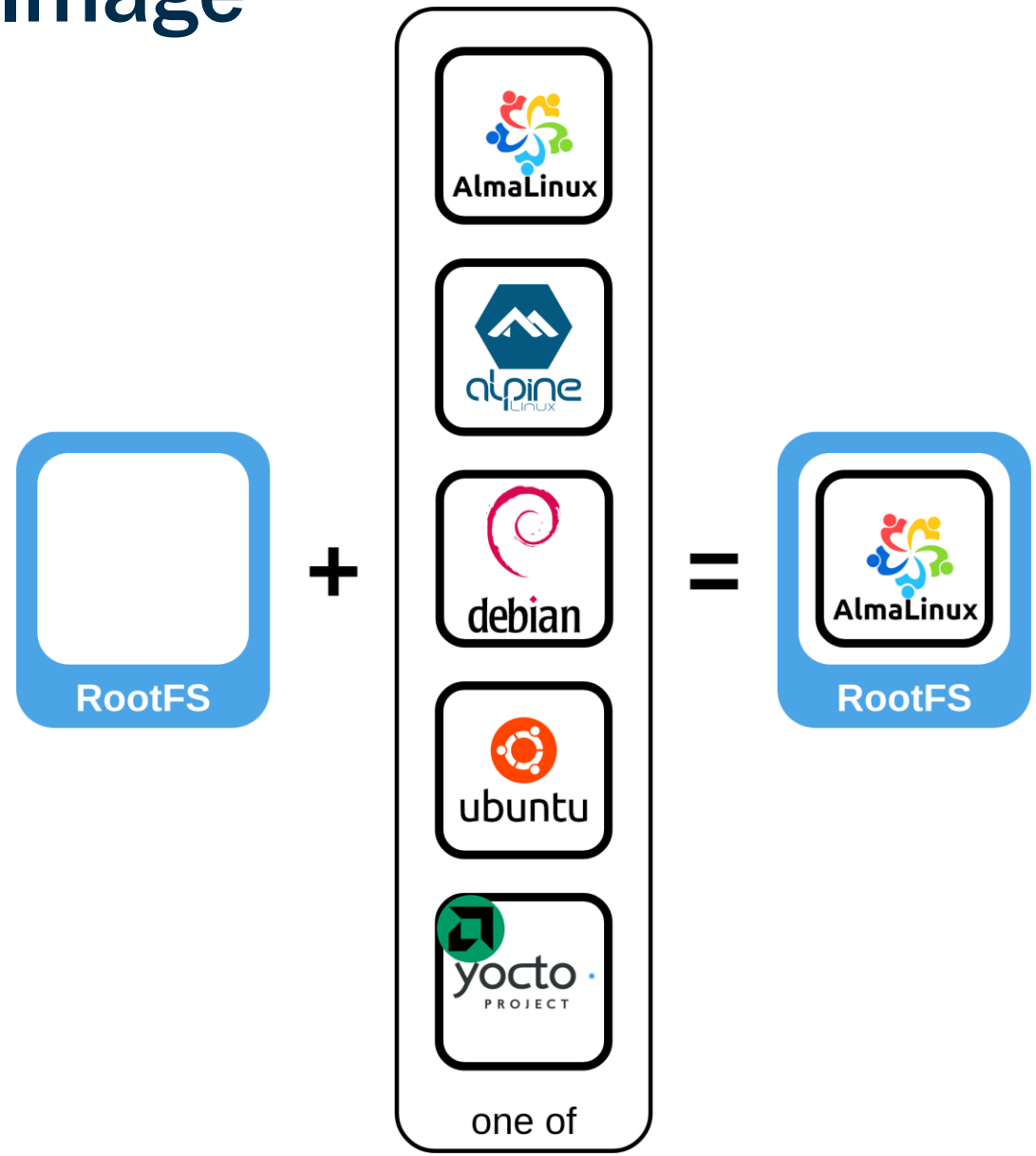
SoCks Software and Firmware Image

Builder:

- Builds one version of the block
- Multiple builders per block available
- Select one builder per block

```
rootfs:
  source: build
  builder: ZynqMP_AlmaLinux_RootFS_Builder
  project:
    release: 9
    addl_pkgs: ["python3", "nano", "rsync", "git"]
    ...
    users:
      - name: root
        pw_hash: $1$X6DWcY5/$GQ3ZFNgCGxOSgOZMT6Bkc1
    dependencies:
      kernel: temp/kernel/output/bp_kernel_*.tar.gz
      ...
  container:
    image: alma9-rootfs-builder-alma9
    tag: socks
```

RootFS section in project.yml



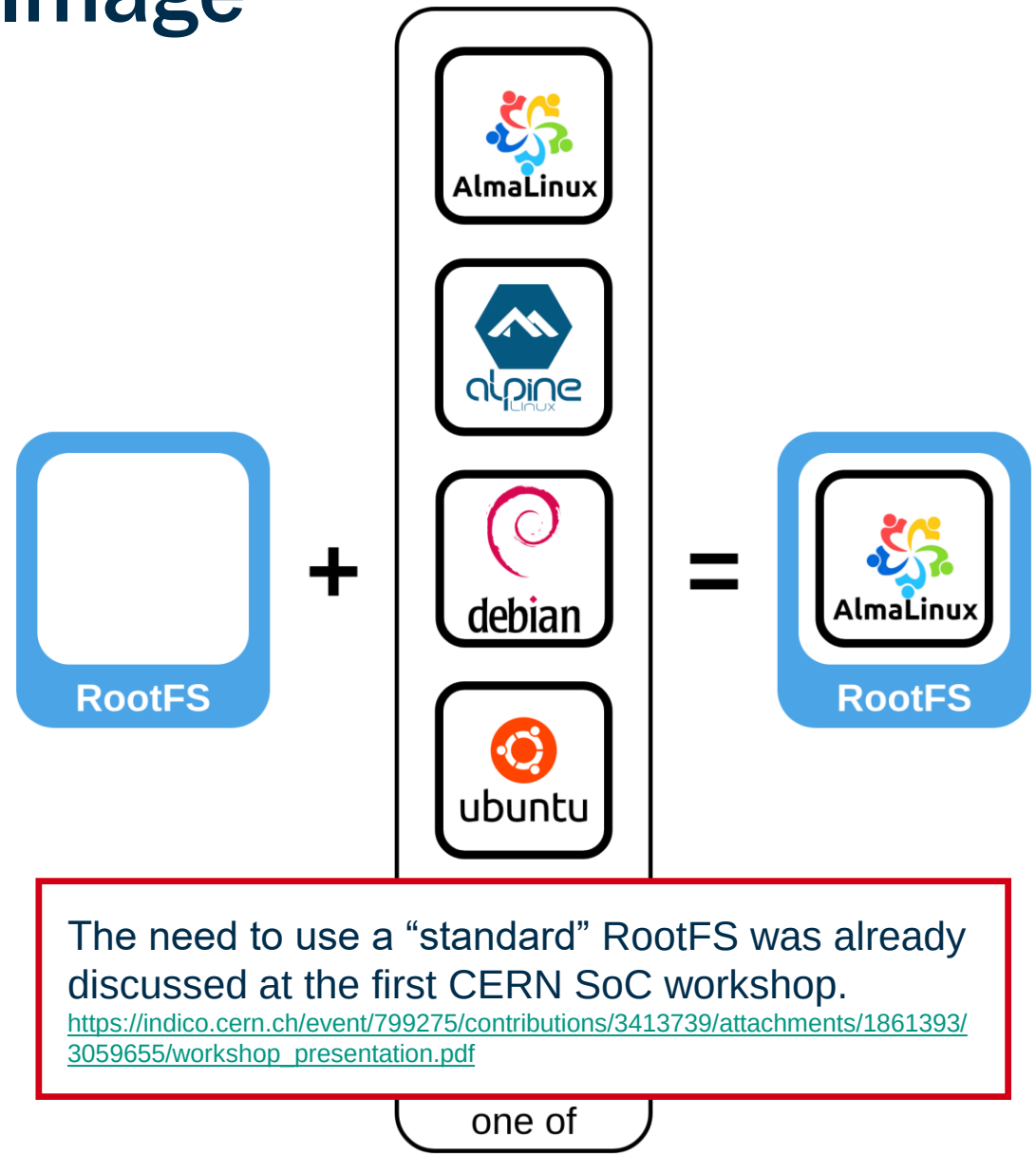
SoCks Software and Firmware Image

Builder:

- Builds one version of the block
- Multiple builders per block available
- Select one builder per block

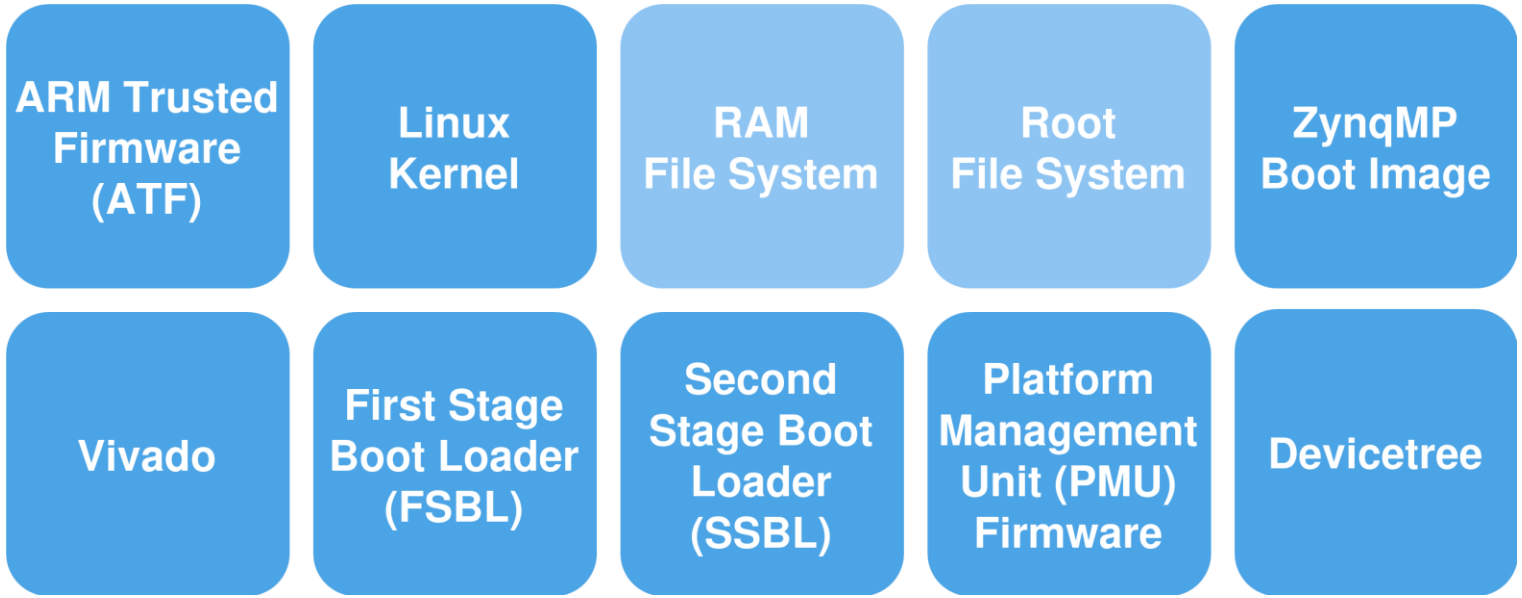
```
rootfs:
  source: build
  builder: ZynqMP_AlmaLinux_RootFS_Builder
  project:
    release: 9
    addl_pkgs: ["python3", "nano", "rsync", "git"]
    ...
    users:
      - name: root
        pw_hash: $1$X6DWcY5/$GQ3ZFNgCGxOSgOZMT6Bkc1
    dependencies:
      kernel: temp/kernel/output/bp_kernel_*.tar.gz
      ...
  container:
    image: alma9-rootfs-builder-alma9
    tag: socks
```

RootFS section in project.yml



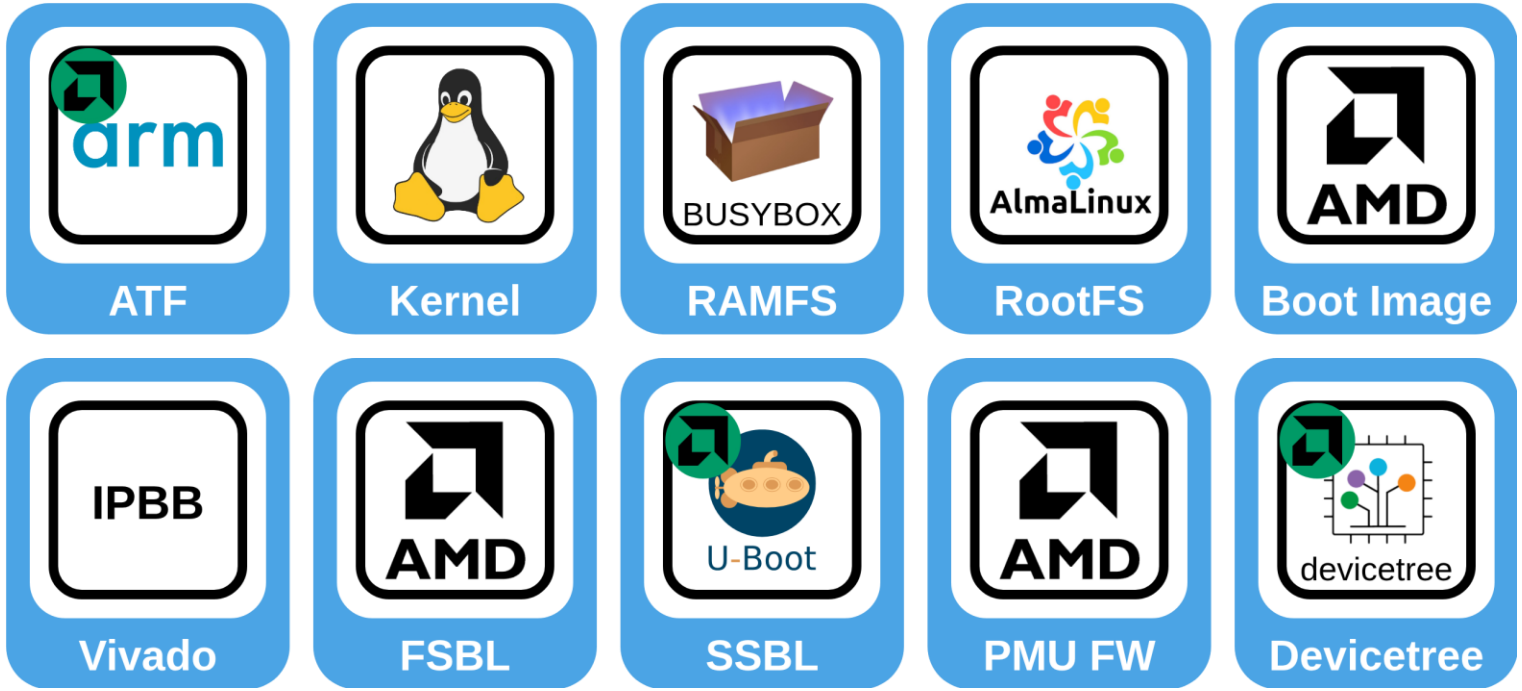
The need to use a “standard” RootFS was already discussed at the first CERN SoC workshop.
https://indico.cern.ch/event/799275/contributions/3413739/attachments/1861393/3059655/workshop_presentation.pdf

SoCks Software and Firmware Image



Layout of an AMD Zynq US+ image

SoCks Software and Firmware Image



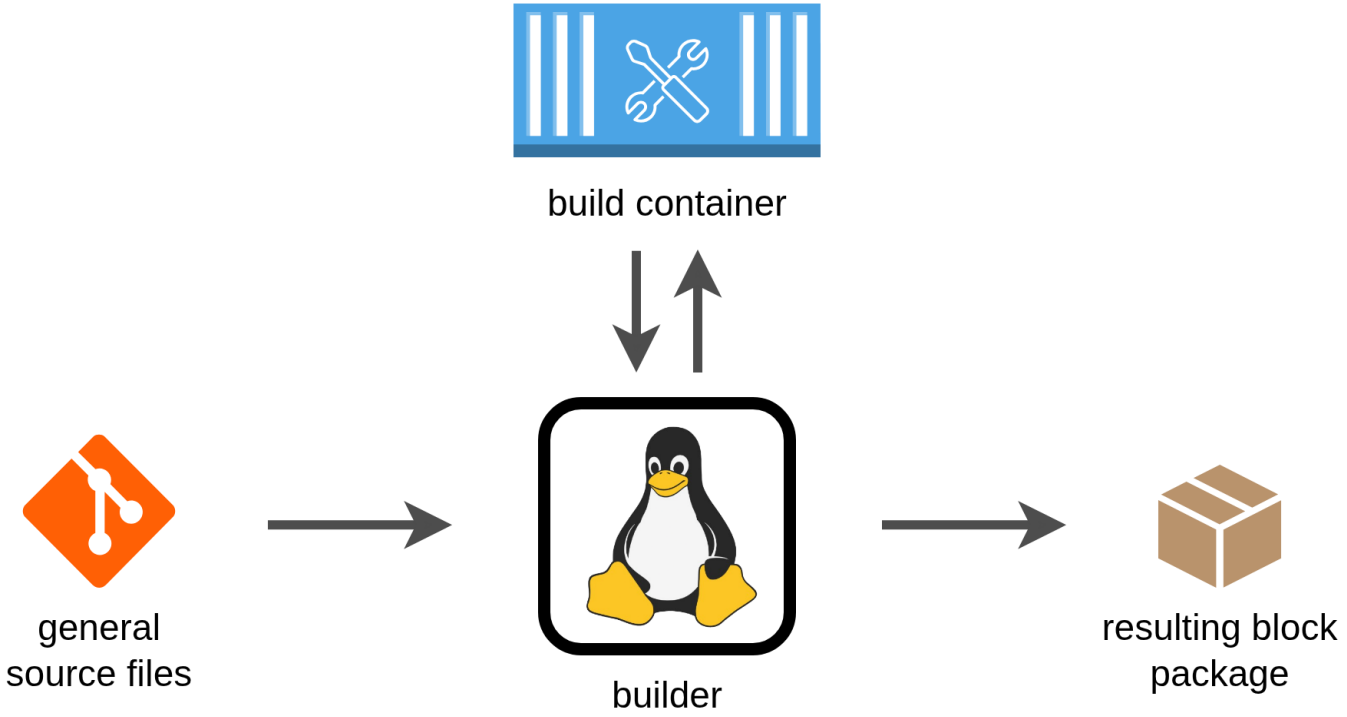
Example of a configured AMD Zynq US+ image

How is a block built?

Building the **Linux Kernel** with SoCks:

Shell Command:

```
$ socks kernel build
```

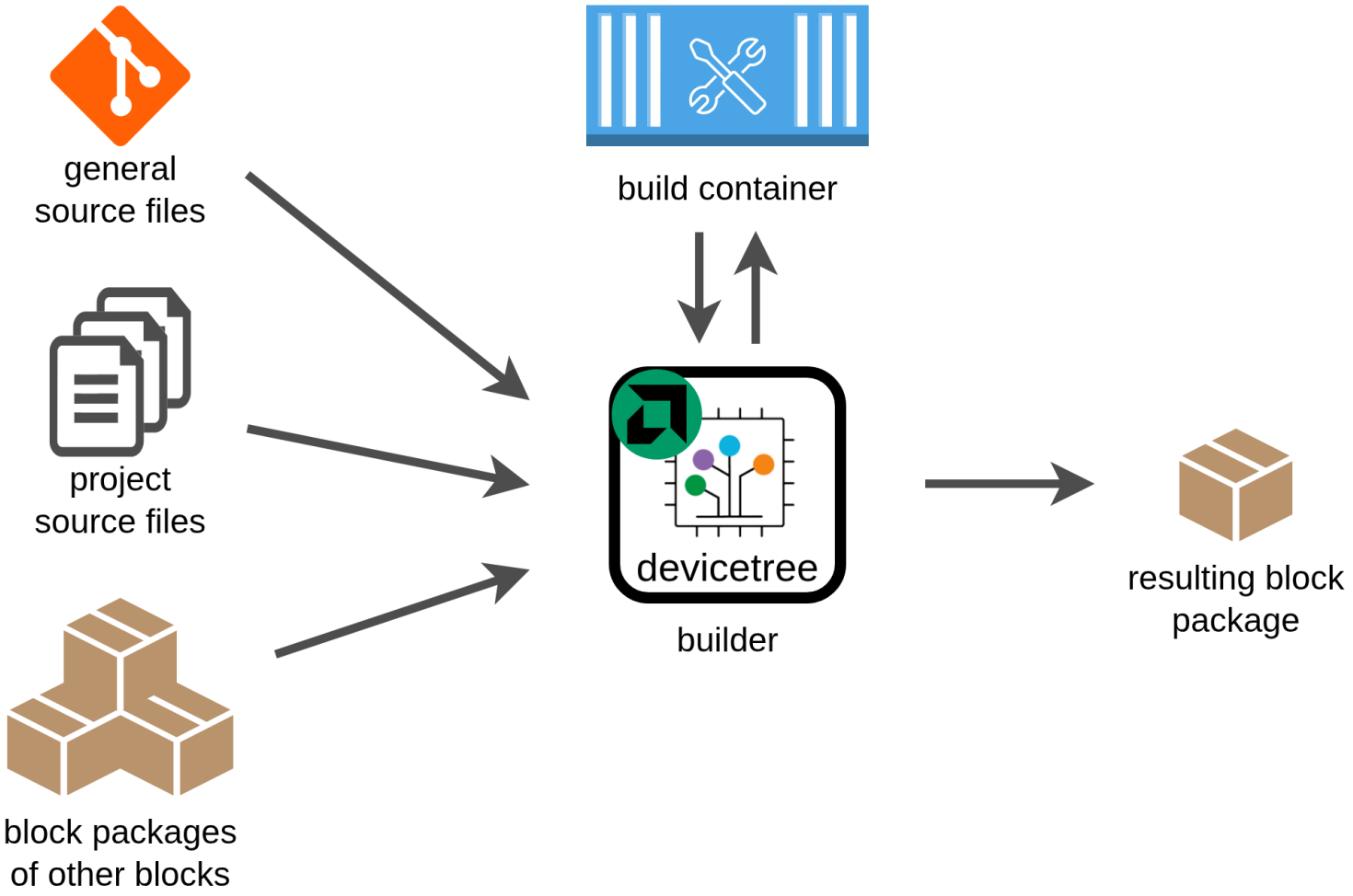


How is a block built?

Building the **Devicetree** with SoCks:

Shell Command:

```
$ socks devicetree build
```



How is a block built?

Building the **Devicetree** with SoCks:

Shell Command:

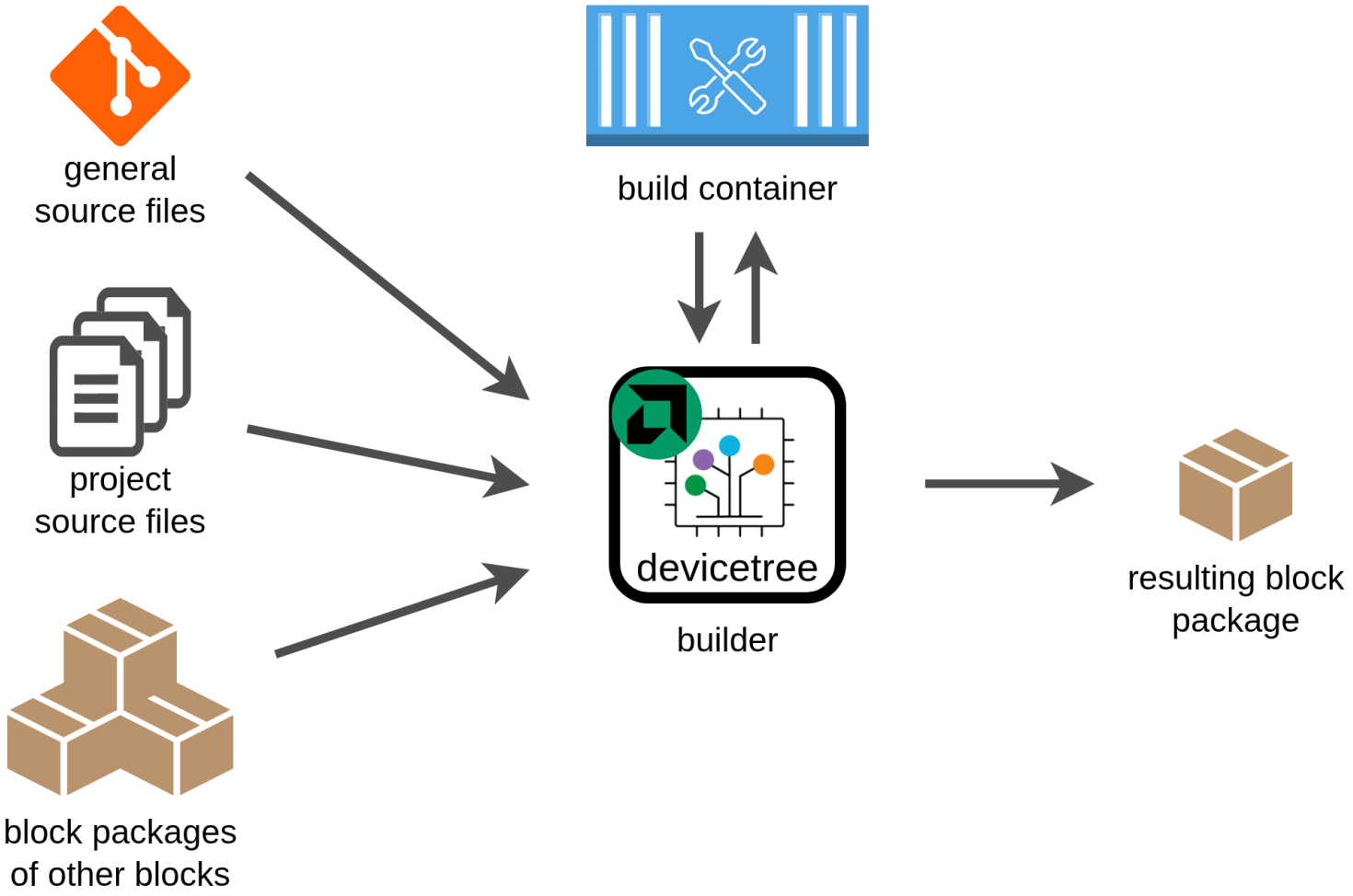
```
$ socks devicetree build
```

Supported container tools:

- Docker
- Podman
- None

Tested host systems:

- Ubuntu 24.04 LTS
- AlmaLinux 8



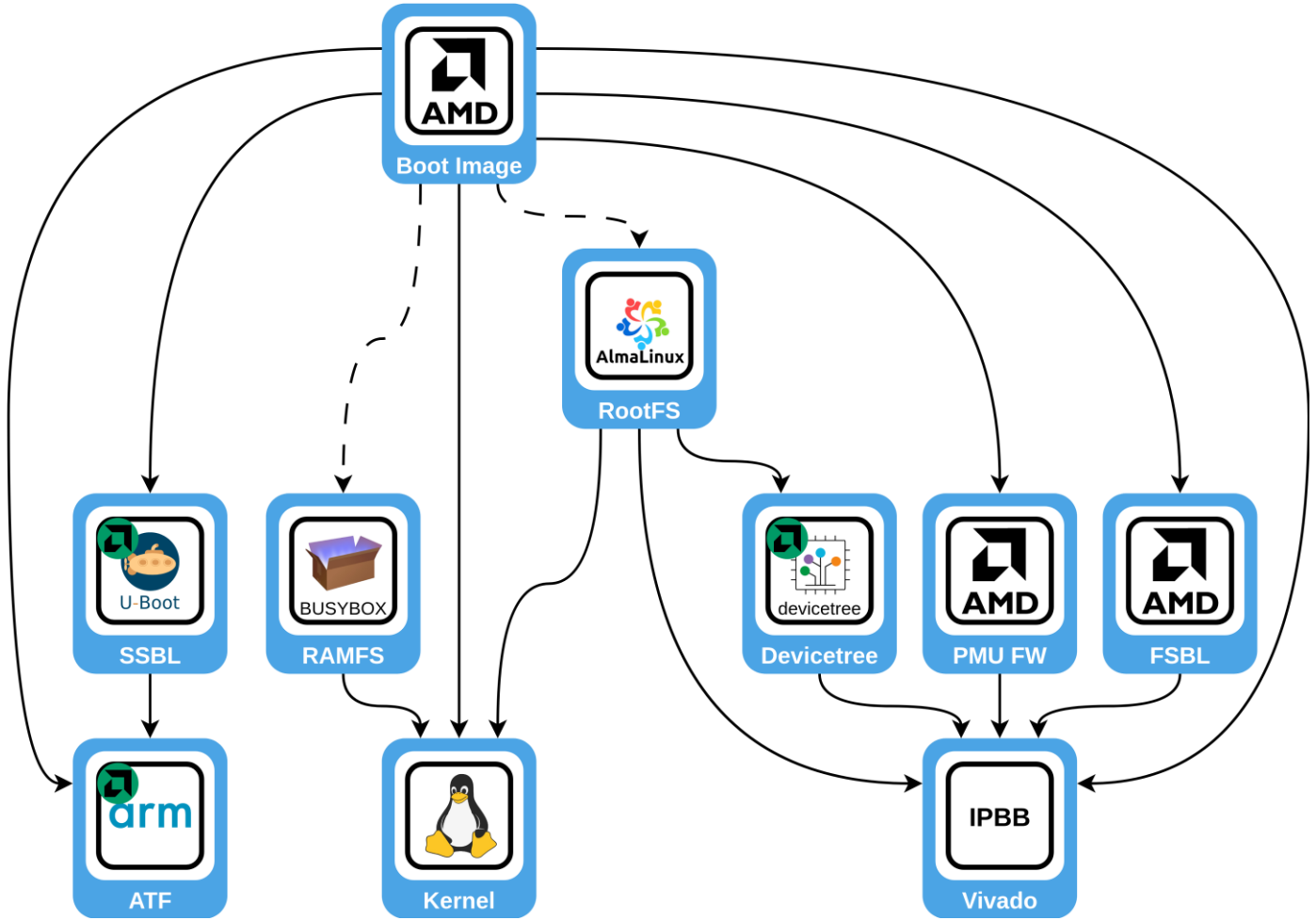
How are multiple blocks built?

Building a **Complete Image** with SoCks:

Shell Command:

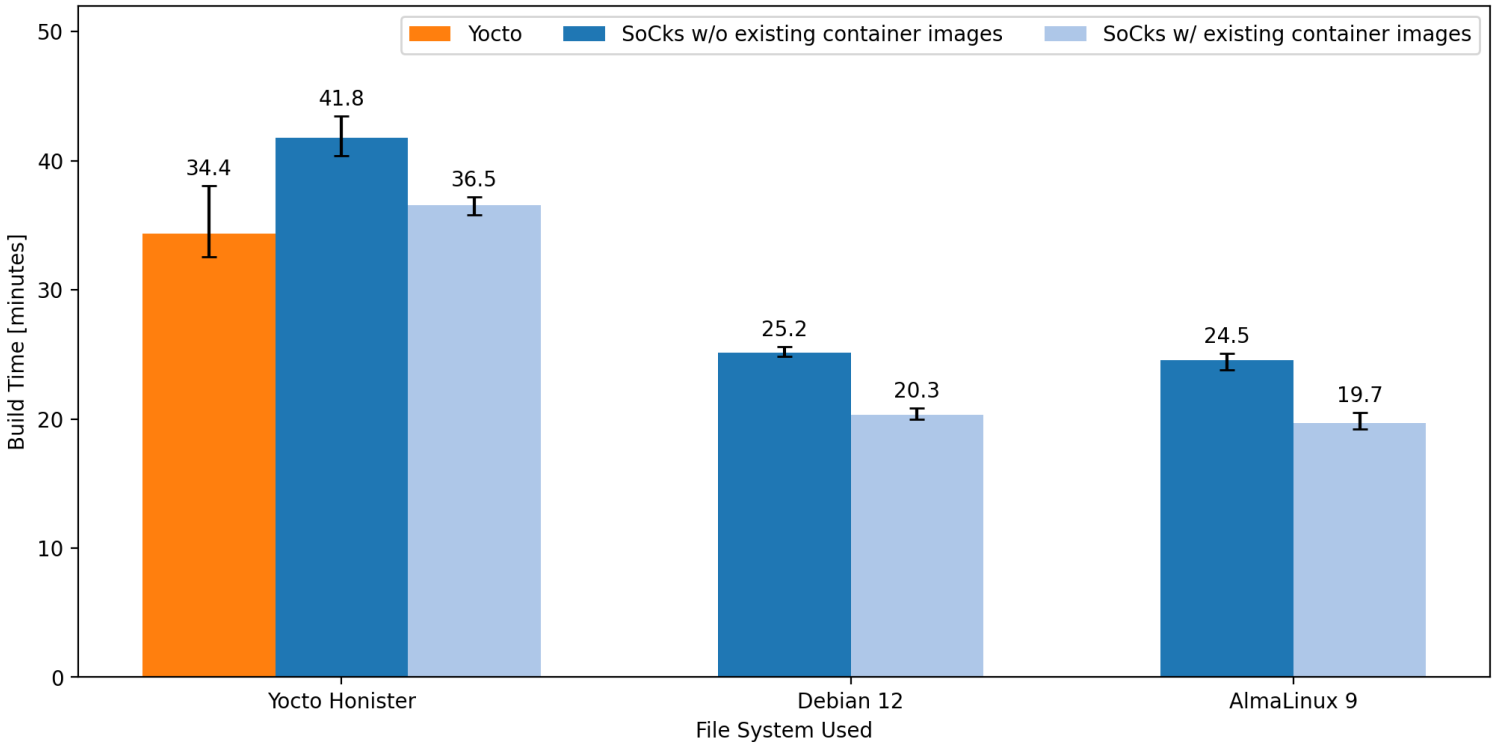
```
$ socks all build
```

- Sequential building of blocks, internal parallelization
- Processing sequence is determined by dependencies



Build Performance – Complete Images

- Equivalent Yocto and SoCks projects
- Linux OS files system has major influence on build time
- SoCks is fast in building complete images
  - Only with a Yocto file system SoCks is slower than Yocto
  - SoCks uses Yocto internally to create the Yocto file system
- Container images are only built if they do not already exist

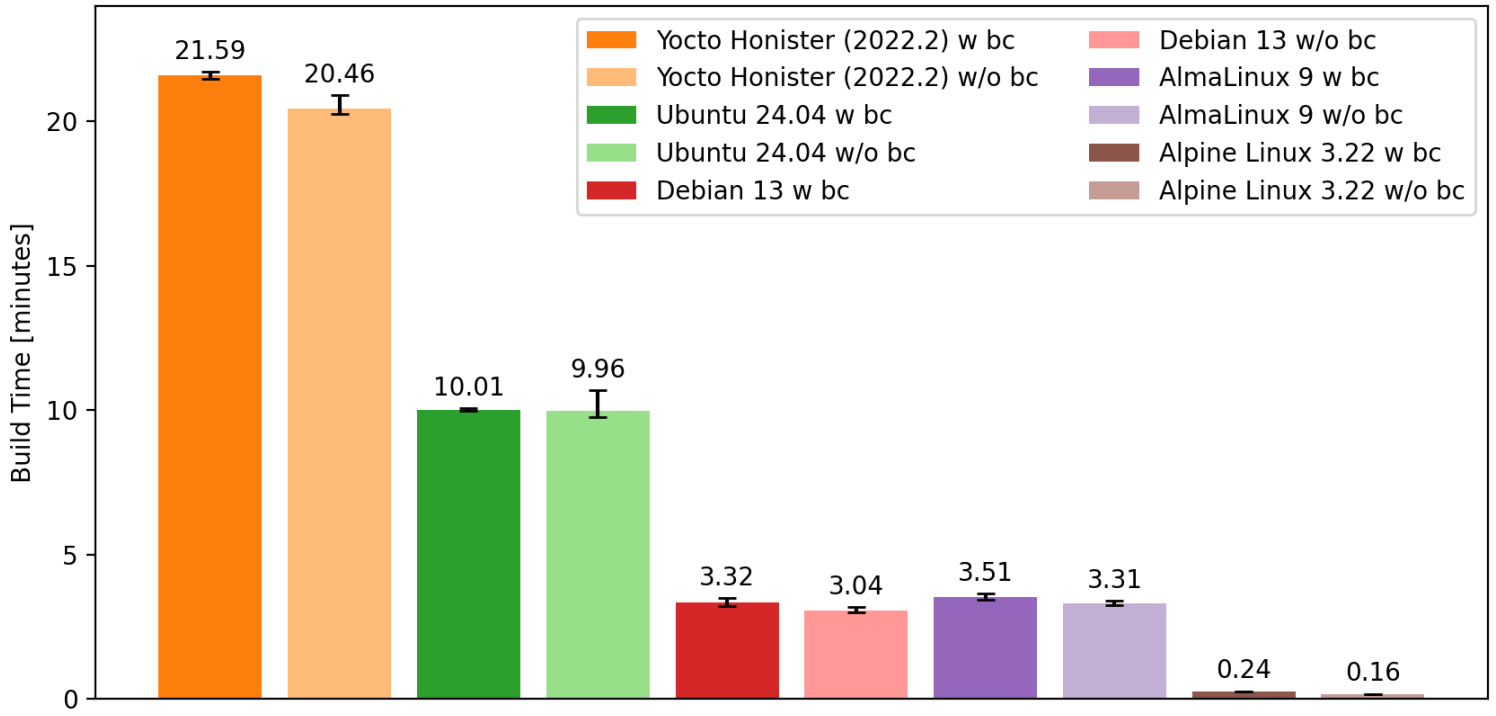


Test image: ZCU102

Test system: Intel Core i9 14900K, 128 GB of DDR5 memory, 2 TB Samsung 990 EVO NVMe SSD

Build Performance – OS File System Comparison

- All file systems in this test were built with SoCks
- Yocto is the most demanding because it is built from source
- All other file systems are built from official binary packages
- Yocto and Alpine Linux images are very similar in size

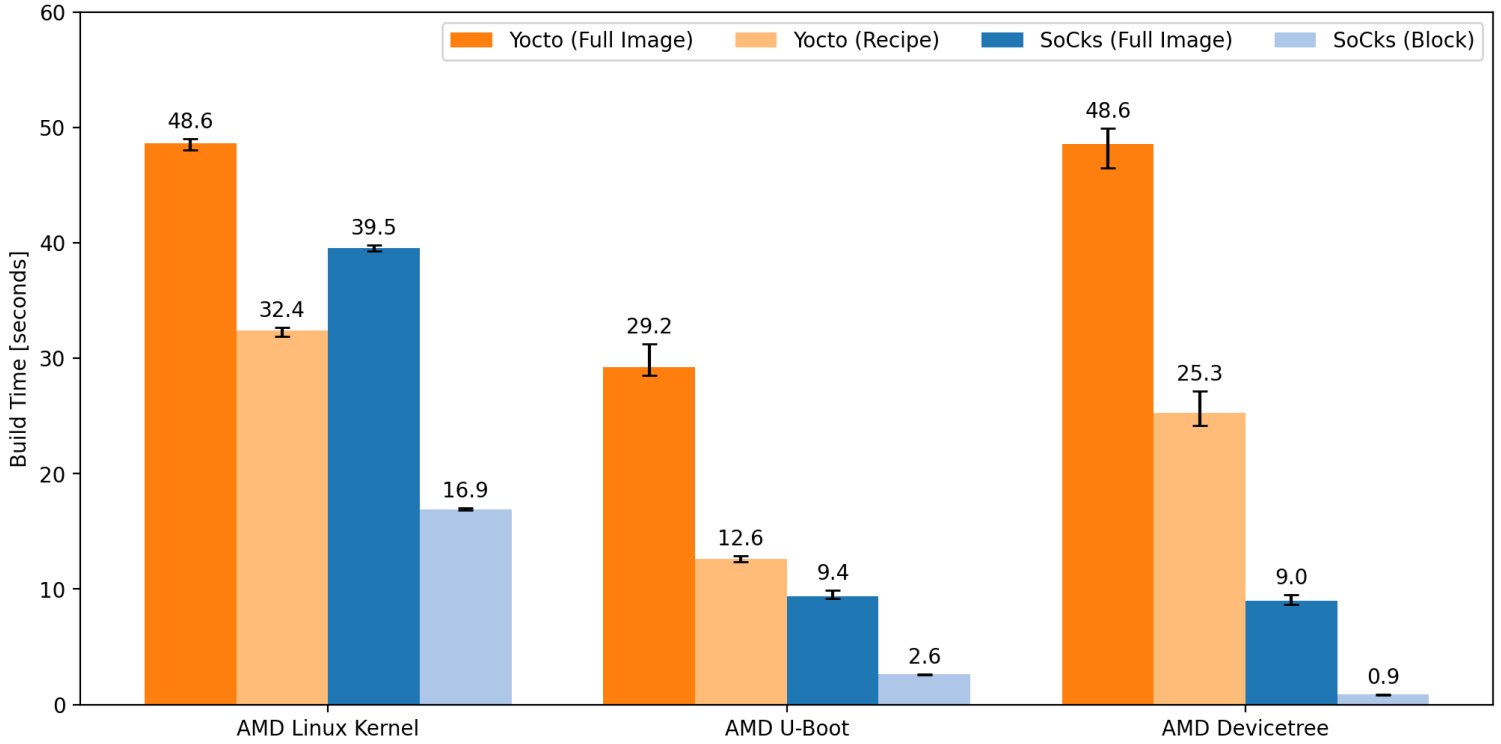


Test image: ZCU102

Test system: Intel Core i9 14900K, 128 GB of DDR5 memory, 2 TB Samsung 990 EVO NVMe SSD

Build Performance – Rebuild

- SoCks is fast at building components and images after source code modifications
- Only one line of code was changed for the measurements
  - Value of a variable toggled
- Devicetree modifications are particularly relevant in practice



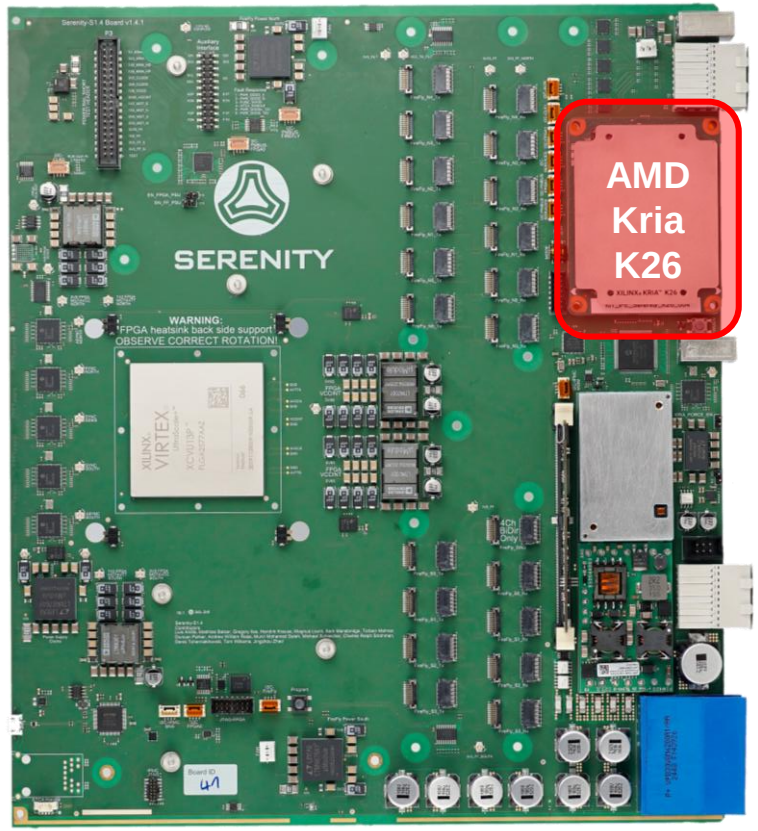
Test image: ZCU102

Test system: Intel Core i9 14900K, 128 GB of DDR5 memory, 2 TB Samsung 990 EVO NVMe SSD

Supported SoC Architectures

- SoCks is currently mainly used at KIT
- Various devices have already been tested
- Most advanced for AMD Zynq US+
- Raspberry Pi for demonstration purposes

AMD Zynq US+	AMD Versal	Raspberry Pi
Kria K26 SoM	VEK280 AI Edge Eval. Kit	Raspberry Pi 4B
ZCU102 MPSoC Eval. Kit		Raspberry Pi 5
ZCU208 RFSoC Eval. Kit		
ZCU216 RFSoC Eval. Kit		
DTS-100G (custom)		
iWave iW-RainboW-G42M SoM		

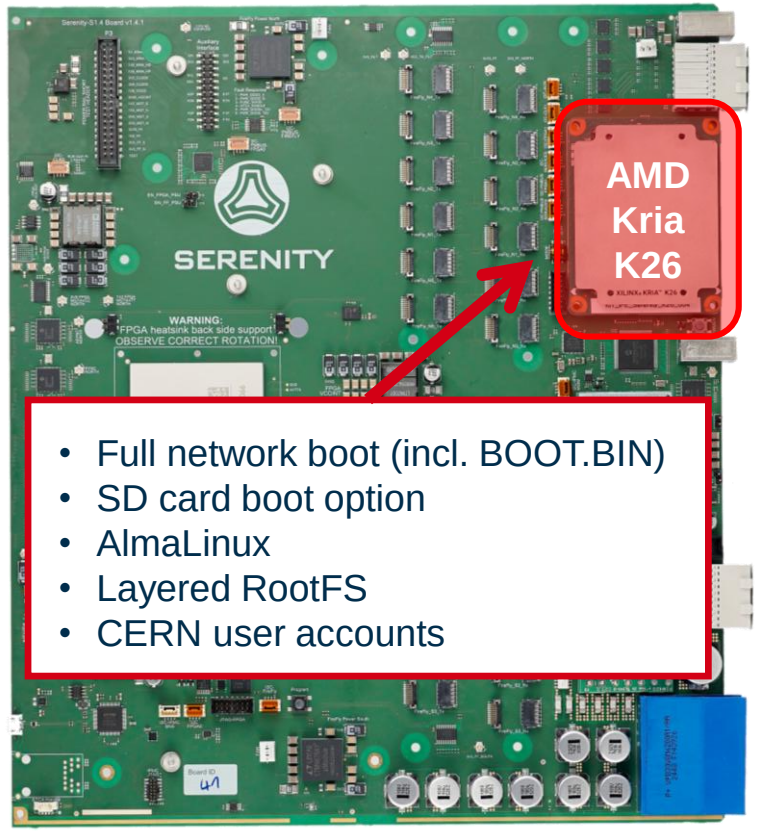


Current usage at CERN: AMD Kria K26 SoM on Serenity-S

Supported SoC Architectures

- SoCks is currently mainly used at KIT
- Various devices have already been tested
- Most advanced for AMD Zynq US+
- Raspberry Pi for demonstration purposes

AMD Zynq US+	AMD Versal	Raspberry Pi
Kria K26 SoM	VEK280 AI Edge Eval. Kit	Raspberry Pi 4B
ZCU102 MPSoC Eval. Kit		Raspberry Pi 5
ZCU208 RFSoC Eval. Kit		
ZCU216 RFSoC Eval. Kit		
DTS-100G (custom)		
iWave iW-RainboW-G42M SoM		

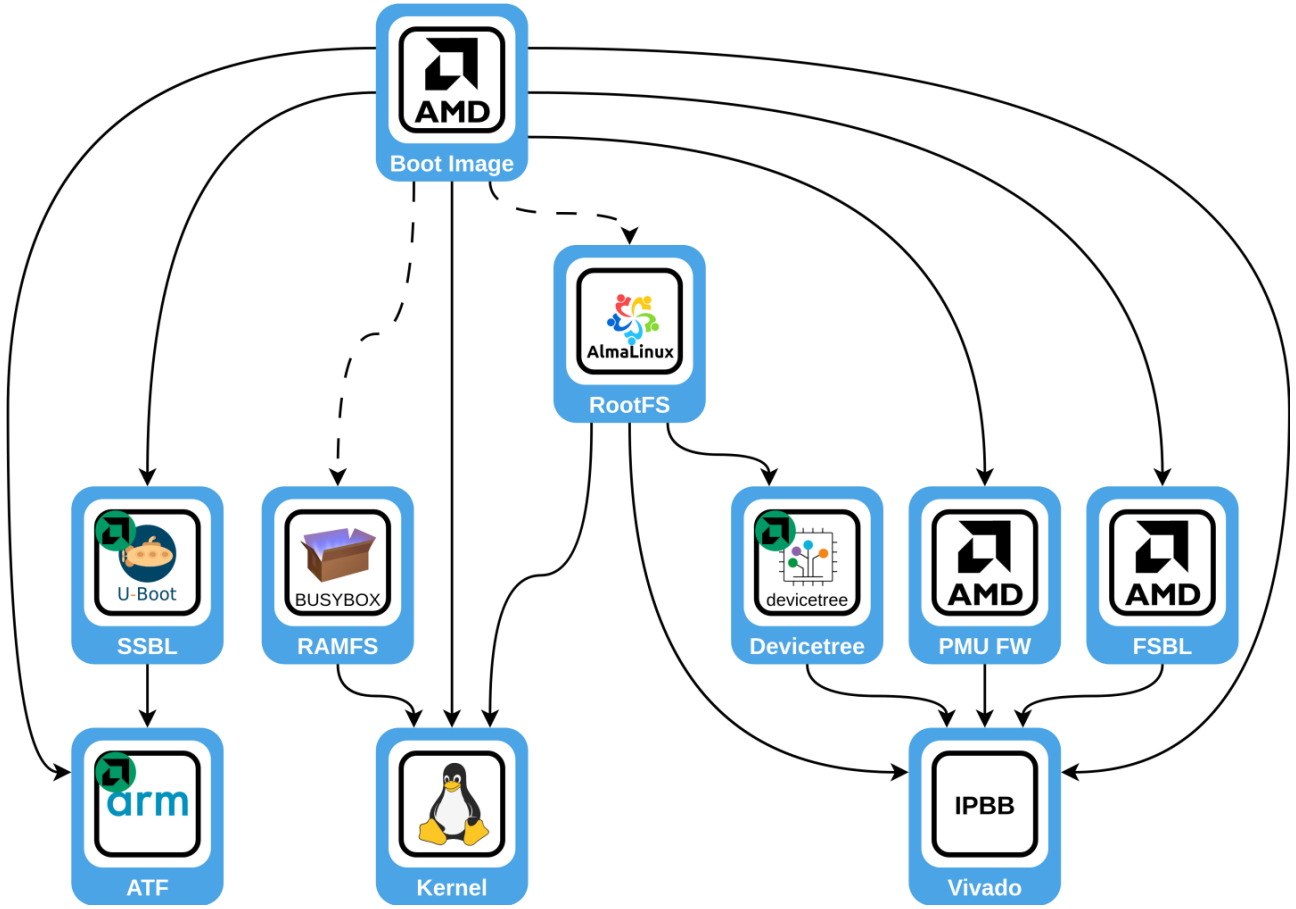


Current usage at CERN: AMD Kria K26 SoM on Serenity-S

Conclusion

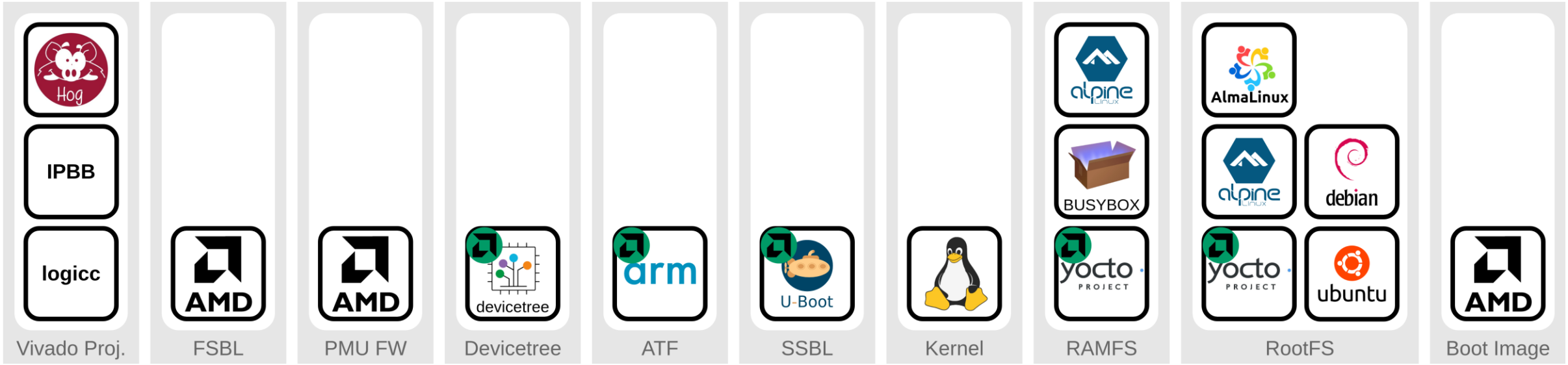
- SoCks is a lightweight framework to build SoC images
- Containerized builds
- Modular and expandable
- Fully compatible to GitLab CI/CD
- Can replace PetaLinux in many cases

- **SoCks is open source!**
  - https://github.com/kit-ipe/SoCks
  - https://arxiv.org/abs/2510.15910



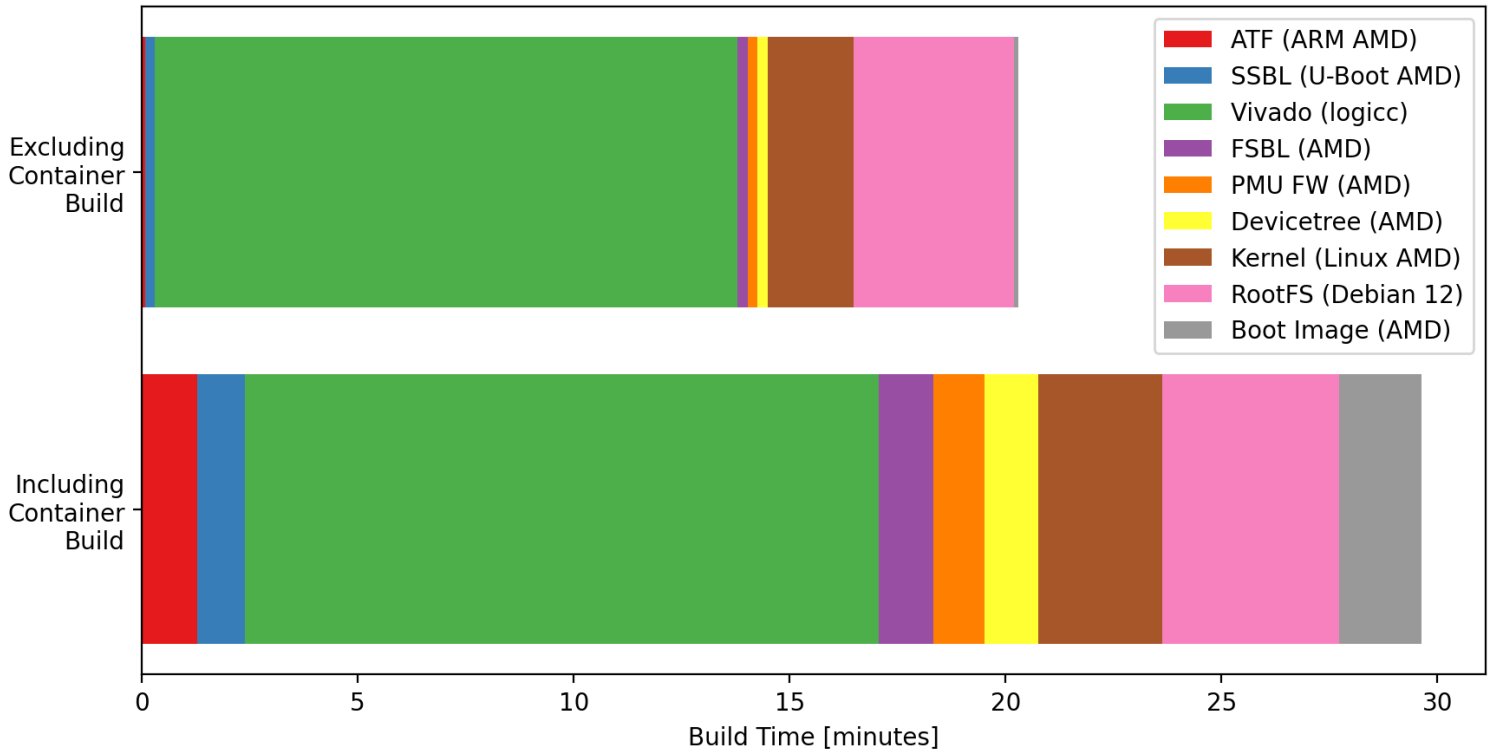
Backup

Overview of Builders Currently Available in SoCks



Build Performance – Individual Blocks of a SoCks Image

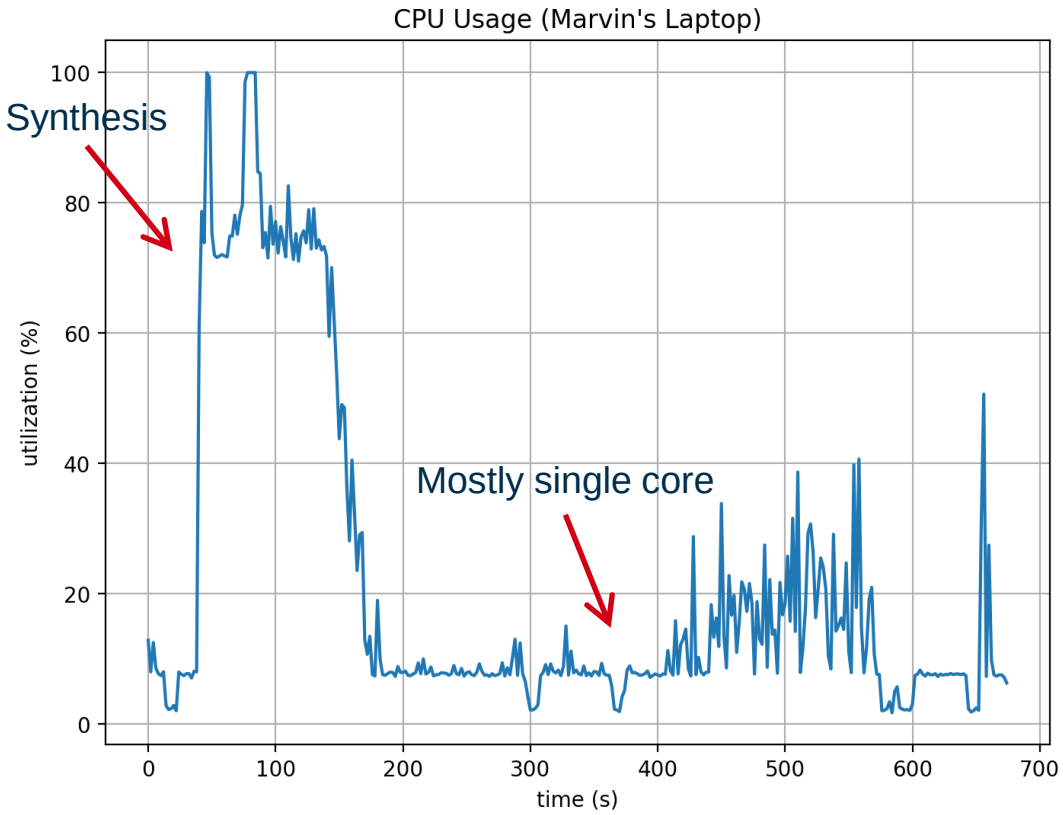
- The Vivado project has the longest building time
  - Depends heavily on the Vivado project
- The RootFS of the OS has the second longest building time
  - Depends heavily on the distribution/framework
- The Linux kernel has the third longest building time
  - Depends heavily on the kernel configuration



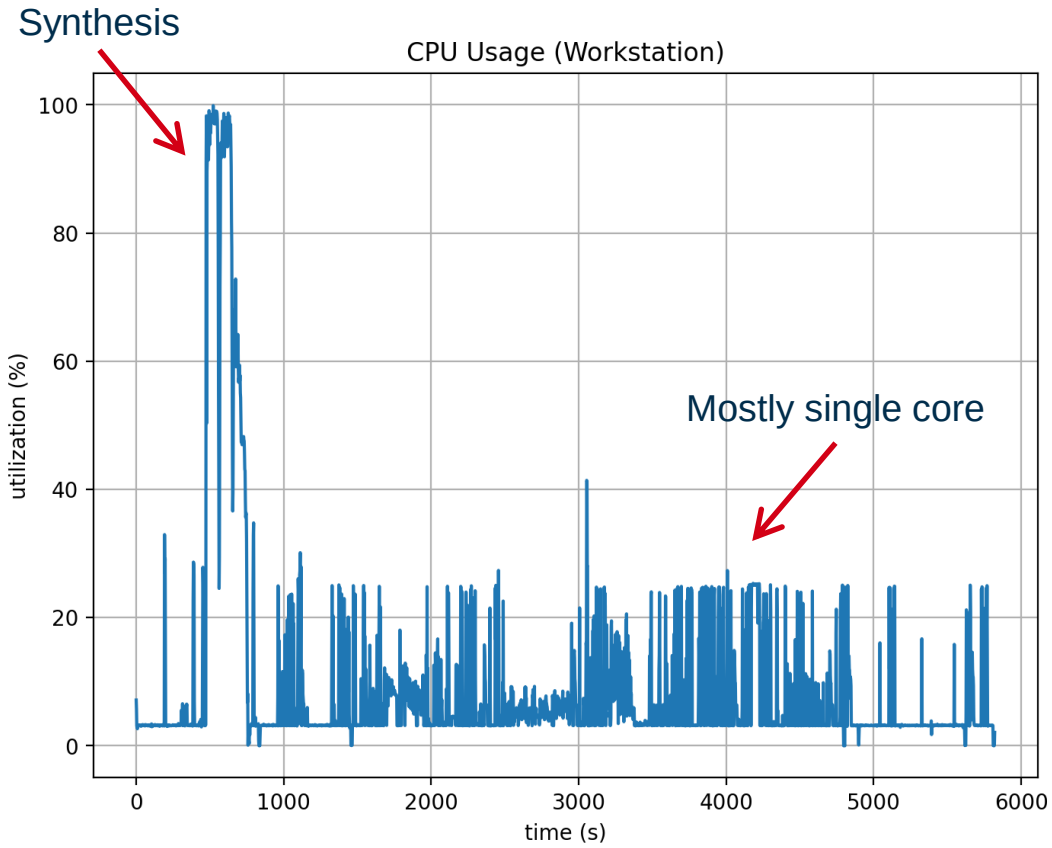
Test image: ZCU102

Test system: Intel Core i9 14900K, 128 GB of DDR5 memory, 2 TB Samsung 990 EVO NVMe SSD

CPU Utilization During a Vivado Project Build



Project: Vivado project A
Test system: Lenovo ThinkPad P14s Gen 2a, AMD Ryzen 7 PRO 5850U, 32 GB of DDR4 memory, 1 TB SK Hynix HFS001TDE9X081N SSD



Project: Vivado project B
Test system: Intel Core i9 14900K, 128 GB of DDR5 memory, 2 TB Samsung 990 EVO NVMe SSD