What is a System Device Tree?
• Modern SoCs are very heterogenous
  • MPSoC: A53s, R5s, PMU, MicroBlazes

• System Software needs a lot of HW info
  • Memory allocated for each domain
    • Including shared pages
  • Devices assigned to each domain
  • Addresses of memory and registers
    • Same device can have different addresses
  • Topologies (clocks, busses, …)

• Allocation and configuration is complex
  • Typically done in an ad-hoc way
    • Editing Device Trees and #defines
  • Especially tricky for shared resources
    • E.g. shared pages for OpenAMP/virtIO

• Industry standards and common tools needed
  • One well-defined true source for all configuration
    • Common for Linux, firmware, RTOSes, etc.
  • Open source tooling to manipulate configuration
    • Split up allocated resources to “Execution Domains”
Device Trees and System Device Trees

- **Device Trees (DTs)** express HW information relevant to Operating Environments
  - Been used by PPC and ARM SSW to define HW that can not be dynamically discovered
  - Used by uboot, Linux, Xen and increasingly being used by RTOS vendors

- **Device Trees describes HW nodes and topologies**
  - Traditional Device Trees are only describing the world seen from one Address Space

- **Additional system level Device Tree information is proposed**
  - A System Device Tree (S-DT) describes all HW that later can be divided into different partitions

- **System DT additions include two parts:**
  1. DeviceTree.org specification and tooling additions
     - Describing multiple cpu clusters and corresponding views of their address spaces
     - Enabling source-to-source translations by adding options to keep labels and comments
  2. AMP configuration information
     - Resource allocation using Execution Domains
     - Using the a special Device Tree section to specify AMP configuration
     - Specification of shared resources, such as pages for virtIO buffers
     - Intent is to align with hypervisor information (e.g. Xen Dom0-less configuration)
What is an “Execution Domain”? 
Execution Domains and Operating Environments

- **Domains**
  - A Domain is a separate address space, including devices
  - Defined by cores clusters, Execution Levels and security environments

- **Core clusters – Heterogeneous cores**
  - E.g. A53s, R5s, PMU, MicroBlazes

- **Execution Levels (EL)**
  - EL0 – User space
  - EL1 – OS
  - EL2 – Hypervisor
  - EL3 – Firmware

- **Security Environments**
  - TrustZone (TZ) – HW protecting resources (e.g. memory)
  - Trusted Execution Environment (TEE) – SEL1

- **Operating Environments (OE)**
  - An OE is the system SW that runs in a Domain, including:
    - Linux (including Android), Free and commercial RTOS’s
    - Bare metal (no OS), Hypervisors
    - Firmware/boot loaders – Trusted FW, PLM, PMU FW, uboot, ...

---

![Diagram of execution domains and operating environments](image-url)
What’s the difference between System DT and DT?
The current proposal: new concepts

- **Hardware description**
  - `cpus,cluster`: multiple top-level nodes to describe heterogeneous CPU clusters
  - `indirect-bus`: a new type of bus that does not automatically map to the parent address space
  - `address-map`: a property to express different address mappings of CPU clusters; it can map indirect-buses

- **AMP Configuration**
  - execution domains
Hardware Description: an example

```c
/* default cluster */
cpus {
    cpu@0 {
    }
    cpu@1 {
    }
};

/* additional R5 cluster */
cpus_r5: cpus-cluster@0 {
    compatible = "cpus,cluster";
    /* specifies address mappings */
    address-map = <0xf9000000 &amba_rpu 0xf9000000 0x10000>;
    cpu@0 {
    }
    cpu@1 {
    }
};

amba_rpu: indirect-bus@f9000000 {
    compatible = "indirect-bus";
};
```
Why we have a default?

- It is convenient to have an execution domain that owns everything by default
- It is also very common: e.g. Linux running on a Cortex-A cluster
- It turns system device tree into an addition to device tree
- It makes it more natural to introduce system device tree concepts to the device tree spec
- It allows us to maintain compatibility with existing systems, i.e. Linux booting on system device tree
How do we describe interrupts?
Interrupts

- Multiple clusters
- Each cluster sees only its own interrupt controller
- Other hardware hard-wired to a specific bus can be specified the same way

```c
/* default cluster */
cpus {
};

/* additional R5 cluster */
cpus_r5: cpus-cluster@0 {
    compatible = "cpus,cluster";
    /* specifies address mappings */
    address-map = <0xf9000000 &amba_rpu 0xf9000000 0x10000>;
};

/* bus only accessible by cpus */
amba_apu: bus@f9000000 {
    compatible = "simple-bus";
    gic_a72: interrupt-controller@f9000000 {
    }
};

/* bus only accessible by cpus_r5 */
amba_rpu: indirect-bus@f9000000 {
    compatible = "indirect-bus";
    gic_r5: interrupt-controller@f9000000 {
    }
};
```
Interrupts

All devices have interrupts routed to both interrupt controllers
How do we dedicate assignable resources to CPUs clusters?
Configuration: execution domains

- An **execution domain** is a collection of software, firmware, and board configurations that enable an operating system or an application to run a CPUs cluster.
  - **cpus**: physical CPUs where the software is running
  - **memory**: memory assigned to the domain
    - Memory ranges can be shared across multiple domains, e.g. for communication
  - **access**: devices assigned to a domain

```c
domains {
    openamp_r5 {
        compatible = "openamp,domain-v1";
        cpus = <&cpus_r5 0x2 0x80000000>;
        memory = <0x0 0x0 0x0 0x80000000 0x0 0x10000000 0x0 0x1000>;
        access = <&can@ff060000>;
    }
};
```
How do we configure Bus Firewalls?
Bus Firewalls & Device Assignment

- Devices are assigned to execution domains using **access**
- **memory + access** have the information necessary to configure bus firewalls
  - Memory ranges dedicated to one execution domain
  - Devices dedicated to one execution domain
- In the example below, the bus firewall can be configured to allow access to the following address ranges only from the Cortex-R5 cluster:
  - 0 - 0x80000000
  - 0xff060000 - 0xff066000

```c
domains {
    openamp_r5 {
        compatible = "openamp,domain-v1";
        cpus = <&cpus_r5 0x2 0x80000000>;
        memory = <0x0 0x0 0x0 0x80000000>;
        access = <&can@ff060000>;
    }
};
```
Bus Firewalls & Priorities

- The bus firewall configuration can be derived from memory, access, and the capability of the bus firewall
  - It can be implemented as a backend to lopper
- Bus firewalls might not be able to protect everything
- We need to set priorities for bus firewall protection
- From one execution domain point of view:
  - Priorities for protecting my memory/MMIO regions from foreign accesses (most important)
  - Priorities for protecting others from my memory accesses
  - We might need higher granularities, to specify priorities per device, per memory range

```c
domains {
  openamp_microblaze {
    compatible = "openamp,domain-v1";
    priority_self = <9>;
    memory = < ... >
    access = < ... >
  }
};
```
What about chosen and reserved-memory?
Chosen & Reserved-Memory

- **chosen** and **reserved-memory** are top-level nodes dedicated for configurations.
- In the system device tree, they are dedicated to the configuration of the **default** CPUs cluster.
- Other execution domains have their own chosen and reserved-memory nodes:

```c
/* configurations for the default cluster */
chosen {
};
reserved-memory {
};

/* execution domains configuration */
domains {
    openamp_r5 {
        compatible = "openamp,domain-v1";
        cpus = <&cpus_r5 0x2 0x80000000>;
        memory = <0x0 0x0 0x0 0x80000000>;
        access = <&can@ff060000>;
        chosen {
            bootargs = "console=ttyPS0,115200";
        };
        reserved-memory {
            [...]
        };
    };
};
```
What is “Lopper”? 
Lopper

- **Lopper**
  - Is a tool for manipulating System Device Trees
  - Primary goal is to produce standard devices trees to support existing platforms/OSs
    - Produces any number of outputs through plug-ins: device trees, generated code, custom, etc
  - Integrates with various development workflows
  - Is data driven (there is no magic!)

- **A few details:**
  - OpenSource, BSD-3 License
    - [https://github.com/OpenAMP/open-amp/wiki/System-Device-Trees#Lopper](https://github.com/OpenAMP/open-amp/wiki/System-Device-Trees#Lopper)
  - Written in python, using libfdt for tree manipulations
  - Works with dts and dtb inputs
  - Supports basic/simple operations (lops) and more complex python assist modules
    - Depending on the task, both can be used
  - Flexible output is provided via python backends
  - Performs validation and consistency checking during output
What are the components of Lopper?
Lopper Components

Standards Based Inputs
- devicetree.org
  - bindings
  - schemas
  - ...
- system device tree
- device / board dts
- overlays
- lops

Open Source Tools
- Lopper
  - python3 frontend
  - c frontend
  - libfdt
  - pyfdt
  - dtc
  - cpp
  - libdevice
  - dtc: schema validation
- Lopper assists

Backends
- dtb -> dts
- dtb -> c
- device tree blob
- device tree
- c device-drievers

Conformant Outputs
- lopper-validator
- device tree blob
- Linux
- dev pipeline
- RTOS
- Bare Metal

Lopper assists dev pipeline
How do I run Lopper to create a traditional DT?
Generating a traditional DT (1/3)

- **Inputs:**
  - System Device tree
  - Domain node
  - Lopper operations (custom, built-in, or both)

- **Outputs:**
  - Standard Device tree
  - Optional: Custom device trees

- **What lopper does:**
  - Applies operations to the tree as specified in the lopper operations file (lops)
    - If specified, Finds the specified domain node
      - Applies logic based on the domain node
        - Built-ins, or via python assist
  - Performs built-in operations to remove non-standard elements
  - Outputs the modified system device tree
    - Either as a raw dump, or as a validated "pretty printed" version
    - Either way, the output is standard device tree
Generating a traditional DT (2/3)

% lopper.py --pretty -i lops/lop-load.dts -i lops/lop-domain-r5.dts device-trees/system-device-tree-versal-v2.dts output/linux-r5.dts

SDT summary:
  system device tree: ['system-device-tree-versal-v2.dts']
  lops: ['lops/lop-load.dts', 'lops/lop-domain-r5.dts']
  output: output/foo.dts
...
[INFO]: deleting node /opux
[INFO]: resetting all refcounts
[INFO]: tracking access to node /chosen/openamp_r5
[INFO]: tracking access to node /chosen
[INFO]: tracking access to node /cpus_r5
...
[INFO]: deleting node /amba/can@ff060000
[INFO]: deleting node /amba/can@ff0700000
...
[INFO]: deleting node /amba/pc1@fca10000
[INFO]: deleting node /amba/watchdog@fda4d0000
[INFO]: deleting node /amba/zynqmp_ipl
...
[INFO]: modify property found: /opux_r5/::/cpus/ to opux/
[INFO]: renaming /opux_r5/ to opux/
...
[DBG+]: outfile is: linux.dtb
[DBG+]: output selected are: ['*']
[INFO]: dts format detected, writing ./linux-dtb
...
Generating a traditional DT (3/3)

% lopper.py --pretty -i lops/lop-load.dts -i lops/lop-domain-r5.dts device-trees/system-device-tree-versal-v2.dts output/linux-r5.dts

```
% cat system-device-tree-versal-d2.dts | grep {} | wc -l
84

% cat device-trees/system-device-tree-versal-v2.dts | grep -A 10 "cpus {}"

  cpus: cpus {
    #address-cells = <0x1>;
    #size-cells = <0x0>;
    #cpus-mask-cells = <0x1>;
    compatible = "cpus,cluster";

    cpu@0 {
      compatible = "arm,cortex-a72",
      "arm,armv8";
    }

% cat output/linux-r5.dts | grep {} | wc -l
41

% cat output/linux-r5.dts | grep -A 10 "cpus {}"

cpus: cpus {
  #address-cells = <0x1>;
  #size-cells = <0x0>;
  #cpus-mask-cells = <0x1>;
  compatible = "cpus,cluster";
  #ranges-size-cells = <0x1>;
  #ranges-address-cells = <0x1>;
  address-map = <0xf1000000 &amba 0xf1000000 &memory00000000 0x0 0x80000000 0x0 &tcmffe90000 0xffe90000 0x10000>;
  phandle = <0x28>;

  cpu@1 {
    compatible = "arm,cortex-r5";
  }
```
Lopper: RTOS without DT support?
Lopper and non-DT aware OSs (1/2)

- **Inputs:**
  - System Device tree or Standard device tree
  - Optional: Domain node
  - Lopper operations (custom, built-in, or both)
  - Lopper assist module for the OS

- **Outputs:**
  - Device tree: partial, standard or unmodified
  - Code / headers for the OS

- **What lopper does:**
  - Applies operations to the tree as specified in the lopper operations file (lops)
    - Calls input/output assists specific to the target OS
  - Performs built-in operations to remove non-standard elements
  - Outputs the Device tree
    - Either as a raw dump, or as a validated “pretty printed” version
  - Outputs OS specific modules based on tree manipulations and output assists
/* Lopper RTOS header generation */

#define cpus = "236"
#define cpus_cpu0 = "464"
#define cpus_cpu1 = "540"
#define cpu_opp_table = "620"
#define cpu_opp_table_opp00 = "700"
#define cpu_opp_table_opp01 = "768"
#define cpu_opp_table_opp02 = "836"
#define cpu_opp_table_opp03 = "904"
#define dcc = "976"

...
How Do I Engage with System Device Trees?

- System Device Tree project is part of the Linaro Device Tree Evolution Project
- Driven under the OpenAMP umbrella
  - Includes silicon vendors, OS vendors and others
  - Openampproject.org
- Join the mailing list
  - https://lists.openampproject.org/mailman/listinfo/system-dt
- Be part of the regular discussions
Thank you

Accelerating deployment in the Arm Ecosystem