Bus scaling QoS update

Georgi Djakov
Agenda

- Background
- The problem
- The solution
- Next steps
Background

- **Challenges of SoC architecture**
  - More and more features (IP cores)
  - Many components talking to each other
  - Multiple sources of traffic
  - Concurrent transfers
  - Predictability

- **Evolution of on-chip interconnects**
  - Buses, crossbars, network-on-chip

- **Benefits**
  - Scalability - packet communication
  - Power efficiency - shorter wires
An example NoC topology
The problem

- On-chip interconnect buses can handle high throughput data transfers, but most of the time they may be idle.
- Simultaneous data flows across the SoC with different sources and destinations, interleaved traffic.
- Interconnect buses can be configured according to the use-case and demand.
- Each SoC vendor has its own custom implementation in the downstream kernel.
- Need a common solution in the upstream Linux kernel.
Requirements

- Description of the topology.
- Expose path between endpoints as resources. The path is claimed by consumer drivers and they set constraints on it.
- A common framework for that can traverse the topology and allow a consumer driver to get() a path.
- Track and aggregate requests received from drivers.
- Update constraints and set hardware to most optimal configuration.
- The vendor specific interconnect controller drivers register the topology with the framework and implements low-level operations.
The solution

- A new framework - interconnect framework
- Vendor specific low-level drivers that control the hardware
- Consumer API to set constraints from drivers
Device-Tree binding

- Interconnect provider

```

snoc: snoc@0580000 {
    compatible = "qcom,msm-bus-snoc";
    #interconnect-cells = <1>;
    interconnect-port = <&bimc MAS_SNOC_CFG>,
    <&bimc SNOC_BIMC_0_MAS>,
    <&bimc SNOC_BIMC_1_MAS>,
    <&pnoc SNOC_PNOC_SLV>;
}
```


Device-Tree binding

- Interconnect consumer

```
sdhc_1: sdhci@07824000 {
    ...
    interconnect-port = <&pnoc MAS_PNOC_SDCC_1>;
    interconnect-path = <&mem>, <&usb_otg>;
    interconnect-path-names = "mem", "usb";
```
Interconnect API

- struct interconnect_path *interconnect_get(struct device *dev, const char *id);
- int interconnect_set(struct interconnect_path *path, u32 bandwidth);
- void interconnect_put(struct interconnect_path *path);
Interconnect API

- Currently works only with device-tree and platform drivers
- Use device-tree to denote path endpoints
- Types of constraints
- PM QoS
The first vendor specific driver

- Dragonboard 410c as a test platform.
- Test scenarios
  - Data transfers between peripherals
  - Data transfers from/to memory
Future work

- Feedback
  - LKML
  - Should we use PM QoS?
  - Remove dependency on DT
  - Support for ACPI?
  - Use integers instead of string ids for the binding

- Add support for other types of QoS constraints
  - Average bandwidth
  - Peak bandwidth
  - Round-trip latency
  - Priority

- Extend the Qualcomm msm8916 interconnect driver
  - Include the full topology

- Practical and easy example for testing
Thank You

#BUD17

For further information: www.linaro.org
BUD17 keynotes and videos on: connect.linaro.org