* Without an SMMU, devices use physical addresses directly, which can:
  + Cause security vulnerabilities (unauthorized memory access)
  + Create conflicts when multiple devices access shared memory
  + Lack flexibility in address translation

Solution with an SMMU:

* Address Translation: Converts device-generated I/O virtual addresses (IOVAs) into physical addresses (PAs).
* Isolation: Ensures that a faulty/malicious device cannot corrupt another device’s memory.
* Access Control: Enforces permissions (read/write restrictions) for each device.

How Does the SMMU Work?

1. A device generates an IOVA (I/O Virtual Address) when trying to access memory.
2. The SMMU translates the IOVA into a Physical Address (PA) using a page table.
3. The translated PA is then used to access the actual memory in RAM.
4. If the device does not have proper access rights, the SMMU blocks the request and triggers a fault.

Key Components of SMMU

1. Translation Table

* Holds mappings from IOVA → PA.
* Similar to MMU page tables used by a CPU.
* Stored in RAM and managed by the OS/driver and pointer to that is held by TTBR register in case of MMU, though it is a discussion for SMMU. It changes based on process.

2. Context Bank

* Stores translation configurations for each device.
* Each device gets its own "virtualized view" of memory.

3. Translation Buffer (TLB)

* A cache that speeds up address translation.

4. StreamID and ASID

* StreamID: Identifies which device is making a memory request.
* ASID (Address Space Identifier): Identifies memory context per process.

4. Address Translation in SMMU

Two Types of Translations

1. 1-Stage Translation:
   * Used when the device's virtual addresses directly map to physical addresses.
   * Common in embedded systems where memory management is simpler.
2. 2-Stage Translation (Nested Translation):
   * First stage translates IOVA → Intermediate PA (IPA).
   * Second stage translates IPA → Final PA.
   * Used in virtualized environments where a guest OS sees only an intermediate memory layout.

6. SMMU in a System-on-Chip (SoC)

In an ARM-based SoC (e.g., Qualcomm Snapdragon, NVIDIA Tegra, Apple M-series):

1. The CPU communicates with the SMMU to configure mappings.
2. PCIe and DMA devices use SMMU for secure memory access.
3. GPUs and network devices use SMMU to prevent unauthorized access.

7. Example: SMMU in Linux Kernel

In the Linux kernel, SMMU support is provided by the IOMMU framework.

* Device Tree Entry (ARM-based SoC example):

smmu: iommu@50000000 {

compatible = "arm,smmu-v3";

reg = <0x50000000 0x10000>;

};

* Kernel Driver Interface:

struct iommu\_domain \*domain;

domain = iommu\_domain\_alloc(&platform\_bus\_type);

iommu\_attach\_device(domain, dev);

9. Debugging SMMU Issues

* Check if the device is assigned an IOMMU domain:

cat /sys/class/iommu/\*/devices

* Check translation faults:

dmesg | grep iommu

* Enable IOMMU in the kernel boot parameters:

iommu=on iommu.passthrough=0

Identify the SMMU Version

dmesg | grep smmu

Example output:

[ 1.234567] msm\_iommu: SMMU v2 enabled

or

[ 1.234567] arm-smmu-v3 5040000.iommu: SMMU v3.2 initialized

2. Device Tree Configuration for Qualcomm SMMU

smmu: iommu@5040000 {

compatible = "qcom,msm-iommu-v2";

reg = <0x5040000 0x10000>;

#iommu-cells = <1>;

qcom,stall = <1>;

qcom,bypass-allow = <1>;

};

* qcom,stall = <1>; → Enables fault handling by stalling the device instead of terminating transactions.
* qcom,bypass-allow = <1>; → Allows devices to bypass the SMMU when not explicitly assigned.

Example 2: SMMU-v3 Configuration (For Newer Qualcomm SoCs)

smmu: iommu@5040000 {

compatible = "arm,smmu-v3";

reg = <0x5040000 0x10000>;

stream-match-mask = <0x7f>;

qcom,stall = <1>;

qcom,bypass-allow = <1>;

};

* stream-match-mask = <0x7f>; → Helps match incoming StreamIDs to appropriate translation tables.

3. Enabling SMMU in Linux Kernel (Qualcomm Specific): If SMMU is disabled, you need to enable it in the kernel command line. Modify Kernel Boot Parameters

iommu.passthrough=0 iommu.strict=1

* iommu.passthrough=0 → Ensures that all devices go through SMMU.
* iommu.strict=1 → Enables strict memory protection.

If building your own kernel, enable these options in menuconfig:

Device Drivers -> IOMMU Hardware Support -> ARM Ltd. System MMU (SMMU) Support

Device Drivers -> IOMMU Hardware Support -> ARM Ltd. System MMU (SMMU) v3 Support

For Qualcomm-specific SMMU:

CONFIG\_ARM\_SMMU=y

CONFIG\_ARM\_SMMU\_V3=y

CONFIG\_QCOM\_IOMMU=y

4. Attaching a Device to SMMU

Step 1: Check if the Device Uses SMMU

ls /sys/class/iommu/

If the device is using SMMU, you’ll see entries like:

iommu0 iommu1

Step 2: Check Devices Attached to SMMU

cat /sys/class/iommu/\*/devices

Output:

0000:01:00.0 (This is a PCIe device using SMMU)

Step 3: Attach a Device to an IOMMU Domain. If a device is not using the SMMU, manually attach it:

echo "0000:01:00.0" > /sys/class/iommu/iommu0/devices

5. Debugging SMMU Issues on Qualcomm Hardware. If SMMU blocks a memory transaction, you’ll see faults in dmesg:

dmesg | grep iommu

Example:

[ 3.456789] arm-smmu 5040000.iommu: Unhandled translation fault at IOVA 0x00000000ff000000

Analyze Faults

* Translation Faults → Check if the device has the correct mappings.
* Access Permissions Faults → Verify memory access rights.
* StreamID Issues → Ensure the device is assigned the correct StreamID.

Enable Debugging Logs for SMMU

Modify kernel parameters:

echo 'file arm-smmu.c +p' > /sys/kernel/debug/dynamic\_debug/control

This enables debug logs for ARM SMMU driver.

6. Performance Optimizations for Qualcomm SMMU

1. Enable Page-Table Caching

Add the following to the kernel parameters:

iommu.passthrough=0 iommu.strict=0

* iommu.strict=0 → Allows SMMU to cache translations, improving performance.

2. Increase SMMU TLB Entries

Modify the SMMU TLB settings in the device tree:

smmu {

arm,tlb-flush-interval = <1024>;

};

3. Use DMA-Coherent Buffers

Ensure your DMA buffers are properly allocated:

dma\_alloc\_coherent(dev, size, &dma\_handle, GFP\_KERNEL);

7. Summary

* Qualcomm SoCs use SMMU-v2 or SMMU-v3 to provide secure memory access.
* Device tree (.dts) is used to configure SMMU settings.
* Linux kernel must enable ARM SMMU support (CONFIG\_ARM\_SMMU=y).
* Check IOMMU domains using /sys/class/iommu/.
* Attach devices to IOMMU manually if needed.
* Debug SMMU faults using dmesg.
* Optimize performance using page-table caching and TLB settings.