Hypervisor-less virtio

Assembling Multi-OS systems using standards-based protocols for intra-SoC connectivity and device sharing

Dan Milea, Technology Office, Wind River
dan.milea@windriver.com

The quantity of software content in intelligent systems has been growing in size and complexity, enabled courtesy of Moore's Law. For hardware that is designed for embedded systems, the monotonically increasing numbers of transistors available to embedded systems are being consumed by additional processing cores, caches, RAM, flash and IO. All of which can now all fit into a single complex multicore chip.

Modern application-specific Systems-on-a-Chip (SoCs) are realized faster than in the past when discrete multicore processors were more general-purpose. This integration of silicon building blocks has reached the point where one OS instance running across all the cores with appropriate drivers for all the included hardware is either simply not possible, not available, or does not scale in a way that satisfies the real-time, concurrency or safety requirements of the embedded applications that it must host.

Allowing subsystems to have their own OS instance reduces testing since only changed subsystems need to be unit tested while the rest of the system can be validated with system-level integration tests and via stability monitoring. Partitioning applications and services with their own OS not only decouples subsystem interdependence, interference, and lifecycle, it is aligned with the architectural best practice of element separation.

Additionally, such an architecture acknowledges that subsystems and applications may come from multiple providers and may need to be partitioned for reasons such as resource provisioning, fault containment, diagnosis, support, licensing or security reasons.

The distribution of resources in modern SoCs often mirrors the CPU core architecture distribution (e.g., big.LITTLE), with resources like accelerators, FPGAs, GPUs, and DSPs assigned preferentially to the big cores. There are scenarios where the operating system running, for example, on an Arm R5 CPU cluster may need to access resources available on a system partition running Linux.

A couple of years ago I was investigating communications frameworks suitable for IPC in a single physical machine heterogeneous OS solution involving Linux and a real-time operating system. One of the available transports was shared memory, so when I discovered OpenAMP (Open Asymmetric Multi-Processing) and analyzed its capabilities I realized this could be the solution I was looking for.

Since there are plenty of examples available, a skeleton communication infrastructure using shared memory and the OpenAMP framework was not extremely complicated to put in place.

OpenAMP uses a messaging protocol called RPMsg as the basis for inter-core communication. RPMsg requires only two basic hardware components - shared memory, that is accessible by both communicating sides, and inter-core interrupts (see Figure 1). Virtio provides the transport abstraction, while RPMsg defines the communication channels and the endpoints which provide logical connections on top of a RPMsg channel. (See Figure 2)

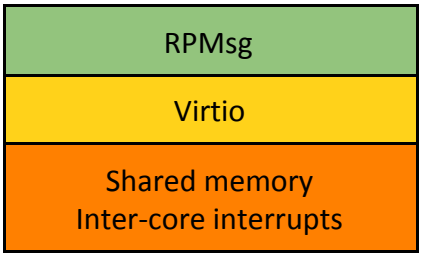


Fig. 1 RPMsg Protocol Layers

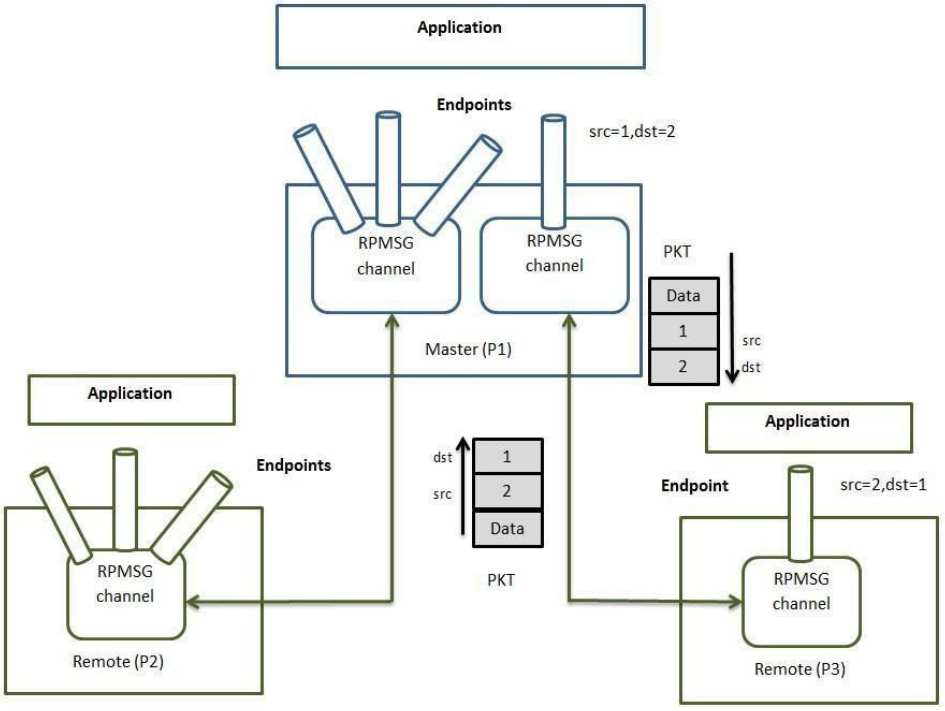


Fig. 2 RPMsg Channels and Endpoints

OpenAMP includes a proxy application which runs on the master processor and handles printf(), scanf(), open(), close(), read(), and write() calls from remote contexts. While this is a good start for resource sharing between the master and the remote, applications often require richer APIs and infrastructure to support their end goals.

Wind River has software IP components which can reduce the effort required for building a complete resource sharing solution on top of the OpenAMP proxy application. However, this implied creating a complete RPC mechanism on top of the RPMsg protocol.

What did this achieve?

An application running on the master processor was used to access the console of the OS running on the remote core. The remote core had gained complete access to the master's file system, thus removing the need for supporting multiple file systems in the RTOS running on the remote core. Since the RPC framework was extensible, services were added for remote TCF debugging of the remote OS via a proxy application running on the master or for allowing the remote application to access OpenCL devices from the master partition.

Although the solution was functional, the implementation had started to deviate from the initial goals of IP reuse and of being able to open-source the software for the benefit of the OpenAMP community.

With this lesson learned, we went back to the drawing board.

All this time virtio kept showing up on our radar.

The virtio transport abstraction was originally developed at IBM for para-virtualization of Linux-based guests. It now serves as a standardized interface that hypervisors provide for virtual hardware presented to guest operating systems.

Why is para-virtualization good?

In a standard hypervisor deployment, virtio can enable guests to achieve high performance by eliminating some of the complexity and latency associated with interruptions in guest execution called VM-exits that happen due to device emulation required by full virtualization.

When a guest tries to execute sensitive or privileged instructions, a VM-exit happens, and the CPU exits from guest mode back to the hypervisor which either handles it or relays the event to a VM-specific virtual machine monitor (VMM) for handling after which control returns to the guest.

Virtio has specifications for console, serial, IPC (vsock), file system (9p, virt-fs) and various other device classes. Virtio also specifies multiple underlying transports. In particular, the MMIO transport is interesting for embedded systems since it can be used both to communicate downwards to an embedded hypervisor but also communicate laterally between runtimes deployed on a multicore CPU SoC.

Such lateral virtio has been labelled ‘hypervisor-less” virtio. The goal of the "Hypervisor-less virtio" initiative is to prototype and define a framework for using virtio as a communication infrastructure, while removing the penalties associated with transitions from guest to hypervisor and back. And, in the process, enable virtio to be used for heterogeneous multicore devices which leverage compute islands rather than virtualization for hardware partitioning.

Why virtio?

- Virtio is standardized and designed to allow interoperable and independent implementations.
- There is an established community and a significant possibility of software reuse.
- Virtio enables system integration via expected mechanisms e.g., sockets, console, filesystem sharing.
- Virtio increases development velocity and application portability.
- Virtio provides flexibility to deploy workloads on modern multicore OT and IT hardware, including heterogeneous SoCs.

Here are some useful virtio concepts which will help frame the next paragraphs. **Device** refers to the implementation of the virtual/para-virtual device, also known as Backend or Server. **Driver** refers to the guest driver, also known as Frontend or Client.

At its core, virtio makes a set of assumptions which are typically fulfilled by a virtual machine monitor which has complete access to the guest memory space. Removing the hypervisor from the virtio equation would imply the following:

- Feature negotiation is replaced by predefined feature lists defined by the system architect.
- Virtio can be used without virtualization hardware or virtualization enabled.
- The term VMM is a misnomer when there is no virtualization so PMM (Physical Machine Monitor) is used instead.
- The pre-shared memory region is defined / allocated by the PMM.

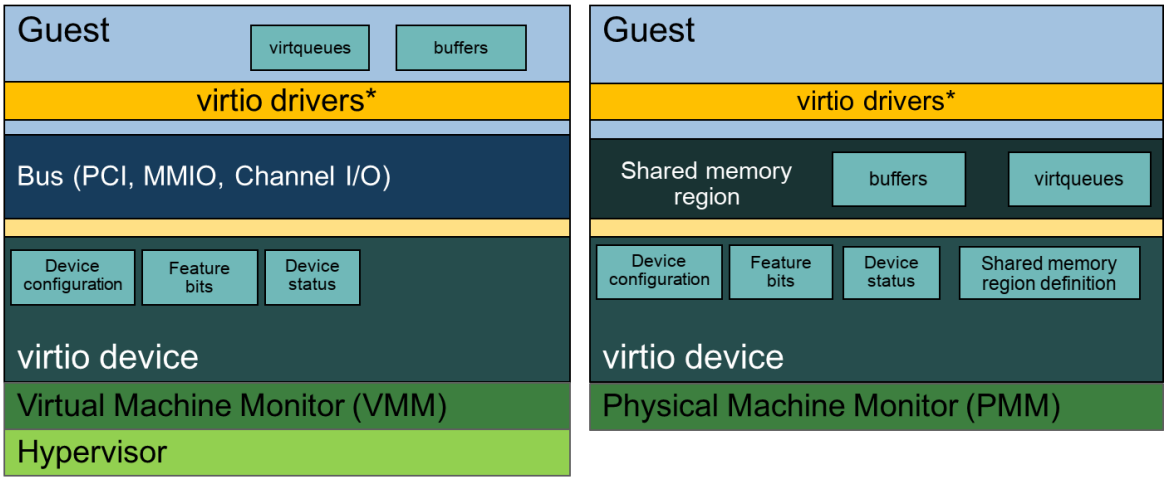
Note: In virtio, the guest driver decides where memory is allocated and is responsible for virtqueue and data buffer allocation.

What makes virtio hypervisor-less?

- MMIO transport over shared memory
- Unsupervised AMP support
- Static configuration (features, queues)
- Hardware notifications
- A PMM instead of VMM (see Figure 3)

It is worth mentioning that performance is highly dependent on the type and number of hardware notifications available on the platform, since the hypervisor-based notification infrastructure will no longer be available.

A research activity started by Intel, which aims at introducing MSI-based notifications for virtio MMIO, is of particular interest. Additional information is available here: https://github.com/OpenAMP/open-amp/wiki/OpenAMP-Application-Services-Subgroup-Meeting-Notes-20201027



*Drivers: file system (9p), console (serial), IPC (vsock), network (virtual ethernet)

Fig. 3 Standard vs Hypervisor-less virtio

Hypervisor-less virtio use-cases

While not meant to be an exhaustive list, the following scenarios are a good fit for deploying hypervisor-less virtio.

- OpenAMP, in the application development framework for high level services.
- Multiple virtio back-ends which can be implemented as Linux user-space processes serving different guests.
- Real-time or safety applications with a path to certification which require access to resources available in a rich OS environment.

Let us take OpenAMP as an example.

OpenAMP is a framework providing the software components needed to enable the development of software applications for AMP systems. It allows operating systems to interact within a broad range of complex homogeneous and heterogeneous architectures and allows asymmetric multiprocessing applications to leverage parallelism offered by the multicore configuration.

OpenAMP is a perfect candidate for hypervisor-less virtio deployment, since most OpenAMP configurations use shared memory to communicate between heterogeneous CPU clusters.

The transport primitives are already based on virtqueues with the RPMsg protocol on top providing channel-based communication between RPMsg named endpoints. Above the RPMsg layer resides vendor specific implementations for serial communications, remote audio processing, etc.

OpenAMP includes a limited form of remote system calls (open, read, write, close) implemented via a proxy application running on the master and which provides this service to the remote node.

Standardizing on virtio as the communication infrastructure between the master and the remote node would automatically enable a wide range of APIs, from console and IPC via vsock to file systems and access to hardware accelerators available on the master.

Technology

In the case of real time operating systems which have complete control over the memory mapping, the virtio drivers can be modified to place the virtqueues and the data buffers in the pre-shared memory region, not unlike the current implementation of virtqueues in OpenAMP.

On systems with less constraints, a bounce buffer shim located in the virtio framework would allow complete reuse of the virtio drivers by intercepting the enqueue / dequeue operations, making the virtqueue and buffer allocations in shared memory transparent to the drivers.

Linux KVM tool (https://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool.git/) is a convenient starting point to build a PMM as it allows reuse of the virtio device support it already has. The information on the location and size of the shared memory allocated to each virtio device is published in the virtio device header as a shared memory region definition (found in virtio 1.2).

Virtqueue configuration and supported feature bits are statically defined in the virtio device header which is populated in the shared memory region dedicated to a virtio device.

On the guest side, the virtqueues are explicitly located in the shared memory region mapped by the guest. Data buffers are either allocated in the pre-shared memory region or are copied there by a transparent shim layer which implements a bounce-buffer mechanism.

Hypervisor-less virtio can complement and enhance the OpenAMP RPMsg framework by providing communication and resource sharing between a rich execution environment like Linux, running the PMM software and acting as the OpenAMP master, and an application specific real-time or safety island deployed on the remote processor.

By eliminating the hypervisor, a “lateral” communication framework like hypervisor-less virtio enables role assignments (similar to OpenAMP master and remote) based on application requirements and the ability of each of the runtimes to support the PMM.

If you’d like to know more, visit the OpenAMP Project web site www.openampproject.org. The mailing lists for the OpenAMP Project working groups are available at https://lists.openampproject.org/mailman/listinfo. You can subscribe to the mailing lists to get the up to date information.

Glossary of terms

- AMP - Asynchronous Multi-Processing
- RPMsg: Remote Processor Messaging
- TCF: The Target Communication Framework is a lightweight, extensible network protocol from The Eclipse Foundation for embedded system development.
- Virtqueue: A virtqueue is simply a queue into which buffers are posted by the guest for consumption by the host. Each buffer is a scatter-gather array consisting of readable and writable parts and the structure of the data is dependent on the device type.
- VMM: Virtual Machine Monitor
- PMM: Physical Machine Monitor
- KVM: Kernel-based Virtual Machine
- VM-exit: A transition from guest mode to VMM which happens when the guest tries to execute “sensitive” instructions. VM-exits signal the transfer of control to the hypervisor, enabling it to execute the sensitive instruction on behalf of the guest system.

References

- Koning, M. (2021). Embedded Systems Go Mainstream. Embedded World Exhibition and Conference.
- Virtual I/O Device (VIRTIO) Version 1.1, Committee Specification 01 (11 April 2019). https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html
- Authoritative source of the VIRTIO (Virtual I/O) Specification document. https://github.com/oasis-tcs/virtio-spec
- OpenAMP Project. https://www.openampproject.org
- RPMsg Messaging Protocol. https://github.com/OpenAMP/open-amp/wiki/RPMsg-Messaging-Protocol
- OpenAMP RPMsg Virtio Implementation. https://github.com/OpenAMP/open-amp/wiki/OpenAMP-RPMsg-Virtio-Implementation
- LKVM (Linux KVM tool). https://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool.git/