Managing co-processors for Linux PV domains by running a Xen hypervisor on ARM platforms

Can ARM SoCs perform high-load tasks?

As modern ARM SoCs become faster and faster, they are now capable of performing the same high-load tasks that desktop PCs were performing a few years ago, such as HD video playback and high-speed graphic rendering. The structure of an ARM SoC is also now quite complicated. In addition to containing a CPU module, it also includes several peripheral modules (e.g., UARTs, Wireless, HDMI ports, etc.) and co-processors that are designed to help with high-load tasks. For example, a standalone Graphic Processor Unit (GPU) and Video Processor Unit (VPU) are assembled together with the main CPU on almost all modern ARM SoCs that are designed for mobile and automotive markets.

What is a co-processor?

A co-processor is a standalone processor designed to perform a single type of task (e.g., mathematical calculations, audio or video encoding / decoding, etc) in order to avoid overloading the main CPU. From the main CPU's point-of-view, a co-processor is a blackbox with a well-defined connection interface. Although the main CPU can utilize the co-processor, it is not familiar with the module's internal components.

How does a co-processor work without virtualization?

It is quite simple. The CPU configures its co-processors, powers them up, loads proper firmware, configures their MMUs, establishes a communication channel, and does everything necessary to make the co-processors work as standalone processors. While performing high-load tasks, the main CPU "kicks" a co-processor by passing specific data that both components understand through a previously established communication channel. This data essentially signals a co-processor to begin performing a specific task. Typically, such high-load tasks start inside userspace libraries that are dedicated to graphic rendering (e.g., OpenGL / OpenEGL) or video playback (e.g., OpenMAX). Co-processors serve as the endpoints for this data flow. Figure 1 shows an example of this data flow for a video decoding use case and a VPU co-processor.

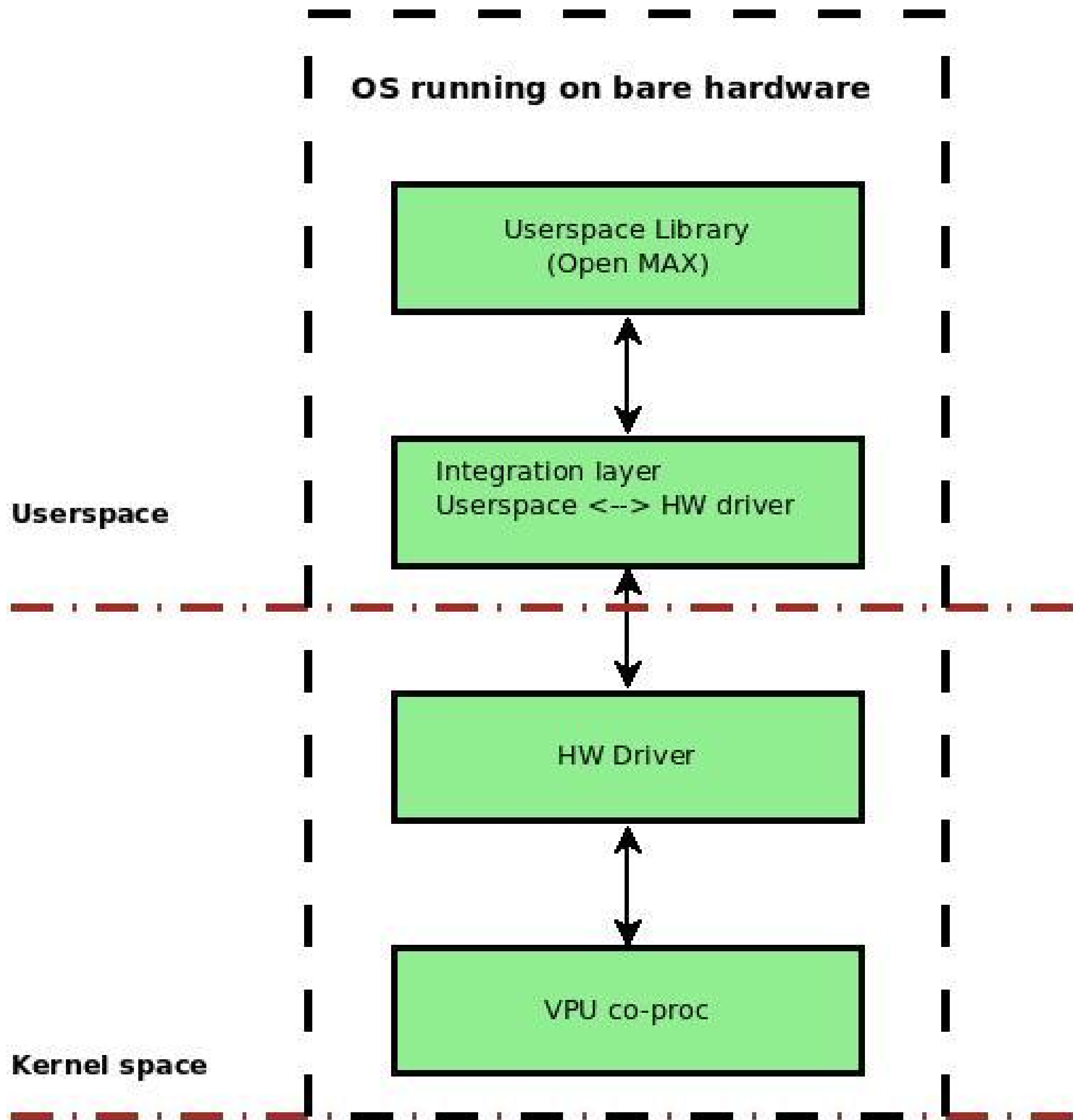

Figure 1. VPU usage in OS running on bare hardware

What is wrong with using virtualization?

Nothing! But if you decide to use an OS that was previously running on bare hardware as the paravirtual (PV) domain, then a co-processor will not work. Below are the primary reasons why co-processors cannot work in a virtualized environment:

1. **MMU management.** PV domains don’t have access to real physical addresses in physical memory. At the same time, co-processors themselves need a fully functional MMU to work properly. As such, a configuration of a co-processor’s MMU will not succeed.

2. **DMA operations.** Similarly, without access to real physical addresses, DMA operations are not possible. (This issue occurs only for co-processors, which use DMA.)

3. **Power management.** Because PV domains don't have access to platform hardware, operations such as powering up, frequency scaling, and reseting are not available.

Can co-processors be made to work in a PV environment?

Yes, but it requires some work. Taking into account that co-processors are leveraged by userspace libraries (even if they don't know about it), we may perform the following steps within a PV environment:

1. Move a co-processor's hardware and its driver to domain 0. This step solves power management and MMU / DMA management issues.

2. Introduce a co-processor’s frontend driver into a PV domain and the backend driver into domain 0. They will be connected using the existing Xen hypervisor frameworks.

3. Connect a backend driver with a native co-processor driver. A new integration layer is needed for this step.

4. Connect a PV frontend driver to a userspace library that is dedicated to the co-processor. A new integration layer is also needed for this step.

Starting at this point, an information bidirectional pass-through channel is established between domains. A wide range of data that a co-processor requires to perform its tasks may be passed between domains. For example, a VPU needs a hardware pointer to buffer in order to perform decoding tasks. This buffer is prepared by a userspace library (typically OpenMAX), passed through the PV frontend–backend connection to the existing hardware driver, and then passed on to a co-processor. As soon as the VPU finishes decoding, it sends a status reply to the userspace library in the opposite direction. Figure 2 describes this process, with the existing modules marked in green and the new modules marked in blue.

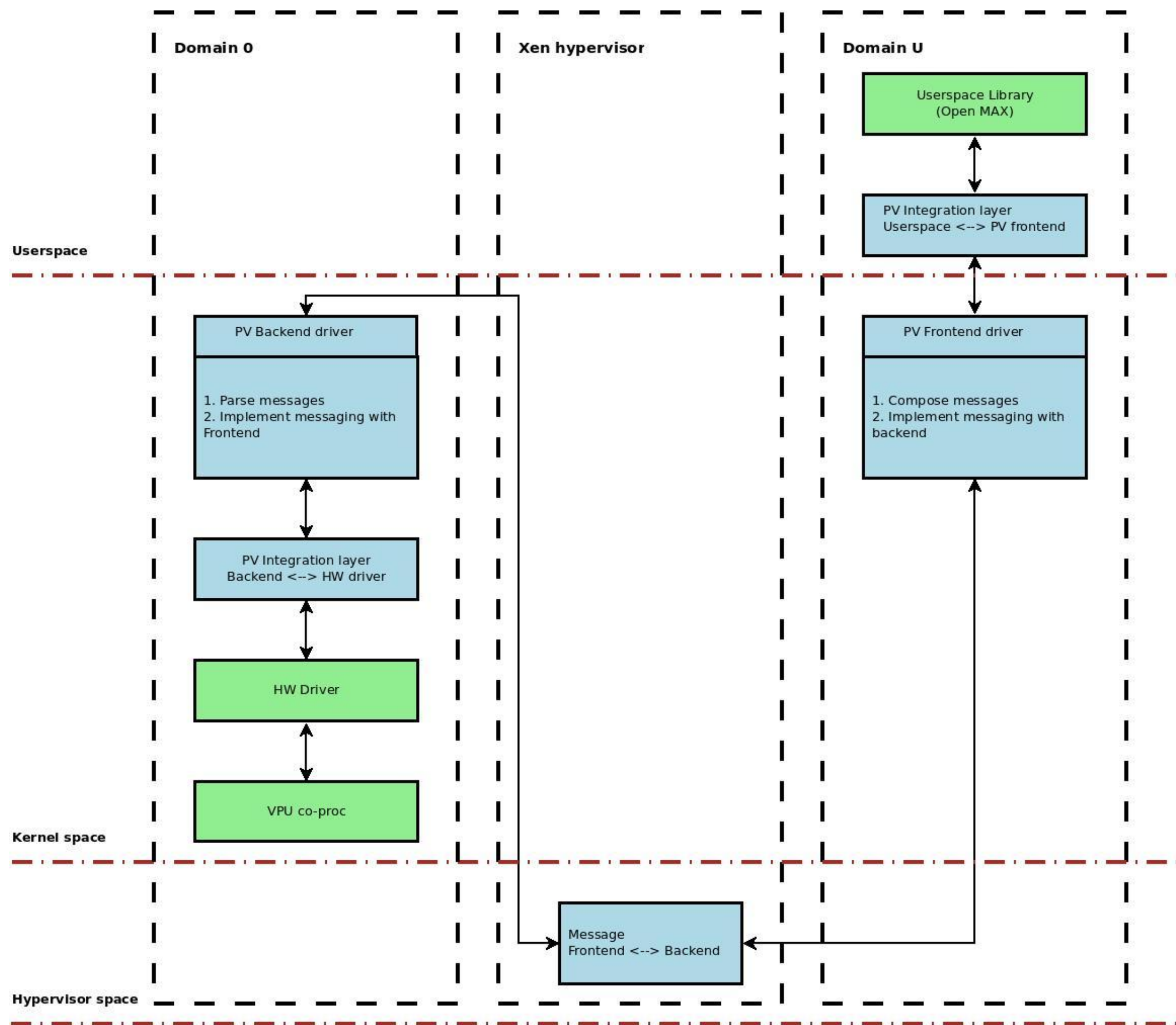

Figure 2. VPU usage under Xen virtualization

At first glance, this solution seems quite simple, but the devil is in the details. Let's take a look at each module one-by-one:

1. **Userspace library.** The userspace library, which is a top element in a system that uses a co-processor, operates with high-level APIs. In this scenario, let's assume it calls an API to “decode buffer.” Without virtualization, this call would trigger a kernel ioctl call. Under virtualization, calling such an API should be caught in the PV frontend driver. In other words, it is necessary to introduce a new integration layer between a userspace library and a kernel space.

2. **Integration layer for userspace library.** In this layer, it is necessary to re-implement / wrap all APIs that are used by this library and to make a connection with a frontend

driver. The connection may be implemented using typical file operations (e.g., ioctles, etc.). I would personally use file operations in this scenario.

3. **PV frontend driver.** The PV frontend driver is a module that receives data from the top userspace library and then passes it to the backend in domain 0. Passing data between domains is a simple operation that is already frequently implemented by many PV drivers (e.g., block-device, network, etc.). A connection is implemented using a messaging technique. As such, another function of the PV frontend is composing messages that encapsulate the data retrieved from the userspace. Each userspace API call will generate a message that is then passed to the backend driver.

4. **Backend driver.** The backend driver receives a message from the PV frontend driver, parses it, identifies the API call encapsulated by the message, and prepares a data request for a native hardware driver. The main point here is that some kind of integration layer should be introduced in a similar manner as in the frontend driver since the backend itself is generic. Although the integration layer may be reused across different co-processors, the hardware drivers are different for each type of co-processor.

5. **Integration layer between backend and native hardware driver.** An integration layer is needed to translate the received messages with the encapsulated API calls to the data format required by the hardware driver. Typically, hardware drivers that manage a co-processor provide a high-level API for communication with the co-processor, as well as a message structure that the co-processor understands. In this scenario, we have another level of integration / translation that is similar to what we have in the PV frontend.

6. **Native hardware driver.** The native hardware driver provides a direct connection to the co-processor. Not many changes are required after it provides an API call to deal with a co-processor. The existing API will be called by an integration layer of the backend driver, and the API will retrieve existing notifications from the co-processor.

Existing solutions that are related to co-processor management

The XenGT solution introduces a GPU sharing mechanism, as can be seen here: https://01.org/xen/blogs/srclarkx/2013/graphics-virtualization-xengt

Conclusions

Using co-processors in a virtualized environment in the aforementioned manner provides several benefits:

1. It extends existing PV domain functionality. If a native OS can play HD videos and use high-quality boosted graphics, why can’t the same OS running as a PV domain do it?

2. It offers the possibility to share a co-processor between different domains. Leveraging existing Xen frameworks and the approach described in this post, we may soon see

several PV domains using similar userspace libraries for similar tasks (e.g., OpenMAX for video encoding, OpenGL for graphic rendering, etc.). Several domains could be connected by their frontends to one backend and send their data to one co-processor. Of course, this may require updating a co-processor's existing software.

3. Security is strengthened when all co-proccesor hardware remains in domain 0. The PV domain will not manage functions such as a co-processor’s MMU, powering up and resetting, etc.