MIPS OS Remote Processor Driver

Whitepaper

Filename	:	MIPS OS Remote Processor Driver.Whitepaper.1.0.9.External.docx
Version	:	1.0.9 External Issue
Issue Date	:	31 Aug 2016
Author	:	Imagination Technologies Limited

Contents

List of Figures

1. Introduction

Embedded systems become more complicated year by year, with increasing requirements for high-speed communications and network stacks to address many different functions and services. These needs are increasingly being met by using a Linux kernel but in most cases the system still needs to deliver the kind of real-time response and determinism associated with simpler Real-time Operating Systems (RTOS).

This can be implemented using multiple processors with one running Linux, the other an RTOS, and with communication between them. However, this forces an inflexible partitioning of the system that makes it hard to adapt the product to different market segments and product lines, particularly as price pressure usually dictates that the whole system is on one chip.

There have been several attempts to run Linux under an RTOS so that Linux is always pre-emptible, but the complexity and often proprietary nature of these products has limited their adoption.

This document describes a different approach that allows a modern multi-core, and optionally multi-threaded, MIPS processor to be dynamically configured between the needs of Linux and the demands of an RTOS. This approach is the 'MIPS remote processor driver'.

Consider a system with short-term real-time processing requirements, which also requires a level of general networking, reporting and data processing. For example an environmental monitoring system that reads a series of sensors at a low rate when background levels of a particular gas is measured, then collects data at a much higher rate whenever an increase in gas concentration is detected, and is then uploaded to a server for later analysis. Traditionally a microprocessor would be provisioned, scaled for the highest rate of processing required for real-time data capture, and another processor would be provisioned for the background processing and network operations required to upload and report the data to the server.

The MIPS remote processor driver enables the MIPS CPUs to be flexibly allocated between the Linux general purpose operating system and one or more CPUs running bare metal code or a real-time operating system. This allows greater flexibility is system design.

In the environmental monitoring example, the system designer would need to provision the processing capacity of the system. For some situations a capacity of two VPEs is required for real-time capture of sensor data. Whereas, ordinarily, a capacity of three VPEs is required for Linux based data processing and networking. The system designer would therefore provision a system with three cores, two VPEs per core to meet these peak demands, as shown in Figure 1 – Traditional provisioning of CPUs.

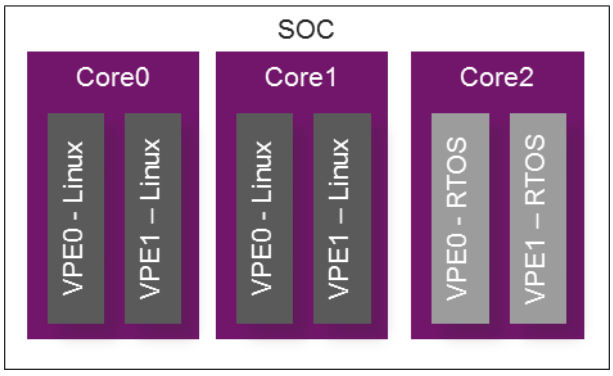

Figure 1 – Traditional provisioning of CPUs

The MIPS remote processor driver allows CPUs to be flexibly allocated between Linux and firmware or RTOS, and therefore the system provisioning can be done differently. Under normal circumstances, the data processing and networking of Linux is allocated three VPEs, and a single VPE can be dedicated to the RTOS managing the sensors. When required, a VPE can be removed from Linux and given an RTOS task instead. Thus the system can instead be provisioned with only two cores, two VPEs per core, as shown in Figure 2 and Figure 3.

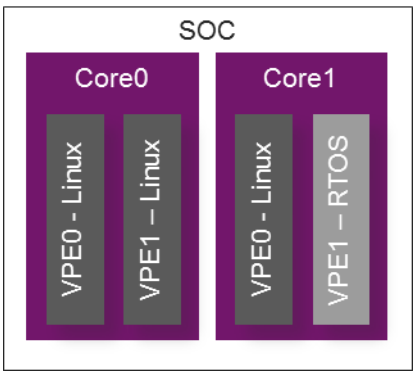

Figure 2 - Dynamic Provisioning – processing / networking phase

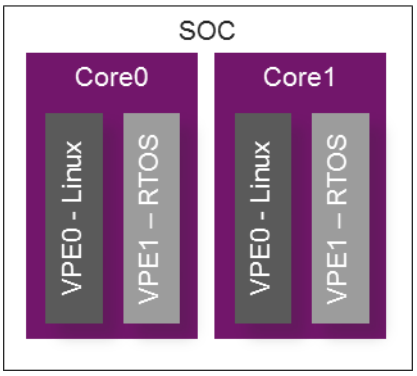

Figure 3 - Dynamic Provisioning – real-time phase

This flexible allocation could reduce the total processing power required, saving the cost and complexity of a system with multiple processors or additional cores, or improving system performance with the ability to increase processing power on demand.

Equally, each of the CPUs may be continuously configured for real-time or Linux use if flexible allocation is not required and the benefits of using a single multi-core multi-threaded microprocessor will still be gained.

1.1. Related Documents

The MIPS® MT Module for the MIPS32® Architecture, MD00378

See Section 7 for other material referenced in this document.

2. Background

A differentiating feature of the MIPS architecture vs other embedded CPU cores is hardware multi-threading (MT) which is included in the MIPS architecture and several MIPS cores including the interAptiv™ and Warrior™ cores. MT can make a single processor core appear to an SMP operating system like two separate cores. The increase in die size is relatively small and there is no need to pay a license fee or royalties for a second core. Furthermore, these two virtual processors can support multiple operating systems or a combination of an OS and a bare metal environment.

Typically, when Linux is booted on a system containing MIPS MT capable cores, it will run across all available virtual processors and cores unless configured to always make certain logical CPUs available for other tasks. This set up is restrictive because it does not allow flexibility in the allocation of CPUs to tasks.

The MIPS remote processor driver allows logical CPUs within the system to be managed dynamically. CPUs may run Linux and be available for scheduling Linux tasks, or they may be removed from Linux and given a separate program to run. That could be bare metal real time code, or even an RTOS. This may be useful in system designs which require real-time control of hardware, or for providing a coprocessor dedicated to media playback or data processing, for example, while having Linux performing general processing on the remaining CPUs.

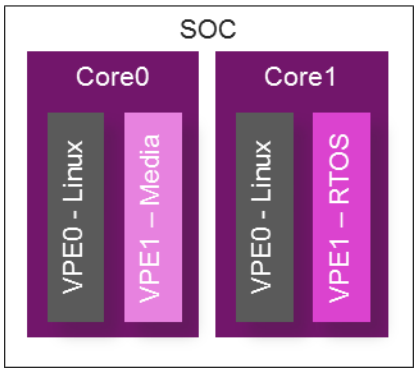

Figure 4 – Real-time co-processors

The remote processor framework within the Linux kernel provides mechanisms facilitating the allocation of memory, resources and communication paths to the remote CPU.

2.1. Remote Processor Framework

The remote processor framework has been in the mainline Linux kernel for several years[1], see Section 7 for list of references. It provides a generic interface for starting and stopping processors, loading firmware, managing memory and allowing standard virtio device based communication. The mainline kernel also provides rpmsg, a generic virtio based messaging framework for communicating with firmware running on the remote processor.

The MIPS remote processor driver implements the remote processor API[2] to allow CPUs that are offline in Linux to be used as a remote processor running separate firmware.

Other remote processor implementations typically use device tree nodes to specify the firmware name that each remote processor should be running. Since the MIPS driver treats one of the generic MIPS CPUs which also runs Linux as the remote processor, instead a sysfs interface is used to allow dynamic management of the firmware running on the remote processor.

2.2. Security

A VPE using the MIPS remote processor driver runs in kernel mode with full access to the system memory. It may be impossible for an RTOS, for example, to function without this privileged access. The firmware and Linux are equally privileged and must trust each other with full access to hardware. Since any firmware running has access to the whole of system memory and is in kernel mode, it effectively runs with Linux root privileges. Loading and interaction with firmware should be subject to the necessary security policy in the target system to prevent exploits.

The system must also be designed to ensure that there are no resource conflicts between firmware and Linux, as both accessing a peripheral concurrently would likely have unpleasant consequences and be difficult to debug.

2.3. MIPS Hardware Multithreading

Hardware multi-threading (MT) is included in the MIPS architecture and several MIPS cores including the interAptiv™ and Warrior™ cores. Pre MIPS architecture Release 6, MT was implemented as VPEs – multiple processing environments sharing the single execution resource of the core. Additionally, VPEs within the same core can only be controlled (started / stopped) by other VPEs in the same core. In the context of the MIPS remote processor driver, this necessitates at least one VPE in each core runs Linux such that it can start / stop and manage any other VPEs in the core running firmware.

In Release 6 of the MIPS architecture, MT is implemented as VPs, which are closer to cores and can be managed centrally by the MIPS Coherence Manager. Therefore with R6 devices there is no restriction to how VPs are allocated between Linux and firmware. This increased flexibility makes the configuration shown in Figure 5 possible, when previously it would be limited as Figure 3.

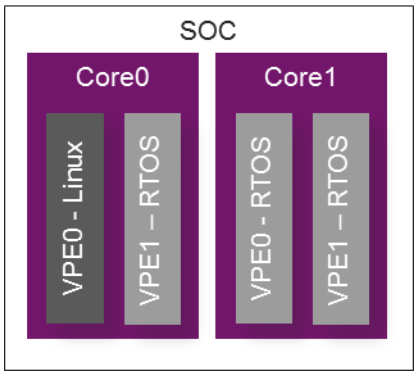

Figure 5 – MIPS R6 VP provisioning of CPUs

2.4. Hardware Support

The MIPS remote processor functionality is supported on MIPS32 devices implementing the MIPS MT ASE for multithreading, such as interAptiv™ and MIPS R6 devices implementing VPs such as Warrior™.

2.5. Limitations

The current remote processor framework in the Linux kernel supports only 32-bit ELF executables, which limits the MIPS remote processor driver to using only 32-bit firmware images. Since the Linux kernel and firmware share pointers (to shared memory) which must be a consistent size, the Linux kernel must also be 32-bit.

3. Using the MIPS Remote Processor Driver

3.1. Control interface - sysfs

The MIPS remote processor interface is designed to allow dynamic management of the system resources. CPUs can be hotplugged from Linux and reassigned to firmware, then stopped and brought back online in Linux once the task is complete.

First a CPU must be made available by hotplugging it from Linux, via the "online" sysfs interface, e.g.

```
# echo 0 > /sys/devices/system/cpu/cpu1/online
```

Once the CPU has been removed from Linux control, the MIPS remote processor driver takes hold of it and creates a sysfs interface for it:

```
# ls /sys/class/mips-rproc/rproc1/
firmware     remoteproc0  stop         subsystem    uevent
```

The firmware file can be written with the name of a firmware file, which should be located in /lib/firmware. When this sysfs file is written, the firmware will be loaded and appropriate virtio devices created for it as described by the firmware image. The CPU will then begin executing the firmware.

3.2. Communication via virtio devices

The remote processor framework supports the creation of virtio devices for communication with firmware running on the remote CPU. These are the same types of devices that are used by virtualisation software to create pseudo devices implemented in software. For example, a firmware may provide a virtual serial port, virtual Ethernet adaptor, etc. which is implemented by the firmware and which will be instantiated as a device on the Linux side. In the case of a virtual serial port, a device node such as /dev/vport0p0 is created to represent the virtual device implemented in firmware.

3.3. Memory layout

When a firmware image is linked to run on a MIPS remote processor, it is statically linked to a particular address. If that region were in one of the unmapped regions of the MIPS address space (KSEG0/KSEG1), then that area would have to be reserved for the firmware image and Linux prevented from using it. If multiple remote processors were to be used, each one would have to have a separate reservation in the virtual address space. This would be difficult to guarantee and unwieldy to manage. For that reason, the MIPS remote processor implementation is designed to allocate the remote processor memory in the mapped region (KUSEG). This means that the firmware can have constant addresses even though the physical memory allocated for it by Linux will not be.

Before the firmware is booted, Linux will create wired TLB entries covering all of this memory to ensure that the firmware has access to it without any TLB refills being necessary.

3.4. Remote processor resource table

The remote processor core code handles a section within the elf image (".resource_table") which has a structure described in [2][3]. Basically this table specifies one or more memory carveouts that the firmware requires, one or more virtio devices provided by the firmware, one or more device memory records and one or more trace buffers.

3.4.1. Carveouts

These records specify a chunk of memory that is made available to the firmware. The format is specified in [3]. If the firmware needs this memory mapped to particular virtual addresses, for example to hold sections of the statically linked firmware, then the address should be set in the da (device address) member. If the firmware just needs to allocate a chunk of memory, then the da member may be set to FW_RSC_ADDR_ANY, in which case Linux will allocate the memory and then put the address in the da member, which the firmware can check once it is running. Memory for these regions is allocated from the contiguous memory pool using the DMA API, so is physically and virtually contiguous.

3.4.2. Virtio Devices

These records specify that the firmware provides a virtio device implementation and that Linux should allocate a driver for it. The format is specified in [3]. The id member should be set to one of the standard virtio device IDs to identify the device type. Dependent on the device, it will require a number of vring structures for passing data between Linux and firmware. These are described here. Linux will fill in the da (device address) member of each vring resource with the physical address of each vring that the kernel has allocated. The vring communication is described in section "Virtio".

3.4.3. Device Memory

These records specify that an IO region of memory, for example a peripheral, be mapped into the remote processor's memory space. The format is specified in [3]. In the case of the MIPS remote processor driver, the remote CPU has access to the full virtual address space anyway, so these records are not currently handled. If the firmware wishes to access a region of IO memory, it may access it directly via a KSEG1 address. Of course, care must be taken to ensure that the Linux kernel is not also accessing that peripheral.

3.4.4. Trace Buffers

These records specify a memory region that the firmware will write to with trace information. The format is specified in [3]. The da (device address) member should be set to the virtual address within the remote processor of the buffer. The kernel will convert that into an offset within one of the memory carveout regions and access that buffer directly. If CONFIG_DEBUGFS is enabled, then a debugfs file will be created for each trace entry. These entries can be read, for example:

```
# cat /sys/kernel/debug/remoteproc/remoteproc0/trace0
```

3.5. Exception / Interrupt Handling

To handle interrupts within the firmware, it must be able to point the exception base address (Coprocessor 0 EBASE register) into the memory region used by the remote processor. This relies on

two features of the MIPS architecture being available in the target CPU. First, the EBASE register must be present (older MIPS CPUs used a fixed EBASE of 0x80000000). Secondly, the WG (write-gate) flag in the EBASE register must be available. Without this flag, the top two bits (MIPS32) of EBASE are read-only and 0b10, to ensure that EBASE is always within KSEG0. When the WG flag is implemented, these top bits may also be written to point EBASE anywhere in virtual memory space - this is essential to be able to handle exceptions and interrupts within firmware running from mapped memory (KUSEG).

If the CPU supports the EBASE register and WG flag, then it should be written with the value of the exception handler routines implemented by the firmware.

3.6. IPIs

Each MIPS remote processor uses two Inter-Processor Interrupts. One is used to interrupt the firmware when a message is available on one of its virtio vrings. The other is used to interrupt Linux when the firmware has made a message available to it. The interrupt numbers are passed as arguments (in registers a1 and a2 respectively). If these two interrupts are provided by the MIPS GIC, they are GIC shared interrupts and their number is the argument minus the number of local GIC interrupts, 7.

3.7. Virtio

The virtio framework is described in [4]. The firmware provides in its resource table a virtio device record and as many vring records as necessary for that device. The firmware fills in the required alignment, number of entries and notifyids of each vring. The Linux kernel will then allocate the necessary structures. Each of the vrings is mapped into the remote processor's address space (KUSEG) using wired TLB entries. The kernel fills in the da field of the vring descriptor in the resource table with the address at which the vring has been mapped.

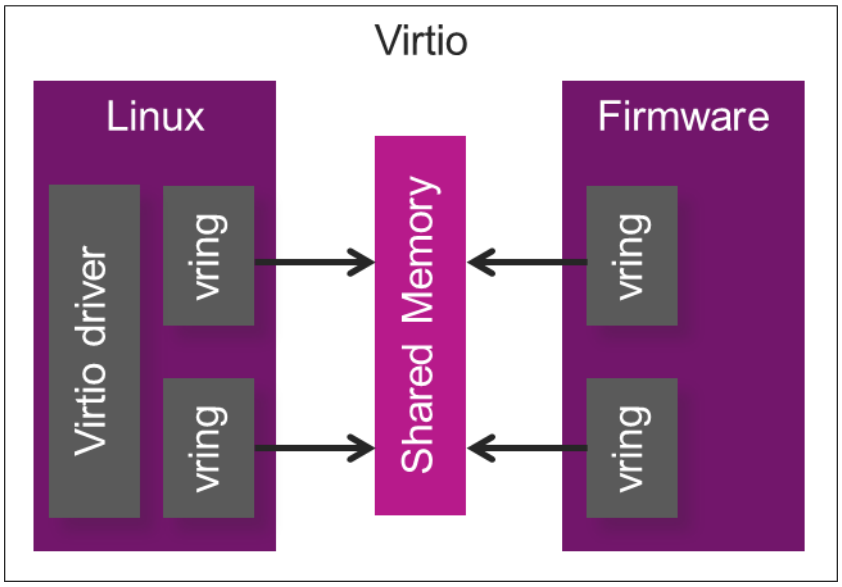

Figure 6 - Virtio

3.7.1. Sending a message to the firmware

When Linux has a message to send to the firmware, it places it in the available ring within the vring. The descriptor at the available index is updated so that its address and length point to the outgoing buffer (the physical address of the buffer). The available index is then incremented. Note that the available index wraps at 65536, not the number of descriptors in the ring. The firmware is then interrupted to handle the buffer.

Once the firmware has handled the buffer of data, it places the descriptor index on the used ring at the used ring index, increments the index and sends an interrupt to Linux. Linux then frees the buffer.

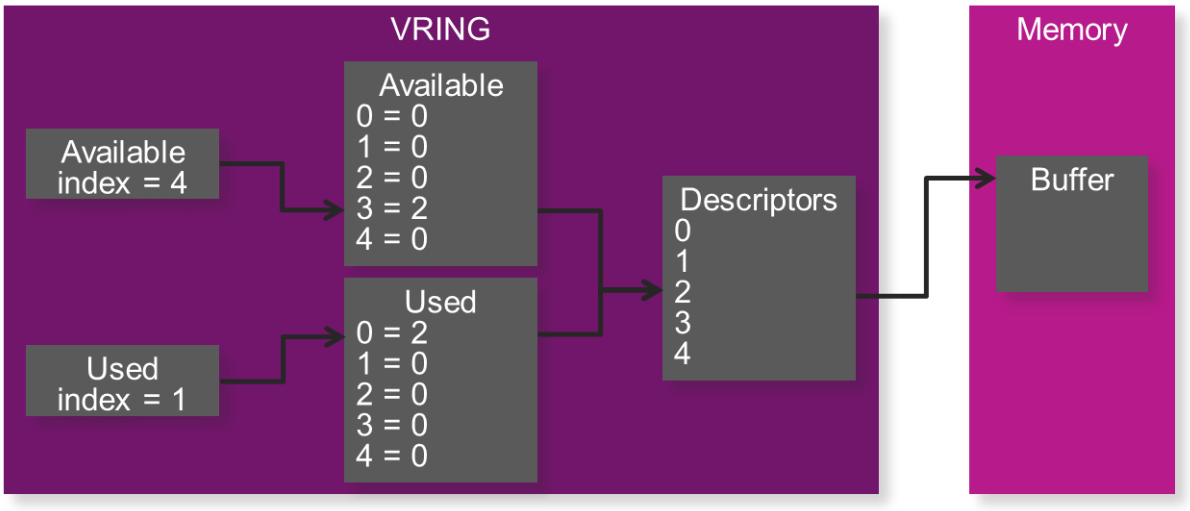

Figure 7 - vrings

Figure 7 shows a vring where Linux has made available a buffer in memory and placed the pointer to that buffer in descriptor index 2. The next index in the available ring was 3, so descriptor ID 2 is placed in the available ring index 3. The available index was then incremented to 4. After the firmware has processed this buffer, it places descriptor index 2 in the used ring at the next available index, which was 0 and then increments the used index to 1.

3.7.2. Sending a message to Linux

When Linux initialises the outgoing vring, it allocates buffers for each descriptor in the available ring. The firmware writes the message into one of the descriptor buffers and writes its index on the used ring at the used ring index. It then increments the index and sends an interrupt to Linux. Linux processes the message in the buffer, frees it, and replaces it with a new buffer.

4. Example

The example MIPS remote proc firmware available at [5] implements a virtio serial port which receives strings; case inverts them, and writes them back. There is also a Linux userspace program which opens the virtual serial port and exchanges messages with the firmware.

The firmware can be configured to either poll for incoming data (which may be preferable in the case of the main loop doing real time processing), or to be interrupted when data is available.

4.1. head.S

Handles the startup of the firmware running on the CPU. It sets the CPUs EBASE register to the value of _exception_vector, a symbol defined in the linker script set to the base of the firmware image (0x10000000). Next it sets the stack pointer to the value of _stack_top, another symbol defined in the linker script after space is reserved for the stack. Finally the bss section is cleared to 0. This uses the _bss_start and _bss_end symbols from the linker script to get the memory range. With all set up complete, it jumps to main().

4.2. main.c

This file contains the main implementation and the resource table. The resource table is placed in the special ELF section ".resource_table", where the remote processor core code will find it. The resource table specifies:

- a carveout region covering the firmware location in memory
- a trace buffer for debug
- a Virtio serial port vdev with two vrings

Within the main() function, first the internal vring structures for the incoming and outgoing rings are initialised using values that Linux has filled in in the resource table.

The interrupts are then configured. This involves finding the address of the GIC from the CM (which first has to be found using the CP0 register CMGGRBase). From this the address of the relevant pending register for the incoming interrupt can be determined. If POLLED_MODE is defined to 0, then here the incoming interrupt will be unmasked. All local interrupts, such as the timer, that Linux may have left unmasked are disabled before enabling global interrupts.

When the incoming interrupt flag is detected, either by polling for it when POLLED_MODE is defined to 1, or in processing the resultant interrupt, the incoming vring is inspected for newly available buffers. Each one found is handed to the handle_buffer() function.

The handle_buffer function gets an available buffer from the outgoing vring and copies the incoming data to it, while case converting ASCII alphabetical characters. The outgoing buffer is then placed in the used ring of the outgoing vring. The incoming buffer is placed in the used ring of the incoming vring. Linux is then signaled by asserting the IRQ flag associated with the firmware to Linux interrupt.

Linux will then free the used buffer that it made available to the firmware, and handle the incoming buffer from the firmware.

4.3. printf.c

A simple printf implementation used with the trace buffer.

4.4. trace.c

The printf implementation is directed to output characters into the trace_buf buffer. This buffer's address is associated with the trace entry in the resource table. If Linux is configured with CONFIG_DEBUGFS, then the remote processor core code will create a debugfs file, which, when read, will read the string contained in this buffer.

4.5. vring.c

This file contains generic functions for dealing with vrings.

4.5.1. vring_init

This function intialises the internal struct vring representation from the values that have been provided to the firmware by Linux in the resource table.

4.5.2. vring_print

A debug function to print the state of a vring.

4.5.3. vring_get_buffer

This function attempts to retrieve a buffer of data from a vring. It inspects the vring_avail structure in memory shared with Linux and compares the index member with a local copy. If Linux has made a buffer available, the index will have been incremented. The available index indicates which descriptor index contains the new data. A pointer to the buffer of data and its length can then be found in the indicated descriptor. The pointer is a pointer into physical memory, so this must be mapped into addressable virtual memory first. The code in handle_buffer gets a KSEG0 address for the buffer to access it without needing a TLB entry for it.

4.5.4. vring_put_buffer

This function marks a buffer of data in a vring as used. It looks for the buffer pointer in one of the descriptors. When it finds the descriptor, it places its index and length in the used ring at the used index. The used index is then incremented.

5. Implementation of the MIPS Remote Processor Driver

This section documents some specifics of the implementation of the MIPS remote processor driver.

5.1. CPS SMP Framework

The MIPS remote processor driver uses some new functions in the CPS SMP framework to steal VPs from Linux. When the CPU is offline from Linux, the mips_cps_steal_cpu_and_execute() function can be used to request that the CPU perform an alternative task. The address of a function to run and a task_struct must be given to this function. The task_struct is used to set the stack pointer and global pointer of the CPU when it is requested to run the function. As such the task structure should be set up with some allocated stack space. The cps_start_secondary function is used to start the CPU running, which is the same function as used to start the Linux kernel running on the CPU at boot.

The mips_cps_halt_and_return_cpu function performs the inverse task and returns the VP to the halted state that it was in while offline to Linux. With the MIPS MT ASE, only sibling VPEs within the same core can write other siblings control registers, so at least one sibling CPU must be kept online to do this. A function is then run on that sibling CPU to halt the VPE. With the new Virtual Processors implemented in the MIPS R6 architecture, any VP can control any other VP via the MIPS CM, so there is no need to keep a sibling online, and the stop function can just write the relevant core's VP_STOP register.

5.2. Inter Processor Interrupts

Linux uses two IPIs for each CPU in the system, one for calling a function on the other CPU and one to request that the other CPU schedule a different task. When the CPU is running alternative firmware, these two IPIs are not needed by Linux so are re-purposed for delivering IPIs between the firmware and Linux. The IPIs are released from Linux using mips_smp_ipi_free(). This frees the IPIs from the IPI IRQ domain. The IPIs can then be re-allocated using irq_reserve_ipi().

When the CPU is returned to Linux, the firmware IPIs are released using irq_destroy_ipi() and the Linux IPIs re-allocated using mips_smp_ipi_allocate().

5.3. Memory Carveouts

As described earlier, having the firmware run from unmapped memory creates a management headache if multiple firmwares or multiple VPs are to be used with the same firmware. To avoid this, the MIPS remote processor framework is designed to have the firmware run from mapped memory. Since TLB exceptions would be rather inconvenient in a real time system, the necessary entries are wired into the TLB before the firmware begins executing. Only the CPU which will execute the firmware can write the TLB with the necessary entries, since the MIPS architecture does not allow VPEs to modify other VPE’s TLB entries. There is already kernel infrastructure available to facilitate adding wired entries to the TLB, so that is reused here.

When the MIPS remote processor driver hands a CPU over to firmware, via the mips_cps_steal_cpu_and_execute function, it passes mips_rproc_cpu_entry as the entry point. This is a function within the driver. The function first determines the maximum pagemask that this CPU's TLB supports, so that carveouts can be covered with as few entries as possible using the largest appropriate page mask.

The rproc struct allocated by the remote processor core contains lists of the requested carveouts and virtio devices - these are iterated by mips_rproc_carveouts() and mips_rproc_vdevs() respectively. These both use mips_rproc_fit_page(), which uses the generic add_wired_entry() function to add TLB entries covering the requested ranges. Carveouts are always allocated with a cached CCA. Virtio device vrings, which will be have memory shared between the Linux kernel and the firmware must be a little more careful with the CCA since it must match on both sides. The remote processor core allocates buffers to share using the DMA api (dma_alloc_coherent()) - so when the TLB entries are created the CCA used matches that which the DMA API will use. If there is no MIPS IOCU present in the CPU, such that IO is not coherent, the DMA API will fall back to using uncached memory, and the remote processor must match this, even though in reality the remote CPU is fully coherent. When per-device IO coherency is implemented for MIPS in the upstream kernel, this pattern could be changed to always use cached coherent memory.

6. Conclusions

As we have seen the MIPS remote processor driver provides the framework for a system to flexibly implement a mix of real-time and Linux based processes on a single MIPS based platform. The ability

to mix multiple cores and multiple threads is particularly suited to the hardware multi-threading available in MIPS interAptiv and Warrior cores. The MIPS remote processor driver enables a simplification of system design, and an increase in performance and flexibility of the system. It may be particularly beneficial to systems where the real time application requirements or general data processing and networking requirements are periodic, although it is equally beneficial for systems with a continuous mix of real-time and Linux based operations.

7. References

[1] https://lwn.net/Articles/464391/

[2] https://www.kernel.org/doc/Documentation/remoteproc.txt

[3] Kernel source include/linux/remoteproc.h

[4] http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.html

[5] https://github.com/MIPS/mips-rproc-example