CHAPTER 7

Xeon Phi System Software

Intel Xeon Phi needs support from system software components to operate properly and interoperate with other hardware components in a system. The system software component of the Intel Xeon Phi system, known as the Intel Many Integrated Core (MIC) Platform Software Stack (MPSS), provides this functionality. Unlike other device drivers implemented to support PCIe-based hardware, such as graphics cards, Intel Xeon Phi was designed to support the execution of technical computing applications in the familiar HPC environment through the MPI environment, as well as other offload programming usage models. Because the coprocessor core is based on the traditional Intel P5 processor core, it can execute a complete operating system like any other computer. The disk drive is simulated by a RAM drive and supports an Internet protocol (IP)-based virtual socket to provide networking communication with the host. This design choice allows the coprocessor to appear as a node to the rest of the system and allows a usage model common in the HPC programming environment. The operating system resides on the coprocessor and implements complementary functionalities provided by the driver layer on the host side to achieve its system management goals.

The system software layer is responsible for following functionalities:

- Boots the Intel Xeon Phi card to a usable state. Works with the host to enumerate and configure the card.
- Manages memory address space for configuration, I/O, and memory.
- Implements the device driver to run under the HOST OS to help system software functions such as interrupt handling, power state management, data transfer, communication, and supporting application layer requests through API support.
- The system software layer implements various protocols such as the socket protocol over TCP/IP to support the HPC programming model, which can treat Intel Xeon Phi as an independent cluster node from the system point of view. It provides low-level virtual ethernet and Symmetric Communication Interface (SCIF) APIs to help implement the standard communication protocols.
- The OS (also called coprocessor OS) running on the coprocessor manages the memory, I/O, and the processes of the application and/or application-offloaded functions running on the coprocessor by implementing all the necessary OS functionalities to support massively parallel tasks. The OS is based on an open-source Linux kernel to provide these functionalities. The OS implements the file system sysfs to expose communicate-system states and provide configurability of the OS runtime parameters, such as whether to use automated 2MB pages support in the OS for *transparent huge page* (THP).
- Allows users to develop their own applications and tools for value-added implementation, such as an MPI on top of the supplied system functionalities.
- Provides underlying support layers for user-level application execution such as Intel MPI, OpenCL, OpenMP 4.0, Cilk Plus, and the Intel proprietary shared memory and offload execution model. Some of these execution models were covered in Chapter 2 and others will be covered in Chapter 8 on tools.

System Software Component

The high-level software components for Intel Xeon Phi are depicted in Figure 7-1. The Xeon Phi system software has symmetric architecture. The software components on the card are complemented by an equivalent component on the host side to help abstract communication between the host and the coprocessor.

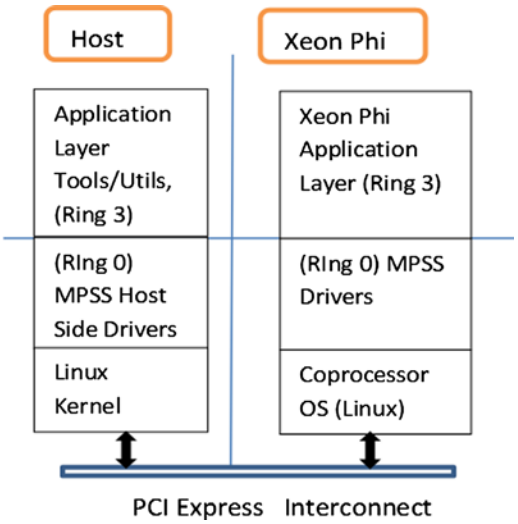

Figure 7-1. *Xeon Phi software layers*

The application layer is built on top of the system software running at Ring 0 (the most protected mode of operation of the processor, where the OS kernel and drivers usually run) on both the host and the coprocessor card. The application layer uses runtime libraries to provide the communication and control necessary to send the code and data to the coprocessor and get the results back. The application layer also contains utilities and libraries that can be used to query the system status and allow socket communications and other communication supports (such as InfiniBand protocols).

The coprocessor tools and utilities are part of the MPSS that allows the platform user to query the device status, as shown in Figure 7-2. Here the application is known as *MIC system management and configuration* (micsmc) and is available with the MPSS. The application runs on the host at the Ring 3 user level and goes through the system drivers to communicate with the corresponding software piece on the coprocessor in order to retrieve such system information as core and memory utilization, core temperature, and power received from the system management component and controller (SMC) hardware on the coprocessor system, as discussed in Chapter 6.

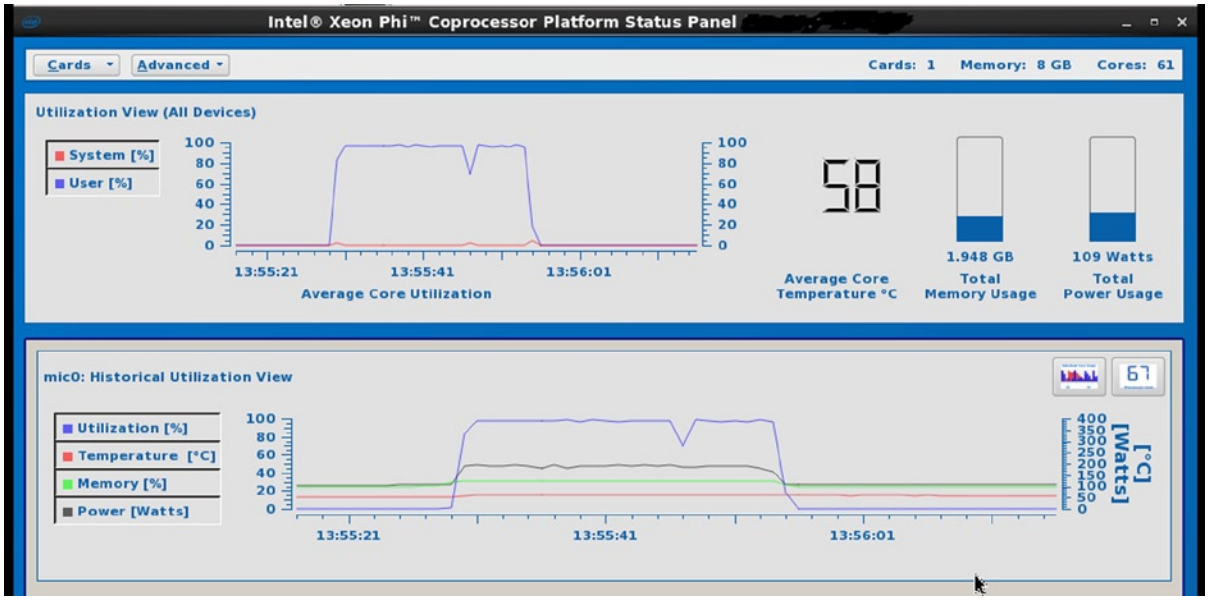

Figure 7-2. *Application layer system utility showing Xeon Phi coprocessor runtime status*

Applications running on the host side can also send computation and data over to the coprocessor via the PCIe bus using the system software layers on the host side. The data and code sent over to the coprocessor side are executed as a separate process (offload_main process) under the coprocessor OS. It is also possible to log in to the coprocessor OS with a remote shell such as the ssh command and run an application natively as one would run it on the host side. The MPSS subsystem also exposes common APIs to the applications running in the Ring 3 layer. These APIs include socket-level APIs over TCP/IP using a virtual ethernet interface and such MPI communication components as the Direct Access Programming Library (DAPL) provided to the fabric (communications hardware) independent API, Open Fabric Enterprise Distribution (OFED) verbs, and the Host Channel Adapter (HCA) library for InfiniBand technology. The coprocessor OS provides a command-line interpreter, such as the host Linux OS, to interact with the user connected directly to the coprocessor OS with tools such as secured shell functionality (ssh).

Ring 0 Driver Layer Components of the MPSS

The Ring 0 layers of the MPSS consist of the following basic components to support application and system functionalities (Figure 7-3):[1]

- *Host-side Linux kernel device driver for the Xeon Phi card*: The primary job of the host-side driver is to initialize the Xeon Phi coprocessor hardware. The driver is responsible for loading the OS in the coprocessor during the coprocessor boot process.
- *Symmetric Communication Interface*: The SCIF layer provides low latency communications between the host and the coprocessor. Other communication abstractions such as an IP-based network depend on the SCIF layer to perform their functions.

[1] *Intel Xeon Phi Coprocessor System Software Developers Guide*: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-software-developers-guide.pdf.

- *Xeon Phi coprocessor OS*: The coprocessor OS is based on the Linux kernel from `kernel.org` modified to fit the specifics of Xeon Phi architecture.
- *Symmetric virtual ethernet drivers*: This software layer provides an IP-based networking communications protocol between the host and the client software. It uses a lower-level SCIF interface.
- Various OFED components to support MPI over the InfiniBand (IB) interface.

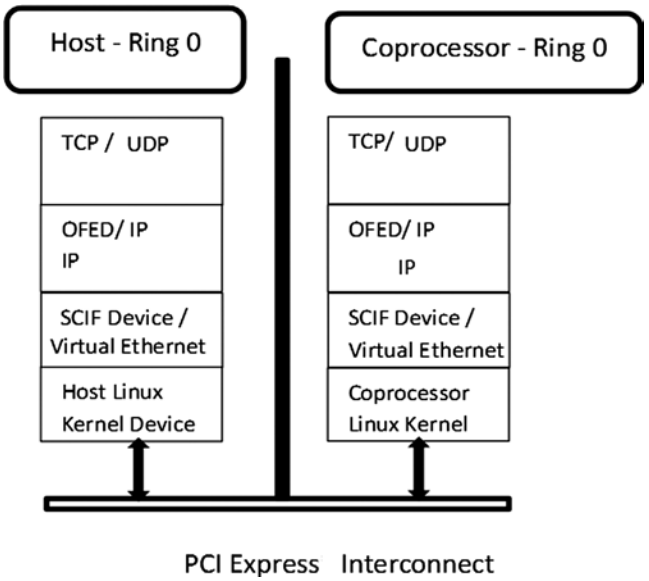

Figure 7-3. *Ring 0 components of the MPSS stack during runtime*

Applications and tools communicate with these components to perform their functionalities. The software layers are available when the system is booted and in the running state. It is possible to boot the coprocessor card through tools provided as part of the MPSS stack without booting the host system itself. The tool can be used to reboot the coprocessor at any time. In a cluster environment, for security purposes, the card may be rebooted to reset all the memory contents, including the RAM drive, to bring the coprocessor to clean reboot state.

System Boot Process

For the system administrator, it may be useful to understand the bootstrap process of the card to debug a system failure. *Bootstrap* is the process of initializing the card and loading the coprocessor OS. The coprocessor card contains firmware (ROM) that helps boot the card when first powered on reset by the host utilities. After the card reset, one of the cores of the coprocessor, known as a *boot strap processor* (BSP), starts execution. The first instruction executed is located in the default location in all x86-family processors: 0xfffffff0. The instruction pointer of the coprocessor points to this memory location after the power reset. This location points to the firmware section known as fboot0. This section of code is trusted code and cannot be reprogrammed, as it serves as the *root of trust*. This section of code authenticates the second stage of the boot process fboot1. If the authentication fails, the power is cut off from the core and ring by the boot strap code in fboot1 ROM area. At this stage, the host can reprogram the fboot1 to recover a bad or compromised fboot1 code block. If the power is cut from the core and ring, the coprocessor goes into zombie state. A jumper on the card then needs to be physically reset for the card to be recovered from this zombie state.

If the fboot1 passes the authentication check, the control is passed over to the fboot1 entry point and continues execution. Fboot1 code initializes the coprocessor cores, memory, and other coprocessor components. It copies code to the GDDR5 memory to improve performance. Then it moves on to boot the rest of the cores. Once all the cores are booted, the cores go into halt state. The coprocessor lets the host know when it is done with initialization. At this point, the host downloads the coprocessor OS image from the hard drive on the host side (Figure 7-4) to the predefined address location on the GDDR5 memory on the card. Once the download is complete, it signals the coprocessor through interrupt. The next step is to authenticate the coprocessor OS image. This is done by all the cores in parallel using an authentication code available in fboot0. If the authentication fails, the coprocessor OS is prevented from accessing sensitive registers and intellectual properties on the card. However, authentication will allow the coprocessor OS to boot by communicating necessary information—such as the number of CPUs, memory, and capabilities—to the OS loader code and handing control over to the OS for further execution. To be able to access the sensitive area in the coprocessor, one will need the maintenance OS released by Intel and signed with a special private key that matches the public key used by the authentication code in the fboot0 authentication code sequence.

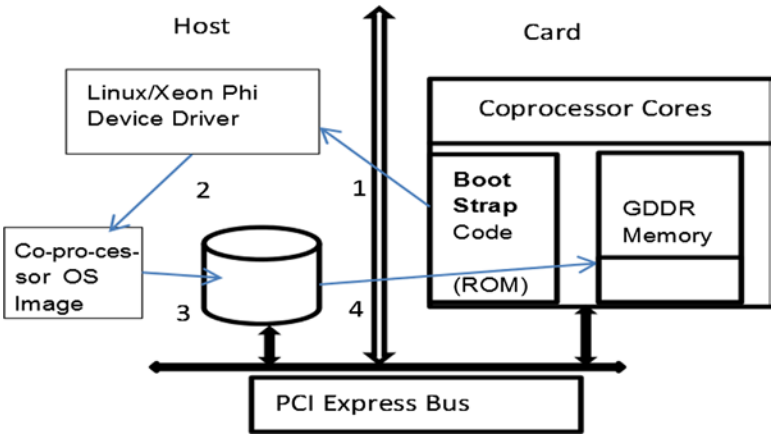

Figure 7-4. *Coprocessor OS load during boot strap process. 1. Coprocessor cores intialize and notify host. 2,3. Host driver reads the coprocessor image from the disk. 4. Host loads the coprocessor OS image to the GDDR memory. Coprocessor starts executing coprocessor OS, completing the boot process*

Coprocessor OS

The coprocessor OS provides the execution environment and runtime support to execute applications on the manycore Xeon Phi hardware. It supports other components of the MPSS stack such as SCIF communication and virtual ethernet. The coprocessor OS shipped as part of the MPSS is based on the standard Linux kernel source from `kernel.org` with minimal changes. Some of the main modifications needed are specific to the Xeon Phi hardware.

The coprocessor provides the necessary support for applications running on the coprocessor. This includes process, memory, and device and power management. One can also add loadable kernel modules to increase the functionality of the OS through kernel drivers. Intel tools such as profiling tools and debugging tools use the loadable kernel modules to interact with the hardware.

The OS provides the standard Linux base libraries such as libc, libm, librt. It also implements a minimal Shell environment using BusyBox.[2]

[2]Refer to `http://www.busybox.net/about.html` for details about BusyBox features.

Creating a Third-Party Coprocessor OS

The Xeon Phi coprocessor supports third-party OSs developed for the Xeon Phi processor. A third-party OS may be desirable for a customized operating system, such as one needing real-time response, light-weight processes, or other requirements not satisfied by the manufacturer's default OS. The boot strap code is written to conform to certain configurations required to boot the Linux OS as documented in the Linux kernel.

There are 16-bit, 32-bit, and 64-bit entry points for the Linux kernel. Xeon Phi supports the 32-bit entry points. The 32-bit entry point requires certain structures, such as the boot parameter and core and other hardware-related structures defined in the Linux documentation. The CPU mode also needs to be set in 32-bit protected mode as expected for a 32-bit entry point with paging disabled. A global descriptor table (GDT) with the proper boot code and data segment selectors must be provided, and other conditions must be fulfilled by the custom coprocessor OS as written in the boot.txt of the Linux kernel documentation.

During the boot process described earlier, the Xeon Phi processor logs the power on selftest (POST) messages to the host kernel log. These messages help in debugging the card in case the card fails to start. So if you are having issues starting the card, look at the Linux dmesg command or look at the /var/log/messages output in the host system. This inspection will require superuser privilege. Table 7-1 deciphers some of the important POST codes for your reference, which can be handy when trying to understand various log messages the card generates while booting and resetting the card.

Table 7-1. *POST Code Provided by Coprocessor to Signal Its State*

POST Code	Interpretations
01	Load Interrupt Descriptor table
02	Initialize System Components and PCIe interface
03	Set GDDR memory
04	Begin memory test
05	Creates a table to describe memory layout
06	Initialize other parts of the coprocessor
09	Enable caching
0b	Initialize application processor (cores other than boot strap processor core-0)
0c	Cache code
0d	Program multiprocessor configuration table (MP table)
0E	Copy application processor boot code to GDDR to speed up processing
0F	Wakeup application processors (AP)
10	Wait for APs to boot
11	Signal host to download coprocessor OS
12	Coprocessor in ready state to receive OS
13	Signal received from host after the coprocessor OS download completed
15	Report platform information
17	Page table setup

(*continued*)

Table 7-1. (*continued*)

POST Code	Interpretations
30-3F	GDDR Memory Training phase
40	Begin coprocessor authentication
50-5F	Coprocessor OS load and setup
bP	Interrupt 3 (int3) break point
EE	Memory test failed
F0	GDDR parameter not found in flash
F2	GDDR failed memory training
F3	GDDR memory module query failed
F4	Memory preservation failed
FF	Bootstrap finished execution
Ld	Locking down hardware access
nA	Coprocessor OS failed authentication

If you do a `sudo dmesg` after a card reboot on the host, you may find the logged message in Listing 7-1, which you can interpreted using Table 7-1. Here you see the card going through GDDR training (POST code 3C-3F) and finally getting to 'ready state' (code '12') to receive the coprocessor OS from the host. After the OS download, it starts the boot process and takes approximately 15 seconds to complete.

Listing 7-1. 'dmesg' output on host OS

```
mic0: Transition from state online to shutdown
host: scif node 1 exiting
mic0: Transition from state shutdown to resetting
mic0: Resetting (Post Code 3C)
mic0: Resetting (Post Code 3d)
mic0: Resetting (Post Code 3d)
mic0: Resetting (Post Code 3d)
mic0: Resetting (Post Code 3d)
mic0: Resetting (Post Code 3E)
mic0: Resetting (Post Code 3E)
mic0: Resetting (Post Code 3F)
mic0: Resetting (Post Code 09)
mic0: Resetting (Post Code 12)
mic0: Transition from state resetting to ready
mic0: Transition from state r eady to booting
MIC 0 Booting
Waiting for MIC 0 boot 5
Waiting for MIC 0 boot 10
MIC 0 Network link is up
```

mic0: Transition from State Booting to Online Host Driver

The MPSS ships with several host-side Linux device drivers to provide the necessary functionality. As we have seen, the host-side driver is responsible for starting the boot process of each of the cards attached to the system, loading the coprocessor OS, and setting the required boot parameters for the cards. Host drivers provide the basic communication layer through the SCIF driver layer and virtual ethernet driver layer, supporting the interfaces necessary for power management, device management, and configuration. User-level programs interact with these drivers to expose the user interface to these functionalities. For example, ssh functionality is built on top of the TCP/IP, which in turn depends on the virtual ethernet layer of host or coprocessor drivers.

The host driver load or unload functionality is provided through the standard Linux system `service start/stop` mechanism, which is provided by the commands `service mpss start/stop/restart` for booting and shutting down the card.

Linux Virtual File System (Sysfs and Procfs)

Two virtual file systems—Sysfs (/sys) and Procfs (/proc)—expose the coprocessor OS kernel and device status and control some of the behavior.

Sysfs (/sys) is a standard mechanism in Linux 2.6 for exporting information about the kernel objects to the user space. The user can query this file system content to view and manipulate these objects that map back to the kernel entities they represent. To see the usefulness of such a setup, suppose that you want to turn on huge page support in the coprocessor OS. In Chapter 4 you saw that the coprocessor supports 2MB-page sizes to reduce the TLB pressure. To reduce the TLB pressure as needed, the coprocessor can transparently promote memory pages to 2MB pages, controllable through the kernel parameter in the virtual file "enabled" located at `/sys/kernel/mm/transparent_hugepage/`.

If you cat the file, you see that the current configuration of the kernel is set to enable the transparent huge page support by default:

- `Cat /sys/kernel/mm/transparent_hugepage/enabled`, you can see the content
- `[always] madvise never`

In order to turn this default off, you can echo "never," as shown in Figure 7-5. Once you have echoed "never" to the enabled file, it is set in the kernel "enabled" property.

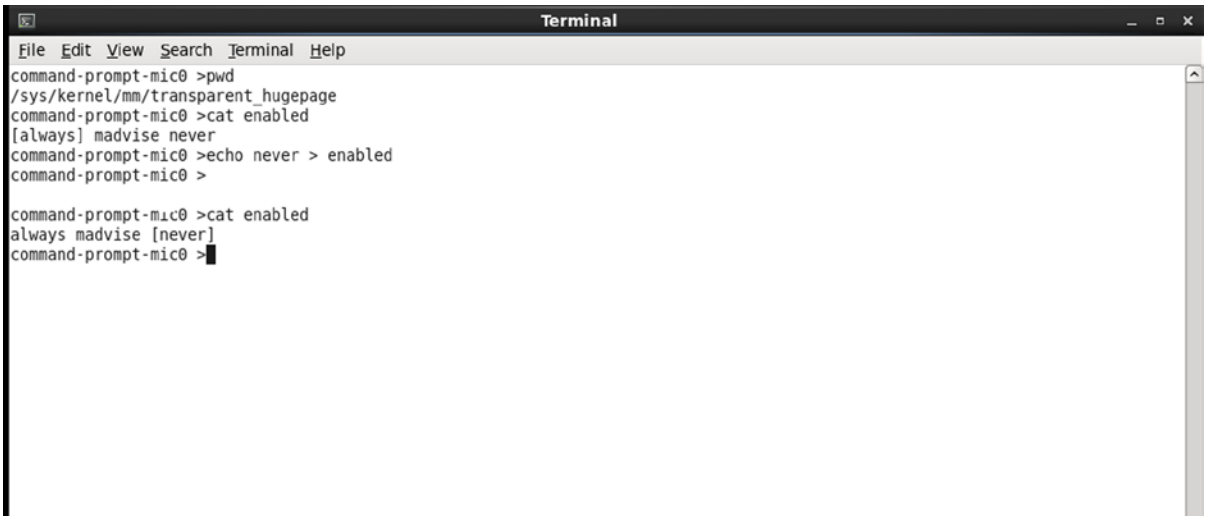

Figure 7-5. *Xeon Phi Coprocessor Sysfs virtual file system /sys*

The host also exposes some of the coprocessor information through its virtual file system. For example, on the Linux host with a Xeon Phi coprocessor, if you type the following cat command, it returns the memory size on the device in hex, which is 8GB in this case.

- `cat /sys/class/mic/mic0/memsize`
- `7c0000`

This useful interface may be used by system and card management software to query Xeon Phi configuration and status.

You can also use the /proc file system on the card to gather information related to the number of cores, memory usage, and so forth. You can type `cat /proc/cpuinfo` on the coprocessor command prompt to get information about the cores running on the coprocessor. Figure 7-6 captures the last fragment of the output of such a command on Xeon Phi. Because the hardware I was using had 61 cores with 4 threads per core, it had a total of 244 (processor id 243 since the first core is id = 0) logical processors.

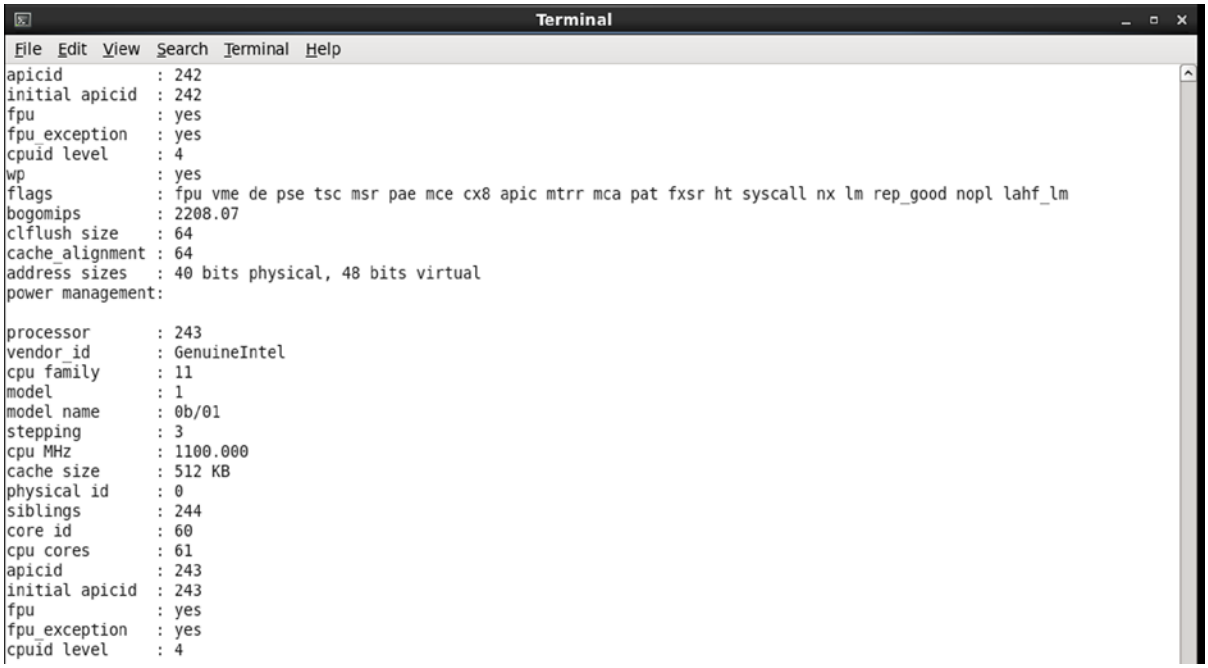

Figure 7-6. *./proc virtual file system listing CPU cores available on a Xeon Phi coprocessor*

Cluster management and monitoring software such as Ganglia[3] can use the information exposed by the Sysfs/Procfs virtual file system in the coprocessor OS and hosts to relay the data to its management interface to help manage the clusters containing Xeon Phi coprocessor cards. SCIF Layer

The SCIF layer sits above the coprocessor OS (Figure 7-3). It is a fast and lightweight communication layer that is responsible for abstracting PCIe data transfers between Xeon Phi devices and the host. It abstracts the end points (the host processor and the coprocessor) connected to the PCIe bus as "nodes" and thus can treat the nodes symmetrically. The symmetry means that the same interface is exposed to the host and the coprocessor. As a result, an application written against the SCIF interface can execute on both the host and the coprocessor.

[3]Ganglia is open-source cluster management software (http://ganglia.sourceforge.net/).

SCIF supports:

- Reliability, accessibility, and serviceability (RAS). SCIF provides the communication channel for RAS feature implementation.
- Power management SCIF supports power management events to allow the coprocessor to enter and exit PC6 states.
- The Xeon Phi coprocessor supports direct-assignment virtualization. *Virtualization* is the process of running multiple virtual machines (VM) and corresponding guest OS or applications simultaneously on the same hardware. In direct assignment, a guest OS has a dedicated Xeon Phi coprocessor and corresponding SCIF network. The SCIF networks of multiple guest OSs do not interfere with one another.
- SCIF can support an arbitrary number of coprocessors by design. The current implementation of SCIF is optimized for up to eight coprocessors.
- Various MPSS system tools supported by SCIF are treated in this chapter.

SCIF layers are implemented on both the coprocessor and the host (Figure 7-3). SCIF supports communication between the host processor and a Xeon Phi coprocessor and among Xeon Phi coprocessors connected to separate physical PCIe buses. Although SCIF supports peer-to-peer communication, it needs support from the PCIe root complex implementation of the host platform to do so.

The SCIF kernel mode interface is exposed through a category of APIs, through which the other drivers and tools can make use of its capabilities.[4]

Networking on Xeon Phi

The MPSS implements a virtual ethernet driver that emulates the Linux hardware network driver underneath network stacks on the host and the coprocessor.

The Intel Xeon Phi MPSS stack implements a virtual TCP/IP stack over the virtual ethernet interfaces and allows many tools and applications to run on top of the TCP/IP interface (Figure 7-3). In addition to basic communication support between the host and the coprocessor, there is support for the bridge to connect two Xeon Phi coprocessors over the TCP/IP network.

The support assigns unique IP addresses to each of the cards, configurable by means of a configuration file `default.cfg` supported by the MPSS. Currently, connections are class C subnets by Internet protocol definition.

If required, it is possible to create a network bridge that allows multiple coprocessors to communicate with one another on a node. If communication is expected between multiple host nodes, it is possible to do so by assigning unique addresses to the cards through the configuration process.

The virtual ethernet (VE) uses the DMA operations to transfer packets over the PCIe bus. It uses the following procedure for data packet transfer using TCP/IP:

1. The host VE device creates descriptor rings as needed by the DMA on the host memory.
2. During initialization, the host and coprocessor provide receive-buffer space using Linux socket buffer structure (skbuffs).
3. The card maps the host descriptor ring to its address space.
4. The card allocates and posts a number of receive buffers to the host-side VE driver.

[4]Refer to Intel Xeon Phi software guides for details of the interfaces, such as *Intel Xeon Phi Coprocessor System Software Developers Guide* (http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-software-developers-guide.pdf).

5. During host transfer, the host VE driver DMAs the TCP/IP buffers to the receive buffers on the coprocessor.
6. The host interrupts the coprocessor.
7. The coprocessor VE sends the data up to the TCP/IP stack and allocates a new receive buffer for host use.
8. On the coprocessor send, the skbuffs is DMAed to the receive buffer posted by the host and interrupts the host.
9. The host interrupt routine of the VE device driver sends the skbuff to the TCP/IP stack.
10. The host allocates a new receive buffer for the coprocessor to use.

Network File System

The coprocessor OS also supports the *Network File System* (NFS) to expose the host file system on the card. It is part of the MPSS. Because the NFS requires TCP/IP networking, it requires you to have the network set up so that the host can act as a file server for the NFS mount and allow access to the disks on the host.

The NFS file system is extremely handy when using the native mode of execution or the symmetric mode of application execution. These modes require the application binaries, the dependent libraries, and input dataset to be available on the coprocessor. As copying these files over would waste time, it is often possible to mount the host file system onto the coprocessor for the run. The program will then write output to its local NFS-mounted drive so that the output can be used on the host side as well without explicit copying. It is important to be aware, however, that the NFS file system is slow on Xeon Phi, and performance-sensitive files may be copied over to the coprocessor RAM drive explicitly.

The process of mounting the NFS file system is similar to the Linux NFS mount process using NFS server–client architecture. The following steps need to be executed by a superuser on both the host and the coprocessor:

1. **Export the file system you want to share with a Xeon Phi coprocessor through /etc/exports.** As a superuser you can add an entry to export fs. For example, to export the /opt/intel folder on the host where the default Intel Compiler library is available to a Xeon Phi coprocessor, you could add the following line to your /etc/exports file. `/opt/intel mic0(rw,no_root_squash)`. Then execute `exportfs –a` to make sure the modification to exportfs is seen by the NFS daemons.
2. **Allow the file exported file system to be accessible to coprocessor.** To allow the coprocessor to access the file system, set /etc/hosts.allow. You can do this by adding the following line, if 172.31.1.1 is your coprocessor card IP address. Add the appropriate address if your card address differs.

   ```
   ALL:172.31.1.1
   ```

3. **Mount the exported file system on the coprocessor.** First create a folder /opt/intel on the coprocessor through the `mkdir` command. Then issue a mount command as follows:

   ```
   mount -t nfs -o rsize=8192,wsize=8192,intr,nolock host:/opt/intel /opt/intel
   ```

Open Fabrics Enterprise Distribution and Message Passing Interface Support

One of the key requirements for Intel Xeon Phi to be applicable to high-performance computing or technical computing applications is the support for MPI communication APIs and high-performance, low-latency communication fabrics. For this purpose, the MPSS stack has built-in support for OFED. The Intel MPI stack is built on top of the OFED module to implement the *remote direct memory access* (RDMA) transport available in the Xeon Phi coprocessor. The Intel MPI library on the coprocessor can use the SCIF or physical InfiniBand host channel adaptor (HCA) between various MPI ranks running on hosts and coprocessors. These allow a coprocessor to be treated like a node in a cluster of nodes.

There are two ways to communicate with an HCA device in a node:

1. *Coprocessor Communication Link* (CCL): This is a hardware proxy driver that allows internode communication by allowing a Xeon Phi coprocessor to communicate with an InfiniBand HCA device directly. This can provide good performance in a cluster environment. The CCL provides the benefits of RDMA architecture, which allows applications to write buffers directly to a network device without kernel intervention, resulting in lower latency and higher performance data transfer. The CCL allows RDMA hardware devices on the host to be shared between the host and the Xeon Phi coprocessor. RDMA operations are performed as "verbs," which are functions that are implemented by OFED package. The user-level verbs can be executed by the application running at the same time on the Xeon Phi and the host.

2. *OFED/SCIF*: The MPI can use the TCP/IP or OFED interface to communicate with other MPI processes. The OFED on top of the SCIF interface allows InfiniBand HCA (IBHCA) on the PCIe bus to directly access the physical memory on the Xeon Phi coprocessor. If the IBHCA device is not available, the OFED/SCIF driver emulates the IBHCA device to allow OFED-based applications such as Intel MPI to run on Intel Xeon Phi without a physical HCS device. OFED/SCIF is only used for intranode communication, whereas CCL can be used for internode communication.

System Software Application Components

The MPSS provides various tools to help manage Xeon Phi–based compute nodes. These tools help in installing and managing the card, particularly in creating user accounts on the coprocessor OS and other utilities to query the cards' status. The sections that follow will look at some important components of the MPSS system software that can help you manage system functions.

micinfo

The micinfo utility, together with the micchek utility (discussed later in this section), can be used to see whether your Xeon Phi hardware is up and running and to collect information about the hardware and system software on your installation. By default, the system installed software puts it in the /opt/intel/mic/bin path. The command must be run in superuser mode to execute properly. The output of an instance of `micinfo` execution is displayed below. Here I invoked the `micinfo` command without any arguments, which tells it to print all the information about the device and

the system software. You can also invoke it with specific arguments, such as –listdevice or –deviceinfo, to display only part of the following output. The output is categorized in the following sections:

- *System software information*: The host OS version and the Xeon Phi coprocessor MPSS version numbers are displayed. Also displayed for each Xeon Phi device on the platform are the logical device number and name, the firmware version (flash version), the coprocessor OS version, and other information such as the SMC boot loader version and device serial number.
- *Board information*: Information relevant to the board is displayed, including the coprocessor stepping, board SKU, ECC mode on the card, vendor, and system.
- *Core information*: This section is specific to the coprocessor core, including information on the voltage, frequency, and number of active cores.
- *Thermal section*: Information includes the SMC firmware version and the die temperature.
- *Memory section*: This GDDR-related information includes supplier, size, and data transfer speed.

Listing 7-2 provides a sample micinfo utility log.

Listing 7-2. Sample micinfo Utility Log

```
Created Wed May 15 09:26:02 2013

    System Info
            HOST OS                 : Linux
            OS Version              : 2.6.32-279.el6.x86_64
            Driver Version          : 5889-16
            MPSS Version            : 2.1.5889-16
            Host Physical Memory    : 65917 MB

Device No: 0, Device Name: mic0

    Version
            Flash Version           : 2.1.02.0383
            SMC Boot Loader Version : 1.8.4326
            uOS Version             : 2.6.38.8-g9b2c036
            Device Serial Number    : [XXXXX]

    Board
            Vendor ID               : 8086
            Device ID               : 225c
            Subsystem ID            : 2500
            Coprocessor Stepping ID : 3
            PCIe Width              : x16
            PCIe Speed              : 5 GT/s
            PCIe Max payload size   : 256 bytes
            PCIe Max read req size  : 4096 bytes
            Coprocessor Model       : 0x01
            Coprocessor Model Ext   : 0x00
            Coprocessor Type        : 0x00
            Coprocessor Family      : 0x0b
            Coprocessor Stepping    : B1
```

```
        Board SKU                : B1QS-7110P
        ECC Mode                 : Enabled
        SMC HW Revision          : Product 300W Passive CS

    Cores
        Total No of Active Cores : 61
        Voltage                  : 1013000 uV
        Frequency                : 1100000 KHz

    Thermal
        Fan Speed Control        : N/A
        SMC Firmware Version     : 1.13.4570
        FSC Strap                : 14 MHz
        Fan RPM                  : N/A
        Fan PWM                  : N/A
        Die Temp                 : 55 C

    GDDR
        GDDR Vendor              : -----
        GDDR Version             : 0x1
        GDDR Density             : 2048 Mb
        GDDR Size                : 7936 MB
        GDDR Technology          : GDDR5
        GDDR Speed               : 5.500000 GT/s
        GDDR Frequency           : 2750000 KHz
        GDDR Voltage             : 1000000 uV
```

micflash

This utility is installed by default in the /opt/intel/mic/bin path and is used to update the coprocessor firmware, also known as flash memory. It can also be used to query and save the existing flash version before update. It is advisable to check compatibility of a given flash to the device in question by using –compatible switch before doing the flash update.

micsmc

The micsmc utility shows the coprocessor status. It can run in graphical or text mode interface. In graphical mode, it displays for all the Xeon Phi coprocessors installed in the system the details of such features as core utilization, memory usage, temperature, and power and error log. This utility can also be used for setting coprocessor features like ECC, turbo, and others. For example to turn off ECC you can run the sequence provided in Listing 7-3.

Listing 7-3. Sequence to Turn Off ECC

```
command_prompt_host > micctrl –r
command_prompt_host > micctrl –w
command_prompt_host > /opt/intel/mic/bin/micsmc  --ecc disable
command_prompt_host > service mpss restart
To check  ECC status:
command_prompt_host > /opt/intel/mic/bin/micsmc --ecc status
```

miccheck

The miccheck utility is used to check the current install status of a Xeon Phi installation in a node. It detects all devices and checks for proper installation, driver load status, status of the Xeon Phi network stack, and successful boot by POST code. An instance of `miccheck` output is displayed in Listing 7-4. It shows that the test failed because the flash version did not match the install log file (the manifest).

Listing 7-4. Sample miccehck Utility Output

```
miccheck 2.1.5889-16, created 17:08:24 Mar  8 2013
Copyright 2011-2013 Intel Corporation  All rights reserved

Test 1 Ensure installation matches manifest : OK
Test 2 Ensure host driver is loaded         : OK
Test 3 Ensure drivers matches manifest      : OK
Test 4 Detect all listed devices            : OK
MIC 0 Test 1 Find the device                       : OK
MIC 0 Test 2 Check the POST code via PCI           : OK
MIC 0 Test 3 Connect to the device                 : OK
MIC 0 Test 4 Check for normal mode                 : OK
MIC 0 Test 5 Check the POST code via SCIF          : OK
MIC 0 Test 6 Send data to the device               : OK
MIC 0 Test 7 Compare the PCI configuration         : OK
MIC 0 Test 8 Ensure Flash versions matches manifest : FAILED
MIC 0 Test 8> Flash version mismatch. Manifest: 2.1.01.0385, Running: 2.1.02.0383
Status: Test failed
```

micctrl

Micctrl is the Swiss army knife for the system administrator. This tool can be used to boot, control, and status check the coprocessor. After the system software is installed on a coprocessor host, `micctrl --initdefaults` is used to create the default set of the coprocessor OS boot configuration files. Any change in configuration can be reflected by using the `micctrl --resetconfig` command. The coprocessor can be booted, shut down, reset, and rebooted using micctrl commands with --boot,--shutdown, --reset, and --reboot commands, respectively. Since the coprocessor runs an OS with full user access, the micctrl utility also allows the system administrator to add, remove, or modify users or groups by managing the /etc/passwd file on the coprocessor.

micrasd

The micrasd process can be run on the host to handle log hardware errors and test hardware in the maintenance mode. It can also run as a host Linux system utility as `service micras start/stop` commands.

micnativeloadex

Micnativeloadex is a handy utility to run a cross-compiled binary targeted for the Xeon Phi coprocessor to be launched from the host. It also allows the same functionality from inside the coprocessor to reverse offload. When executed, it copies the binary and dependent libraries to the destination without needing to do it explicitly, as would be the case when using the scp/ssh command.

Summary

This chapter examined the system software layers needed to support the operations of the Intel Xeon Phi coprocessor. There are three major layers of the system software. Two of these layers run at Ring 0 protection level, which allows them to provide low-latency access to the Xeon Phi hardware. The bottommost layer is the Linux kernel or the Linux coprocessor OS, which interfaces with the hardware and provides basic OS supports, including memory and process management functionalities. The device-driver layer runs on top of the basic OS layer and provides interfaces for the applications and utilities running in Ring 3 layer to access and interact with the Xeon Phi hardware necessary to manage, query, and execute applications on the coprocessor.

The next chapter will look at the tools available for developing applications on Xeon Phi and provide some pointers on how to use these tools.