International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 5 333 - 336

Enhancement of DNVME device driver

Sucheta Shivakalimath
M. Tech in Software Engineering
Dept. of ISE,
R. V. College of Engineering
Bengaluru, India
sucheta032@gmail.com

Dr. Ramakanth Kumar P.
Professor and Dean Academics
Dept. of ISE,
R. V. College of Engineering
Bengaluru, India
ramakanthkp@rvce.edu.in

Abstract- The device driver is the interface between hardware and software applications. It includes all the functionality for handling the devices connected to it. The drivers are device-specific. The storage devices like SSD and HDD use dNVMe driver for handling them. This driver can be enhanced for supporting various features. The enhancement helps in development of storage devices. The main areas for modification includes enhancing IOCTL calls, allowing register level changes and allowance for negative testing. These features will enrich the storage devices for all the qualifications.

Keywords: *Device Drivers, Non-Volatile Memory express (NVMe, Submission Queue (SQ), Completion Queue (CQ), Interrupts.*

*****

I. INTRODUCTION

One of the numerousbenefits of free operating systems, as characterized by Linux, is that their internals are exposed for everyone to view. Earlier the code of operating system was not given for user access. But now it is freely available to understand and also for modification.Linux has helped to democratize operating systems. The Linux kernel is a large withcomplex body of code. But it is prone to hacking. The kernel hackers find and access pointfrom where they can contact all the code. The device drivers act as gateway for such situations [1].

Device drivers are special portion in the Linux kernel [2]. These are separate black boxes which make specific part of hardware retort to a well-defined internal programming interface; they conceal entirely the details regarding how the device works. A set of standardized calls are used to accomplish user activities. These calls are self-governing of specific driver. The device driver then has the role of mapping these calls to device-specific operations which perform on real hardware. The modularity that makes the Linux drivers easy to write is that the programming interface for drivers can be developed independent from the rest of the kernel. There are more than hundreds of them obtainable.

dNVMe is the kernel component of the NVMe Compliance Test Suite [3][4]. The user space application tNVMe wires up components within dNVMe to create various compliance test cases.dNVMe is not the NVMe driver embedded within the Linux kernel. These 2 separate code bases target different audiences and greatly differ in the support they offer to a user space application. Some of the major differences can be summarized as follows:

1. dNVMe driver allows sending illegal commands and allows creating illegal states to verify proper hardware error code generation whereas NVMe driver prevents from sending illegal commands.
2. dNVMe supports every feature of the NVMe specification.
   a. Meta data support
   b. All Admin and NVM command set commands
   c. Allows using MSI, MSI-X, and polling for reaping CE's from IOCQ's.
3. dNVMe driver Opens kernel level resources like queues and memory facilitating maximum visibility for debugging support whereas NVMe driver purposely hides kernel level constructs.
4. dNVMe driver doesn't do anything automatically, must be specifically instructed by a user space application. NVMe driver automates sending commands on behalf of user space applications.
5. NVMe driver safeguards any application but dNVMe driver allows situations which could crash the kernel, an undesirable side effect as a result of allowing maximum interaction with user space applications.

dNVMe logic is platform specific and targets Linux kernel versions based on 2.6.35. It has been developed on Ubuntu distributions only. However, the driver design is generic and should support other Linux kernel versions by changes in the required kernel API's. Additionally, tNVMe could be used without modifications on other platforms if one were to implement the IOCTL's within dNVMe on those other platforms.

dNVMe is a test driver with a goal to verify hardware compliance against a written set of specifications. Functionality, not speed, was the main target for the driver. It was seen that satisfying both speed and functionality could not be addressed simultaneously in all aspects of this design. Thus when a decision had to be made as to which one to choose, functionality always won. As a result we ended up making dNVMe driver a character driver rather than a block driver with an advanced range of IOCTL's to improve user space control required for testing.

II. EXISTING SYSTEM

The existing dNVMe driver provides various well-known functionalities for handling devices appropriately. The different functions are listed with the various tasks that they perform.

1. Commands:This function takes care of all the commands to be sent to device and receive from the device. The commands are sent through SQ from host to device and received through CQ from device to host. The tasks handled are:
    a. Processes all the commands to be sent from SQ to CQ.
    b. Manages storage of commands.
2. Data structure:This function is to create a data structure and manage it. It takes care of all the memory related operations. The tasks are:
    a. Allocate memory for user space and data in kernel space
    b. Copy userspace buffer to kernel memory
    c. Logging metadata and IRQ nodes into user space file
3. IOCTL: The I/O control (IOCTL) is the system call for particular I/O operations. The calls that are not done by regular calls are carried out by IOCTL calls. The tasks are:
    a. Checks the status of device
    b. Performs driver generic read and write
    c. Creates driver admin SQ and CQ and allocates kernel space
    d. Initializes driver IOCTL calls
    e. Creates and deletesmetabuffers with freeing memory after deletion of metabuffer
    f. Acquire queue metrics from global data structure
4. IRQ: The interrupt is a signal for intervening the running operation and perform interrupted event. The various operations are:
    a. Set new IRQ scheme, initialize the IRQ list before any scheme runs and releases IRQ list after any scheme runs
    b. Disables pin to read the command register in PCI space
    c. Validates IRQ inputs for MSI&MSI-X, and also takes care of masking & unmasking interrupts
    d. Allocates and deallocatesIRQ and interrupt CQ nodeand frees memory.
5. Queue: The queue is important part. It is used mainly for sending and receiving of commands and messages. There are different tasks taken care by this function. They are:
    a. Checks if the controller has transitioned its state after controller reset and enables/disables controller
    b. Clean existing driver data structure
    c. Creation of SQ, CQ and I/O SQ
    d. Deallocation of memory after CQ and SQ is deleted
    e. Inquire the number of commands in CQthat are waiting to be reaped
    f. Finds SQ, CQ, command, metadata node and removes command node, SQ node, CQ node
    g. Copyingof CQ data to user buffer for elements reaped and moves the CQ head pointer to point to location of the elements that is to be reaped
    h. Reap the number of elements specified for the given CQ id and send the reaped elements back.
6. Register: This function takes care of all the operations related to registers. The task performed is,
    a. Reads and writes the controller registers located in the PCI BAR 0 and 1 that are mapped to memory area which supports in-order access.
7. Status check: This is the function for checking the status of device with the power management. The tasks are as follows.
    a. Checks PCI device status and controller status
    b. Checks if the NVME device supports NEXT capability item in the linked list
    c. Performs PCI power management control and status
    d. Checks the MSI and MSI-X control and status
    e. Checks the PCIe capable status register of advanced error reporting capability of PCI express device.
8. dNVMe system: The main function which handles all the above mentioned functions is dNVMe system. The tasks are as given.
    a. Enter and initialize the dNVMe driver
    b. Exit the dNVMe driver probe
    c. Examine and remove the dNVMe driver
    d. Get the device list with their metrics and unlock the device
    e. Opens and releases the dNVMe driver
    f. Mapping of the contiguous device mapped area to user space
    g. Take care of IOCTL calls

III. PROPOSED SYSTEM

The existing dNVMe driver provides all the basic functionality to be performed with the device. But it lacks few main areas like IOCTL calls for CQ, providing security, and getting address information from kernel. These new features can be developed and put into the existing system.

IV. IMPLEMENTATION LOGIC

The enhancement for dNVMe device driver can be given by supporting the mentioned three features. An approach is given for implementing these new features.

A. IOCTL calls for CQ

First, the initialization is done. The memory has to be allocated for user structure in kernel space. Then the SQ is acquired. The command size is known and memory is

allocated for command in kernel space. Second, the driver rings SQ doorbell. Retrieve the queue from the linked list, copy the tail pointer with virtual pointer and write the tail pointer value to SQ tail doorbell. Third, reap the CQ. Check for number of unreaped elements in CQ. If this CQ is an IOCQ, not ACQ, then lookup the component entry size. Get the required base address and copy the number of component entry's that should be able to reap. Fourth, free the command identifier node from the command track list. Fifth, process the queues. Get the persistence queue identifier, unique command identifier and allocate memory to copy user data. Sixth, process the admin commands. Perform deletion of IOSQ, creation of IOSQ, deletion of IOCQ and creation of IOCQ. Seventh, process the reaped algorithms. Find the SQ node for given SQ identifier and find the command in SQ node. Eighth, copy the CQ data to user buffer for the elements reaped. Lastly, move the CQ head pointer to point to location of the elements that is to be reaped.

B. Providing locks

There are four layers of locking in dNVMe.

a. The device list read/write lock: The device list lock protects the list of devices that dNVMe maintains.
b. The device lock: It protects the CQ list, the device, the meta, and the IRQ process, as well as the IRQ track node list in the IRQ process structure.
c. The IRQ Lock: protects IRQ track structure
d. The CQ Lock: The CQ Lock protects the CQ structure, and all SQ structures in its list

First, find the SQ node in the given device element node and given SQ identifier. If SQ node is found, it locks the associated CQ and return pointer to the SQ node in the SQ linked list. Then locking on the dNVMe driver is performed. Second, unlock the CQ from SQ. Third, lock the CQ. Find CQ node in the given device element node and given CQ id. If found, returns the pointer to the CQ node in the CQ linked list.

C. Address information

Get addresses from operating system about the PCIe device such as module layout, character device allocation, PCI bus read configuration byte, kernel stack, PCI reset slot, PCI disable device, PCI disable MSI-X, PCI enable MSI-X, PCI enable MSI, and PCI disable MSI.

V. INFERENCE

With these new features, the facts that are observed are given below,

- The enhanced driver can give better performance when compared with existing driver.
- The address information feature helps in performing any register level changes.
- The locking of queues is advantageous for security purposes.

Figure 1 shows the time breakdown. The native driver is compared with the enhanced driver on storage device.

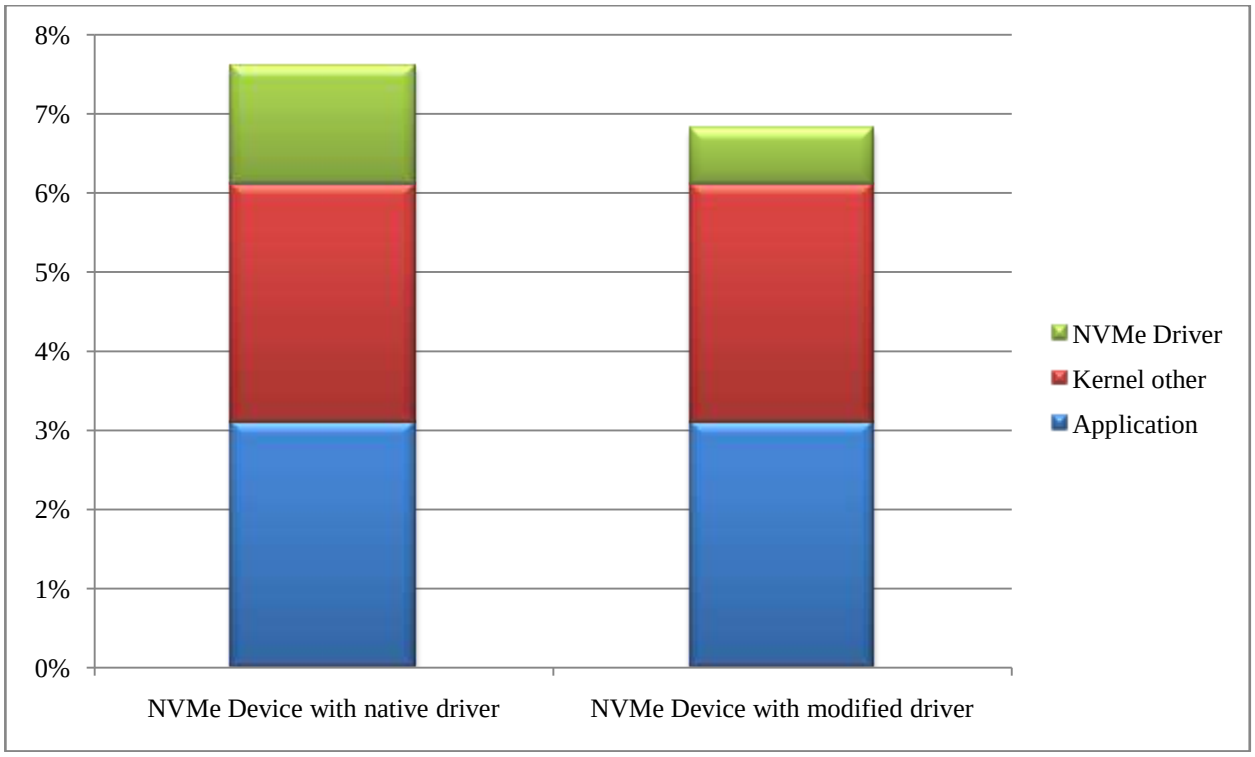

Figure 1: Time spent in different parts of I/O stack

The time spent by devices in driver is important factor to be considered. The device in native driver spends 1.5% of overall time while in the modified driver it spends half the time of native driver.

VI. CONCLUSION

The attention for developing Linux device drivers gradually upsurges as the admiring of Linux system endures to grow. Many users are unaware of issues regarding hardware, and majority of Linux is autonomous of the hardware it runs on. However, the driver exists without which there is no functioning system. The enhancement of device drivers provides various advantages with new features. These help in development of storage devices with even more less time and efforts.

ACKNOWLEDGEMENT

Any achievement, be it scholastic or otherwise does not depend solely on the individual efforts but on the guidance, encouragement and cooperation of intellectuals, elders and friends. We thank Department of Information Science and Engineering, RVCE for their constant support and encouragement.

REFERENCES

[1] "Linux device drivers", A. Rubini and J. Corbet, Sebastopol: O'Reilly & Associates, 2001.

[2] A. Kadav and M. Swift, "Understanding modern device drivers", *ACM SIGARH Computer Architecture News*, vol. 40, no. 1, 2012, p. 87.

[3] "NVM Express Specification, Revision 1.1a," NVMHCI Workgroup, Tech. Rep., September 23, 2013.

[4] Sivashankar and S. Ramasamy, "Design and implementation of non-volatile memory express," *Recent Trends in Information Technology (ICRTIT), 2014 International Conference on*, Chennai, 2014, pp. 1-6.