A non-volatile memory (NVM) express (NVMe) system includes at least one user application, an NVMe controller and a hypervisor. Each user application runs in a respective virtual machine environment and including a user input/output (I/O) queue. The NVMe controller is coupled to at least one NVM storage device, and the NVMe controller includes a driver that includes at least one device queue. The hypervisor is coupled to the user I/O queue of each user application and to the NVMe controller, and selectively forces each user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller or enables a private I/O channel between the user I/O queue and a corresponding device queue in the driver of the NVMe controller.
PRIOR ART

FIG. 1
2 OSLib ISubmit()

3 Submission Queue ...
4 IOResult ...

5 If !IOResult then

7 IOResult.vmKCmd ...
8 IOResult.devDATA ...
9 OrigLink ...
10 IOResult.link ...

11 !sync(submissionQueue.link ...)

12 OrigLink ...
13 IOResult.link ...

14 callIOServ( )

15 return VMK_OK

End

6 Return

FIG. 3
400

2 userLib → kernelDriver

4 if curApp ...

Y

5 new subQueue
6 new compQueue
7 subQueue ...
8 compQueue ...
9 curHandle ...
10 curHandleList+ ...

11 ForwardI0Cmd ...

End

FIG. 4
HYBRID FRAMEWORK OF NVME-BASED STORAGE SYSTEM IN CLOUD COMPUTING ENVIRONMENT

CROSS-REFERENCE TO RELATED APPLICATION


TECHNICAL FIELD

[0002] The subject matter disclosed herein generally relates to non-volatile memory express (NVMe) systems, and more particularly, to a framework for an NVMe-based virtual machine-hypervisor (VM-Hypervisor) platform.

BACKGROUND

[0003] Recently, data centers have been replacing traditional Serial AT Attachment (SATA) and Serial Attached SCSI (SAS) solid-state drives (SSDs) with NVMe SSDs due to the outstanding performance associated with NVMe devices. For historical reasons, however, the current deployments of NVMe in virtual machine-hypervisor-based platforms, such as VMware ESXi, have many intermediate queues along the I/O path, which cause performance bottleneck by locks in the queues. There may also be a response-time delays caused by cross-VIM interference. Most significantly, however, the up-to-64K-byte capability of NVMe SSDs is not being fully utilized in virtual machine-hypervisor-based platforms.

SUMMARY

[0004] An example embodiment provides a non-volatile memory (NVM) express (NVMe) system that may include at least one user application, an NVMe controller, and a hypervisor. Each user application may be running in a respective virtual environment and including a user input/output (I/O) queue. The NVMe controller may be coupled to at least one NVMe storage device and the NVMe controller may include a driver that includes at least one device queue. The hypervisor may be coupled to the user I/O queue of each user application and to the NVMe controller, and the hypervisor may force each user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller. In one embodiment, the hypervisor may send false completion messages to each user I/O queue to force the user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller. In one embodiment, the hypervisor may select an idle I/O server to send I/O traffic in the device queue to an NVMe storage device. In one embodiment, the hypervisor may determine an optimum number of device queues based on an Erlang function.

[0005] Another embodiment provides a non-volatile memory (NVM) express (NVMe) system that may include at least one user application, an NVMe controller and a hypervisor. Each user application may be running in a respective virtual environment and including a user input/output (I/O) queue. The NVMe controller may be coupled to at least one NVMe storage device, and the NVMe controller may include a driver that includes at least one device queue. The hypervisor may be coupled to the user I/O queue of each user application and to the NVMe controller.

The hypervisor may enable a private I/O channel between the user I/O queue and a corresponding device queue in the driver of the NVMe controller. In one embodiment, the private I/O channel enabled between the user I/O queue and the corresponding device queue may be enabled for at least one thread of the corresponding user application. In another embodiment, each private I/O channel may include a corresponding I/O server in the driver of the NVMe controller.

[0006] Still another example embodiment provides a non-volatile memory (NVM) express (NVMe) system that may include at least one user application, an NVMe controller and a hypervisor. Each user application may be running in a respective virtual machine environment and including a user input/output (I/O) queue. The NVMe controller may be coupled to at least one NVMe storage device, and the NVMe controller may include a driver that includes at least one device queue. The hypervisor may be coupled to the user I/O queue of each user application and to the NVMe controller. The hypervisor may selectively force each user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller or may enable a private I/O channel between the user I/O queue and a corresponding device queue in the driver of the NVMe controller. In one embodiment, if the hypervisor selectively forces each user I/O queue to empty, the hypervisor sends false completion messages to each user I/O queue to force the user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller, and if the hypervisor selectively enables a private I/O channel between the user I/O queue and a corresponding device queue the private I/O channel, the hypervisor enables the private channel for at least one thread of the corresponding user application.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] In the following section, the aspects of the subject matter disclosed herein will be described with reference to exemplary embodiments illustrated in the figures, in which:

[0008] FIG. 1 depicts a simplified block diagram of a conventional NVMe-based VM-Hypervisor platform;

[0009] FIG. 2 depicts a simplified block diagram of an NVMe-based VM-Hypervisor platform according to the subject matter disclosed herein;

[0010] FIG. 3 depicts the example parallel queue mode process set forth in Table 1 in a flowchart format; and

[0011] FIG. 4 depicts the example direct access mode process set forth in Table 2 in a flowchart format.

DETAILED DESCRIPTION

[0012] In the following detailed description, numerous specific details are set forth in order to provide a thorough understanding of the disclosure. It will be understood, however, by those skilled in the art that the disclosed aspects may be practiced without these specific details. In other instances, well-known methods, procedures, components and circuits have not been described in detail not to obscure the subject matter disclosed herein.

[0013] Reference throughout this specification to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in connection with the embodiment may be included in at least one embodiment disclosed herein. Thus, the appearances of the phrases “in one embodiment” or “in an embodiment” or “according to one embodiment” (or other phrases having
similar import) in various places throughout this specification may not be necessarily all referring to the same embodiment. Furthermore, the particular features, structures or characteristics may be combined in any suitable manner in one or more embodiments. In this regard, as used herein, the word “exemplary” means “serving as an example, instance, or illustration.” Any embodiment described herein as “exemplary” is not to be construed as necessarily preferred or advantageous over other embodiments. Also, depending on the context of discussion herein, a singular term may include the corresponding plural forms and a plural term may include the corresponding singular form. It is further noted that various figures (including component diagrams) shown and discussed herein are for illustrative purpose only, and are not drawn to scale. Similarly, various waveforms and timing diagrams are shown for illustrative purpose only. For example, the dimensions of some of the elements may be exaggerated relative to other elements for clarity. Further, if considered appropriate, reference numerals have been repeated among the figures to indicate corresponding and/or analogous elements.

The terminology used herein is for the purpose of describing particular exemplary embodiments only and is not intended to be limiting of the claimed subject matter. As used herein, the singular forms “a,” “an” and “the” are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms “comprises” and/or “comprising,” when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components, and/or groups thereof. The terms “first,” “second,” etc., as used herein, are used as labels for nouns that they precede, and do not imply any type of ordering (e.g., spatial, temporal, logical, etc.) unless explicitly defined as such. Furthermore, the same reference numerals may be used across two or more figures to refer to parts, components, blocks, circuits, units, or modules having the same or similar functionality. Such usage is, however, for simplicity of illustration and ease of discussion only; it does not imply that the construction or architectural details of such components or units are the same across all embodiments or such commonly-referenced parts/modules are the only way to implement the teachings of particular embodiments disclosed herein.

Unless otherwise defined, all terms (including technical and scientific terms) used herein have the same meaning as commonly understood by one of ordinary skill in the art to which this subject matter belongs. It will be further understood that terms, such as those defined in commonly used dictionaries, should be interpreted as having a meaning that is consistent with their meaning in the context of the relevant art and will not be interpreted in an idealized or overly formal sense unless expressly so defined herein.

Embodiments disclosed herein provide a hybrid framework for an NVMe-based VM-Hypervisor platform. The hybrid framework, which is referred to herein as “H-NVMe,” provides two virtual machine (VM) I/O path deployment modes. A cloud manager may select from the two modes of the H-NVMe framework for VM deployment in an NVMe-based storage system. The first mode, referred to herein as a Parallel Queue Mode (PQM), increases parallelism and enables lock-free operations by forcing the adapter queue of the hypervisor to be empty, and using enhanced queues in the NVMe driver to accelerate I/O requests. The second mode, referred to herein as a Direct Access Mode (DAM), may achieve better performance than the PQM by bypassing all queues in the hypervisor layer by using an IOFilter file that is attached to each VM disks (VMDKs), thereby allowing the threads of user applications to directly access NVMe driver queues. Although the second mode may be used to encapsulate the storage resources provided by a hypervisor platform, the DAM may suit premium users that may have higher permission-control processes. Specifically, currently a hypervisor abstracts the storage resource into virtualized disks, and VMs have almost no knowledge of the physical information of the disks they are using, and in particular, the VMs cannot directly access their I/O streams to the up-to-64K queues in the NVMe SSDs. In one embodiment, the H-NVMe framework disclosed herein may achieve at least a 50% throughout improvement compared to conventional NVMe solutions.

Fig. 1 depicts a simplified block diagram of a conventional NVMe-based VM-Hypervisor platform. In one embodiment, the VM Hypervisor platform depicted in Fig. 1 may be based on the VMware ESXi Pluggable Storage Architecture (PSA). The conventional VM-Hypervisor platform may include a user layer, a hypervisor layer, and a storage device layer. The user layer may include a plurality of virtual machines, of which only virtual machines 100c-110a are indicated. Each virtual machine 110 may be running an application APP 111 under an operating system GuestOS 112. The respective operating systems GuestOS 112 may be different from each other.

The hypervisor layer 102 may include a VM monitor 121, a virtual SCSI interface vSCSI 122, and a storage stack 123. The VM monitor 121 may directly monitor and manage each of the virtual machines 110 existing at the user layer 101. The vSCSI 122 may provide a virtual SCSI interface to each virtual machine monitored and maintained by the VM monitor 121. The storage stack 123 may be used to manage the I/O traffic between the vSCSI 122 and the storage layer 103.

The storage device layer 103 may include a plurality of NVMe SSD devices (not indicated), a driver 131, a host bus adapter 132, a network fabric 133 and one or more NVMe controllers 134, of which only one NVMe controller is indicated. The driver 131 may receive the I/O traffic from the storage stack 123 and direct the I/O traffic to the host bus adapter 132. The host bus adapter may control the I/O traffic over the network fabric 133 to the NVMe controller 134, which may control I/O traffic to and from an NVMe SSD device (not indicated). There are up to 64K I/O queues that are available in the NVMe controller 133.

There may be multiple intermediate queues in the I/O path between the user layer 101 and the storage device layer 103. When an I/O request enters the I/O path in the conventional NVMe-based VM-Hypervisor platform, the following process generally occurs. In the user layer 101, the platform world queue 141 is responsible for each VM I/O request. There may be one or more I/O queues 142 for each VM 110. The hypervisor layer 102 includes an adapter queue 151 that may gather the I/O requests from the world queue 141. The hypervisor layer 102 may translate each I/O request into a SCSI command and may send the translated I/O request to a device queue 161 in the driver 131. An
The conventional NVMe-based VM-Hypervisor platform 100 completes nearly all I/O requests asynchronously via interrupts. Historically, interrupt-driven I/O may be efficient for high-latency devices, such as hard disks, but interrupt-driven I/O may also introduce a significant amount of overhead that includes many context switches. Moreover, interrupt-driven I/O requests are not efficient for NVMe SSDs. More importantly, an interrupt-driven approach is not lock-free and thereby limits the parallel I/O capacity of the conventional NVMe-based VM-Hypervisor platform 100 because an upper layer-JOSeq (i.e., an IO server having a role to serve each I/O request, and may have different names in different hypervisor solutions) will not send another I/O request until a previous submitted I/O returns a pending status. Consequently the multiple cores/multiple-queue mechanisms of the NVMe devices are not fully utilized.

FIG. 2 depicts a simplified block diagram of the NVMe-based VM-Hypervisor platform 200 according to the subject matter disclosed herein. Similar to the VM-Hypervisor platform 100 depicted in FIG. 1, the VM-Hypervisor platform 200 may include a user layer 201, a hypervisor layer 202, and a storage device layer 203.

The user layer 201 may include a plurality of virtual machines, of which only virtual machines 210_a-210_b are indicated. Each virtual machine 210 may be running an application APP 211 under an operating system GuestsOS 212. The respective operating systems GuestsOS 212 may be different from each other.

The hypervisor layer 202 may include a VM monitor 221, a virtual SCSI interface vSCSI 222, and a storage stack 223. The VM monitor 221 may directly monitor and manage each of the virtual machines 210 existing at the user layer 201. The vSCSI 222 may provide a virtual SCSI interface to each virtual machine monitored and maintained by the VM monitor 221. The storage stack 223 may be used to manage the I/O traffic between the vSCSI 222 and the storage layer 203.

The storage device layer 203 may include a plurality of NVMe SSD devices (not indicated), a driver 231, a host bus adapter 232, a network fabric 233, and one or more NVMe controllers 234, of which only one NVMe controller is indicated. The driver 231 may receive the I/O traffic from the storage stack 223 and direct the I/O traffic to the host bus adapter 232. In one embodiment, the host bus adapter may control the I/O traffic over the network fabric 233 to the NVMe controller 234, which may control I/O traffic to and from an NVMe SSD device (not indicated). There are up to 64K I/O queues that are available in the NVMe controller 233.

The H-NVMe framework permits an operator of the NVMe platform 200 to select either the parallel queue mode or the direct access mode for a particular VM. As depicted in the center portion of FIG. 2, in the parallel queue mode the adapter queue 251 of the hypervisor layer 202 is forced to be emptied (i.e., len=0) by the device queue 261 sending “false” completion signals so that I/O requests in the respective user queues 242 of the world queue 241 are sent to the device queue 261. After the I/O requests are received in the device queue 261, the parallelism of the I/O servers 262 speeds up the I/O request process by each I/O server 262 focusing only on the I/O request contents of their respective queue.

Table 1 sets forth pseudocode for an example process of the parallel queue mode. FIG. 3 depicts the example parallel queue mode process set forth in Table 1 in a flowchart format with the line numbers in Table 1 indicated in FIG. 3.

<table>
<thead>
<tr>
<th>Procedure submissionNVMeHypervisor ( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
</tbody>
</table>

\[
\frac{\text{ErlangC}(\lambda, \mu, n)}{\lambda^2 + \frac{1}{\mu^2}}
\]

\[
\frac{\lambda}{\mu^2}
\]

\[
\frac{1}{1 + (1 - \rho) \frac{c_1}{(c_1)^2} \sum_{k=1}^{c_1} \frac{(c_1)^2}{k^2}}
\]

\[
\text{ErlangC}(\lambda, \mu, n)
\]

\[
\frac{\lambda}{\mu^2}
\]
by multiple I/O Serv threads by focusing on their own queue. (See, for example, lines 11-13 of the process in Table 1.) In particular, the callI0Serv function selects an idle I0Serv in order to assign jobs in a round-robin manner.

[0030] While having multiple I0 servers 262 may improve performance, it should be kept in mind that increasing the number of I0 servers 262 may also cause more processing and memory resources to be occupied by the additional threads formed by the multiple I0 servers 262. Additionally, as the number of I0 servers 262 increases, the service rate of each I0 server 262 may decrease accordingly. An adaptive process that may be used to determine an appropriate number of I0 servers 262 may be as follows. The length of the adapter queue 251 is forced to be 0, the problem of determining how to dynamically assign subqueue and I0 servers may be addressed using the M/M/c queueing theory. For example, let the total arrival rate from the adapter queue to be \( \lambda \), and let vector \( \mu \) denote each I0 Serv’s service rate. The vector \( \mu \) has c dimensions, in which \( e \in \{1, \ldots, c_{\text{max}}\} \) is the number of I0 Servs, and \( c_{\text{max}} \) is the pres set maximal I0 Serv number. If the number c is increased, the service rate of each server will change, and the change may be estimated by a regression function based on periodically measurement. (See, for example, line 24 of the process in Table 1.) Once \( \lambda \) and \( \mu \) have been determined, the H-NVMe framework calls the optimizeServNum() function to determine the best number for c.

[0031] The optimization objective of in this process is to minimize the total latency. Specifically, different combinations of c and corresponding service rates may be tested. This procedure may also use the Erlang-C function to calculate the probability that an arriving job will need to be queued as opposed to being immediately served. Lastly, in order to reduce the cost of changing c, the frequency of updating c may be limited to a preset update epoch window epochLen. (See, for example, line 22 of the process in Table 1.)

[0032] Although PQM may improve queuing performance of an NVMe-based VM-Hypervisor platform by moving all I/O request from the adapter queue of the hypervisor to customized multiple subqueues in the driver queue, the PQM still may not simplify the complex I/O stack and, consequently, may not fully utilize the low latency of the NVMe SSDs and avoid cross-VM interference due to shared queues. To thoroughly reduce the I/O path complexity and to support performance isolation, the Direct Access Mode of the H-NVMe framework provides a user-space I/O framework to fully utilize the performance of NVMe SSDs, while meeting diverse requirements associated with different user applications.

[0033] As depicted in the rightmost portion of FIG. 2, in the direct access mode the H-NVMe framework bypasses the multiple queues in the original I/O path through the hypervisor layer 102, and provides handles to grant each application in the user layer 201 a root privilege to manage the access permission of I/O queues. In one embodiment, the H-NVMe framework attaches a VM Daemon (I0Filter) to each of the VMDKs for polling I/O requests, thereby allowing the threads of user applications to directly access NVMe driver queues.

[0034] Each user thread 271 may be assigned to an application 211 inside the respective VMs 210. A user-mode application 211 may use different I/O queues for different I/O sizes and may implement different polling strategies. The I/O request submissions and completions do not require any driver interventions. As shown at 281, a handle 282 will bypass all other I/O paths in the hypervisor layer 202, and will map the user space to the NVMe space.

[0035] Table 2 sets forth pseudocode for an example process of the direct access mode. FIG. 4 depicts the example direct access mode process set forth in Table 2 in a flowchart format with the line numbers in Table 2 indicated in the FIG. 4.

<table>
<thead>
<tr>
<th>Procedure I0SubmissionNVMeHypervisor( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>userLib \rightarrow kernelDriver ;</td>
</tr>
<tr>
<td>/* Initialization */ ;</td>
</tr>
<tr>
<td>if curApp\epsilon userLevel whitelist then</td>
</tr>
<tr>
<td>new subQueue;</td>
</tr>
<tr>
<td>new compQueue;</td>
</tr>
<tr>
<td>subQueue.memoryRegion.map(curApp.memRegion);</td>
</tr>
<tr>
<td>compQueue.memoryRegion.map(curApp.memRegion);</td>
</tr>
<tr>
<td>curHandle.new handle.submission Queue ;</td>
</tr>
<tr>
<td>completionQueue, doo rbell Reg ;</td>
</tr>
<tr>
<td>curHandleList.add(curHandle);</td>
</tr>
<tr>
<td>ForwardI0Cmd(curHandleList, NVMeCtrl);</td>
</tr>
</tbody>
</table>

[0036] A device driver queue 291 may be directly assigned to each handle 282 without an I/O request being forwarded by the world queue at the user layer 201 and the adapter queue at the hypervisor layer 202. An I0 server 292 handles the I/O requests directly received by a device driver 291 by sending an I/O request to the NVMe controller 234.

[0037] When an application 211 requests a single I/O queue in the D AM, the H-NVMe framework checks whether the application 211 is allowed to perform user-level I/O requests to the storage device layer 203. If the application is in a whitelist to perform user-level I/O, which may be preset based on the SLA and security audit requirements associated with the user, the H-NVMe framework creates a submission queue and a completion queue. (See, for example, lines 4-6 of the process in Table 2.) The H-NVMe framework also maps the memory regions of the submission queue and the completion queue including any associated doorbell registers to the user-space memory region for the application 211. (See, for example, lines 7-8 of the process of Table 2.) After the initialization process, an application 211 may issue I/O commands directly to the NVMe SSD without any hardware modification or help from the kernel I/O stack. (See, for example, lines 9-11 of the process or Table 2.)

[0038] As will be recognized by those skilled in the art, the innovative concepts described herein can be modified and varied over a wide range of applications. Accordingly, the scope of claimed subject matter should not be limited to any of the specific exemplary teachings discussed above, but is instead defined by the following claims.

What is claimed is:

1. A non-volatile memory (NVM) express (NVMe) system, comprising:

   a. at least one user application, each user application running in a respective virtual machine environment and including a user input/output (I/O) queue;

   b. an NVMe controller coupled to at least one NVM storage device, the NVMe controller comprising a driver that includes at least one device queue; and
a hypervisor coupled to the user I/O queue of each user application and to the NVMe controller, the hypervisor forcing each user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller.

2. The NVMe system of claim 1, wherein the hypervisor sends false completion messages to each user I/O queue to force the user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller.

3. The NVMe system of claim 2, wherein the NVMe controller selects an idle I/O server to send I/O traffic in the device queue to an NVM storage device.

4. The NVMe system of claim 2, wherein the hypervisor determines an optimum number of device queues based on an Erlang function.

5. The NVMe system of claim 4, wherein the hypervisor further selectively forces each user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller or enables a private I/O channel between the user I/O queue and a corresponding device queue in the driver of the NVMe controller.

6. The NVMe system of claim 5, wherein the private I/O channel enabled between the user I/O queue and the corresponding device queue is enabled for at least one thread of the corresponding user application.

7. The NVMe system of claim 6, wherein each private I/O channel includes a corresponding I/O server in the driver of the NVMe controller.

8. The NVMe system of claim 5, wherein hypervisor enables based on a service level agreement.

9. The NVMe system of claim 8, wherein the hypervisor enables the private I/O channel further based on a permission level of the user application.

10. A non-volatile memory (NVM) express (NVMe) system, comprising:

   at least one user application, each user application running in a respective virtual machine environment and including a user input/output (I/O) queue;

   an NVMe controller comprised to at least one NVMe storage device, the NVMe controller comprising a driver that includes at least one device queue; and

   a hypervisor coupled to the user I/O queue of each user application and to the NVMe controller, the hypervisor enabling a private I/O channel between the user I/O queue and a corresponding device queue in the driver of the NVMe controller.

11. The NVMe system of claim 10, wherein the private I/O channel enabled between the user I/O queue and the corresponding device queue is enabled for at least one thread of the corresponding user application.

12. The NVMe system of claim 11, wherein each private I/O channel includes a corresponding I/O server in the driver of the NVMe controller.

13. The NVMe system of claim 11, wherein hypervisor enables based on a service level agreement.

14. The NVMe system of claim 13, wherein the hypervisor enables the private I/O channel further based on a permission level of the user application.

15. The NVMe system of claim 11, wherein the hypervisor further selectively enables a private I/O channel between the user I/O queue and a corresponding device queue in the driver of the NVMe controller or forces each user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller.

16. The NVMe system of claim 15, wherein the hypervisor sends false completion messages to each user I/O queue to force the user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller.

17. The NVMe system of claim 16, wherein the NVMe controller selects an idle I/O server to send I/O traffic in the device queue to an NVM storage device.

18. The NVMe system of claim 17, wherein the hypervisor determines an optimum number of device queues based on an Erlang function.

19. A non-volatile memory (NVM) express (NVMe) system, comprising:

   at least one user application, each user application running in a respective virtual machine environment and including a user input/output (I/O) queue;

   an NVMe controller comprised to at least one NVMe storage device, the NVMe controller comprising a driver that includes at least one device queue; and

   a hypervisor coupled to the user I/O queue of each user application and to the NVMe controller, the hypervisor selectively forcing each user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller or enabling a private I/O channel between the user I/O queue and a corresponding device queue in the driver of the NVMe controller.

20. The NVMe system of claim 19, wherein the hypervisor selectively forces each user I/O queue to empty, the hypervisor sends false completion messages to each user I/O queue to force the user I/O queue to empty to a corresponding device queue in the driver of the NVMe controller, and

   wherein if the hypervisor selectively enables a private I/O channel between the user I/O queue and a corresponding device queue the private I/O channel, the hypervisor enables the private channel for at least one thread of the corresponding user application.

* * * * *