SEE TEXT ONLY

bhyve ATA Emulation

The bhyve ATA/ATAPI emulation is part of a larger project that aims to ensure backward compatibility with older versions of FreeBSD guests for Free BSD Hypervisor (bhyve). Currently the bhyve hypervisor emulates AHCI standard for drive and atapi devices. In order to support guests like FreeBSD 4 that do not have ahci drivers, it is necessary to emulate an ATA/ATAPI controller.

by Teaca Ionut-Alexandru,
Mihai Carabas, and
Peter Grehan

Introduction

In this article we present the emulation of a generic ATA drive controller connected on the PCI bus through a Host PCI Adapter. Both ATA controller and Host Adapter are parts of our implementation that is designed and developed from scratch. Using this emulator, we simulate an ATA disk controller that is used by the guest ATA driver from the bhyve virtual machine.

Currently bhyve supports any version of FreeBSD i386/amd64 since the FreeBSD 8.0 release. The standard AHCI mode has also been supported out of the box on FreeBSD since version 8.0. The scope of this article presents a solution to provide compatibility of guest operating systems with older versions such as FreeBSD4/5. Consequently, emulation of the ATA Host Adapter Standard in bhyve hypervisor is required to support guest operating systems that have drivers only for the ATA controllers.

We begin by presenting some of the related work in ATA emulation, the motivation for starting to write this driver from scratch. We continue with an overview of device emulation in bhyve hypervisor. Most of the devices are emulated in userspace in usr.sbin/bhyve. The other ones, such as the Programmable Interrupt Controllers (PIC) and the timers, are implemented into the kernel in vmm/io. There are two categories of devices emulated in bhyve, the ISA and PCI devices. Through the ISA devices we enumerate the UART controller and RTC (real-time clock) controller. One part of the PCI devices is represented by the virtio class containing block, net, and rng (random entropy from /dev/random) subclasses. Also the AHCI controller is a PCI device emulated in bhyve. Our implementation of ATA Host Adapter will be emulated as a PCI device through a register to the PCI controller. That is also emulated in the bhyve hypervisor.

We present some general information about bhyve and some technical standards such as PATA, SATA, AHCI, and PCI that are related to the ATA emulation.

The ATA (AT Attachment) defines the physical, electrical, transport, and command protocols for the internal attachment of storage devices to host systems [4]. There can be Parallel ATA (PATA) or Serial ATA (SATA) interface standards. Parallel ATA (PATA) is the legacy AT Attachment, being an interface standard for the connection of storage devices like hard disks, floppy drives, and optical disc drives in computers [7]. Serial ATA takes the place of the former legacy AT Attachment standard, offering many advantages over the older interface: reduced cable size and cost (seven conductors instead of 40 or 80), native hot swapping, faster data transfer through higher signaling rates, and more efficient transfer through an (optional) I/O queuing protocol [8].

The Advanced Host Controller Interface (AHCI) is a host adapter for the Serial ATA disk drive controller. This specification defines the functional behavior and software interface of the Advanced Host Controller Interface, which is a hardware device that is an interface communication between the software and Serial ATA devices. AHCI is a PCI class device that performs movement data between the system host memory and Serial ATA devices [1].

For the Parallel ATA protocol there is also a host adapter controller interface. The ATA/ATAPI Host Adapters Standard specifies the AT Attachment Interface between host systems and storage devices using Direct Memory Access protocol. The AT Attachment Interface can be used in any host system that has a PCI bus and storage devices connected to the processor [2].

Overview of the bhyve Hypervisor Structure

bhyve stands for BSD hypervisor and is a hypervisor/virtual machine manager introduced into the FreeBSD operating system. It is similar to Linux KVM in that it runs on the host OS and relies on modern CPU features such as Intel VT-x, Extended Page Tables, and VirtIO network and storage drivers.

bhyve is comprised of the vmm.ko loadable kernel module, the libvmmapi library, and the bhyve, bhyveload, and bhyvectl utilities. To use any of these utilities, the vmm.ko module must be loaded first. The bhyveload tool helps load a FreeBSD kernel from a disk image. For example, */usr/sbin/bhyveload -m 256 -d ./vm0.img vm0.*

It shows the FreeBSD loader screen and you should see the device /dev/vmm/vm0.

The bhyve tool is used to boot the VM with 2 vCPUs, the same 256M RAM and the tap0 network interface.

/usr/sbin/bhyve -c 2 -m 256 -A -H -P -s 0:0,hostbridge-s 1:0,virtio-net,tap0 -s 2:0, ahci-hd,./vm0.img -s 31,lpc -l com1,stdio vm0

After the VM has been shut down, its resources can be reclaimed with:

/usr/sbin/bhyvectl –destroy –vm=vm0

The bhyve process starts by initializing the pci controller in the init_pci function. This function begins the bus enumeration to find all PCI devices. It iterates through each bus, each slot, and each function to find if there is a PCI device. For example, if the input parameter of the bhyve program is "3:0,ahci- hd,./diskdev" (that means bnum = 0, snum = 3, fnum = 0), the PCI controller associates this combination to a custom device, and maps it with a pci_devemu structure that has a pointer to the callback which initializes its PCI device, in this case, the 'pci_ahci_hd_init' function. By calling this function, the ahci device is initialized by providing identification information (the vendor and device information and the class, subclass, and progif information) to the pci_devinst structure. The PCI emulator maps the (bus, slot, function) combination with the pci_devinst structure, and, in so doing, the identification information from the pci_devinst structure belongs to the device controller. Also, the pci_ahci_hd_init callback is allocating the BAR register with baridx = 5 and type = PCIBAR MEM32. The ahci driver from the guest operating system programs the Base Address Registers to inform the ahci device of its address mapping by writing configuration commands to the PCI controller.

Related Work

There are no other related implementations with the ATA Host Adapter Standard emulated in the bhyve hypervisor. There is an ATA controller emulator implemented in the GXemul framework that supports full-system computer architecture emulation. It is, however, almost impossible to port this software in the bhyve sources tree due to the fact that it uses a different application interface for communication with the rest of the system and because it is coded in the C++ language. But we could use this software for a better understanding of the ATA protocols implemented there, like PIO, DMA, or other information regarding the ATA commands.

There is an implementation of the Advanced Host Controller Interface standard emulated in bhyve. In order to understand the mechanisms used in the emulation of the AHC standard, documentation of this implementation needs to be done.

In order to catch the host commands that are sent to the AHCI controller, the pci_ahci module in bhyve registers two handlers, pci_ahci_write and pci_ahci_read, that are called whenever the host software tries to address the ahci device through the BAR registers. Basically these callbacks read/write the value from the address computed by the baridx = 5 register and an offset. The implementations of these callbacks are dependent on the interface controller (ahci/ata/atapi). They emulate by accessing the iovec array from the prdt and writing to the diskdev file descriptor. To complete, the command will call the ioctl system call for emulating an interrupt. The io requests are created through these callbacks. The blockif_req requests are processed by the blockif thread in the ata_ioreq_cb callback. The ata_ioreq_cb routine is completed through an ioctl system call to the vmm.

For the manipulation of io requests there are two routines, blockif_dequeue and blockif_enqueue. First there is an io request array of elements and free/inuse queues grouped in a blockif_ctxt structure. The blockif_dequeue routine is called from the blockif thread context and it tails the first element from the inuse queue and updates the free/inuse queues. The blockif_enqueue gets a free element, fills it with a blockif_req request, and updates the free/inuse queues accordingly.

The DMA Write/Read requests are handled by the ahci_handle_dma implementation that builds up the iovec based on the Physical Region Descriptor Table (prdt). These iovec requests are processed in blockif_proc that briefly writes the iovec array to the diskdev file descriptor using the pwritev system call. The DMA command finishes by triggering an interrupt.

Architecture

In this section we focus on the most important design concepts at the base of implementation. First, we explain the primary/secondary and master/slave relationships followed by the register descriptions involved in the emulator interface and finishing with the interrupt-based mechanism that is used in driver-emulator communication and is the reason why the ATA emulator approaches an event-driven architecture.

Master Drives and Slave Drives

Before presenting the architecture of the ATA controller, let's take a look at the primary/secondary and master/slave relationships and the way the ATA emulator will be configured.

Most motherboards have two IDE interfaces (primary and secondary), also known as channels. In both primary and secondary IDE channels, only ATA can be connected to, which means that IDE only supports ATA/ATAPI drives. Each interface could support two devices, for a total of up to four drives. The two drives have to decide for themselves how to share the same ATA channel. To accomplish this, one drive on each channel is designated as the "master" and the other drive (if present) is designated the "slave." So that leaves us with the following possibilities:

- Primary Master Drive
- Primary Slave Drive
- Secondary Master Drive
- Secondary Slave Drive

Each drive can be either ATA or ATAPI.

Our PCI ATA adapter is emulating for the ATA legacy driver located in `sys/dev/ata/ata-pci.c` under the FreeBSD code base. After some investigation of the ATA legacy driver, we have observed that it is supporting only one channel, meaning only two drives. For more details, please see the 'int_ata legacy(device_t dev)' function from the `sys/dev/ata/ata-pci`.c driver device.

With these in mind, we choose to configure the ATA emulator using these parameters:

-s N:M,ata-hd,X,./DISK_MASTER,./DISK_SLAVE

or

-s N:M,ata-hd,X,./DISK_MASTER

where N:M is the pci slot information, X is 0 or 1 (Primary or Secondary channel) followed by the name of the disks, the first one being the master drive, and the second the slave drive. There might be only one disk representing the master drive. In any case, the emulator implementation is supporting two channels, and holding different data structures for both of the channels, even though the guest legacy driver is using only one (Primary or Secondary).

PCI and I/O Register Descriptions

In order to emulate the ATA controller disk, two sets of registers should be implemented as the controller is going to be emulated through the PCI bus, and so the commands sent by the guest driver are provided using this interface, the PCI interface. The set of adapter registers represent the PCI space configuration which the guest driver reads after the enumeration phase to detect controller configuration details. See the Implementation section of this article for details on the configuration of these registers.

The ATA controller also contains a set of registers (ATA registers). Basically this set is configured by the guest driver to communicate with the drive controller. The implementation of the emulator should provide such a register interface in addition to the data structure necessary to maintain the controller states and to emulate the driver commands. The ATA controller registers are divided into two categories. The first represents the ATA Bus Master Registers. The bus master ATA function uses 16 bytes of I/O space. All bus master ATA I/O space registers can be accessed as byte, word, or double word quantities. The base address for these registers is PCI BAR 4 [3]. The second one represents ATA Channel Registers.

The Command Block registers are used for sending commands to the device or posting status from the device. These registers include the LBA High, LBA Mid, Device, Sector Count, Command, Status, Features, Error, and Data registers. The Control Block registers are used for device control and to post alternate status. These registers include the Device Control and Alternate Status registers [5]. For the primary channel, these registers are addressed by the BAR0 and BAR1 registers, and the secondary channel uses BAR2 and BAR3.

Interrupts

The design of the ATA emulator is based on an event-driven architecture, where the events represent the write operations on the emulator registers that are actually the guest driver commands. After these commands are interpreted and processed, the state of the emulator is updated accordingly. When the command is fully emulated, the guest driver is notified by asserting an interrupt. The emulator adapter is supporting two channels conforming to the primary and secondary channel address and have a separate IRQ for each channel. A parallel ATA controller is using the 14 and 15 IRQs. The emulator is reserving these IRQ numbers in the initialization phase and asserts an IRQ at each command completion.

Block Device Emulation

For better management of the disk operations like read/write calls, the ATA emulator uses the implementation for block device emulations in bhyve. An instance of the blockif_ctxt structure is associated with each disk drive. Basically the ATA emulator supports a maximum of two disk drives—the master and slave drives. Hence, each of them will be assigned one blockif_ctxt structure. In addition to the design reasons for deciding on this API, the block model has an extra thread for the read/write requests to be executed in its own context. The public API for read/write operations works by submitting block requests to the block device queue which are pulled and executed in the block context. We want to delay the execution of the IO requests in the block context as the virtual machine loop—where the cpu instructions run—is single thread. If the IO operations were executed in the same context as the virtual machine loop, then the whole system would get stuck.

LBA 28-bit Addressing

Logical Block Addressing defines the addressing of data on the disk device by the linear mapping of sectors. This means the blocks are located by an integer index, with the first block being LBA 0, the second LBA 1, and so on. The read/write commands, whether the data transfer protocol is either PIO or DMA, use 28 bits for addressing one sector. Hence the maximum size supported by the 28-bit addressing is 228 × 512 bytes = 128 GB, since the size of one logical sector is 512 bytes. In order to support LBA 28-bit addressing, the ATA standard uses the LBA group registers and the first 4 bits in the Device register with the following mapping: LBA Low=LBA(7:0), LBA Mid=LBA(15:8), LBA High=LBA(23:16), Device(3:0)=LBA(27:24).

Implementation

This section begins with the initialization phase of the ATA emulator and continues by presenting some software protocols like reset, PIO in/out, DMA, and how the master/slave device addressing is handled in the implementation. At the end we present the ATA commands that have been implemented till now.

Initialization

The initialization of the ATA adapter begins in the pci_ata_init function by opening the backing file as disk storage. The PIO and DMA commands that will be handled by the emulator will result in read and write calls to the backing image file descriptor. Each IDE controller appears as a device on the PCI bus. If the class code is 0x01 (Mass Storage Controller) and the subclass code is 0x1 (IDE), this device is an IDE Device. The PCI Adapter implements a subset of the PCI standard type configuration header register set. First, the PCIR_DEVICE and PCIR_VENDOR registers are set with 0x8211 and 0x1283 values—a Waldo ATA Controller. The PCI Class code must be configured by setting the Base-class, Sub-class Codes with the next values: 01h – Mass Storage and 01h – IDE. The Programming Interface Code should be correctly handled by setting the PCIP_STORAGE_IDE_MASTERDEV bit to indicate that the adapter can do bus master operation, and either the PCIP_STORAGE_IDE_MODEPRIM or PCIP_STORAGE_IDE_MODESEC bit to select the ATA channel. In the initialization function of

the ATA Host emulator, the BAR registers must be allocated. The IDE device only uses five BARs of the six. The PCI BAR0 and BAR2 represent the base addresses of primary and secondary channels. The PCI BAR1 and BAR3 represent the base addresses for the control register for primary and secondary channels. The PCI Base Address Register BAR4 is the base address of the ATA Bus Master I/O registers. The initialization is completed by reserving the proper IRQ numbers for each ATA channel.

Device Addressing Considerations

When a value is written to the registers of both devices, the host discriminates between the two by using the DEV bit in the device register. Data is transferred in parallel either to or from host memory to the device's buffer under the direction of commands previously transferred from the host. The device performs all operations necessary to properly write data to or read data from the media. When two devices are connected on the cable, commands are written in parallel to both devices, and for all except the EXECUTE DEVICE DIAGNOSTIC command, only the selected device executes the command. When the device control register is written, both devices respond to the write regardless of which device is selected. When the DEV bit is cleared to zero, Device 0 is selected. When the DEV bit is set to one, Device 1 is selected. When two devices are connected to the cable, one is set as Device 0 and the other as Device 1 [6].

Software Reset Protocol

At some point the guest driver is going to software reset the ATA controller. One point would be when the guest driver is looking for any signs of ATA/ATAPI present in the channel. To reset the controller, the driver writes the ATA_A_RESET bit in the ATA_CONTROL register. For emulation of this operation, the ATA emulator sets the ATA_E_ILI bit in the ATA_ERROR register and the ATA_S_READY bit in the ATA_STATUS register. For more details and a better understanding please take a look at the ata generic reset function in the `sys/dev/ata/ata-lowlevel.c` FreeBSD driver.

PIO Data-in Command Protocol

The PIO data-in protocol is used to transfer one or more blocks of data from the disk device to the host memory. It is closely related to the commands that use this protocol since they are responsible for preparing the PIO data buffer. The use of this protocol is transparent for the ATA driver since it needs to specify only the parameters for a specific PIO command. Hence there is a PIO data-in class containing commands that use this protocol. Such commands are: IDENTIFY DEVICE, READ MULTIPLE, READ SECTOR(S). After one of these PIO commands is issued by the ATA guest driver, the emulator reads the data from the block disk into the PIO buffer and interrupts the host, which means the data is ready to transfer. At that moment, the host starts to transfer data by polling the DATA register and getting 4 bytes at each read. When the buffer is empty, the PIO command is completed. The host gets interrupts for each transferred block which is 128 sectors by default. If the total count of sectors is not evenly divisible by the block count, the emulator interrupts after the last partial block has been transferred.

PIO Data-out Command Protocol

The PIO data-out protocol is used to transfer one or more blocks of data from the host memory to the disk device. Differing from the PIO data-in protocol where the data buffer is prepared by the emulator before the transfer starts, the host writes the data into the buffer. It is closely related to the commands that use this protocol as they are responsible for transfer parameters like the sector count and the offset in the block disk. There is a PIO data-out class containing commands like WRITE MULTIPLE, WRITE SECTOR(S) that uses this protocol. After one of these commands is issued, the emulator saves the transfer parameters (sector count and offset) and waits for the data from the host without asserting the interrupt. The ATA driver starts to transfer data by polling from the DATA register into the PIO buffer adding 4 bytes at each write. When the total number of sectors has been written on the block disk, the PIO command is completed. For every transferred block, the emulator writes the data on the block disk and interrupts the host. If the total count of sectors is not evenly divisible by the block count, the emulator interrupts after the last partial block has been transferred.

DMA Command Protocol

The DMA protocol is used to transfer one or more blocks of data either from the host memory to the disk device, or from the disk device to the host memory. It is closely related to the commands that use this protocol since they are responsible for the transfer parameters like the sector count and the offset in the block disk. There is a DMA class containing commands like READ DMA, WRITE DMA that uses this protocol. Before further explanation, we want to introduce the concept of DMA transaction. Basically the DMA transaction contains three parts: the READ/WRITE DMA command, the start, and eventually the stop of the DMA procedure. After one of these commands is issued, the host driver starts the DMA transaction by giving the address of the Physical Region Descriptor Table to the emulator and setting the Start/Stop Bus Master bit in the ATA Bus Master Command Register.

The emulator gets the physical address in the guest memory space and needs to translate it into the bhyve process memory space. To achieve this, the emulator uses the paddr_guest2host function which uses the guest address as a parameter and returns a pointer to the host memory space. The PRD Table contains several Physical Region Descriptors (PRDs) which describe areas of the guest memory that are involved in the data transfer. Each PRD entry has 8 bytes and specifies the location where the data will be transferred to/from. The first 4 bytes represent the address of a physical guest memory region; the next 2 bytes specify the number of bytes that are supposed to be transferred to/from that region. The physical address of the entry is translated into the bhyve address space in the same way as the PRDT address using the paddr_guest2host. If bit 7 of the last byte is set, that is the end of the table. At this moment the emulator iterates through the entries to prepare the block request that will be executed by the block instance. Basically a block request contains an array of iovec structures with each iovec structure indicating an entry in the PRD Table. After the block request has been prepared, it is executed in the context of the block instance, meaning a write or read operation on the file descriptor associated with the block. The guest driver is notified when the transfer is complete by asserting an interrupt as it has to initiate the last phase of the transaction by clearing the Start/Stop Bus Master bit in the ATA Bus Master Command Register. At that moment, the emulator verifies the state of the transaction and sets the status register indicating whether the transfer has completed successfully or not and eventually marks the transaction as completed.

Command Descriptions

In FreeBSD, ATA commands are implemented in the `sys/dev/ata/ata-lowlevel.c` driver. This driver is directly controlling our ATA controller emulator. For a better understanding of the command structure, we took a look at some functions that are related to ATA commands in the guest driver: ata_generic_command, ata_tf_write, ata_wait.

An ATA command starts by selecting the master/slave drive, waiting for the controller to clear the ATA_S_BUSY register. When it is ready to issue the command, the driver enables the interrupt by setting ATA_A_4BIT bit in the ATA_CONTROL register and continues with some write register operations such ATA_FEATURE, ATA_COUNT, ATA_SECTOR, ATA_CYL_LSB, ATA _CYL_MSB, ATA_DRIVE to configure the parameters of the command. Actually the ATA command is issued to the controller by writing code into the ATA COMMAND register. The emulator saves these parameters into its internal data structures and uses it to find the meaning of the command. After the command is fully emulated, an interrupt should be asserted to notify the guest driver.

The commands implemented in the ATA emulator meet the general feature set commands supported by the ATA 6 standard. We describe some of the implemented ATA commands below.

• **IDENTIFY DEVICE:** This command enables the host to receive parameter information from the device. The device sets the BSY bit to 1, prepares to transfer the 256 words of device identification data to the host, sets the DRQ and READY bits to 1, clears the BSY bit to zero, and asserts INTRQ. The host may then transfer the data by reading the data register using the PIO data-in protocol. The data is transferred using 128 successive word transfers. The order of bytes inside the word is different between the driver and emulator memories. For example,

to send the *FaK3 MoDeL IDA diSk* string to the host memory, the emulator prepares the *aF3KM DoLeI ADd Si k* string. Using this command, the host finds out the CHS parameters of the disk device like the number of cylinders, heads, and sectors, the model name, serial number, and firmware version, and also the capabilities supported. The capabilities supported by the ATA emulator are: both Multi Word DMA2 and the PIO4 data transfer protocol working with 28-bit LBA addressing, support for write-read verify, and flushcache.

• READ MULTIPLE: This command is issued by the ATA driver to read data using the PIO data-in protocol. It specifies the number of sectors to be transferred and the offset on the block disk using the LBA 28-bit addressing mode. The emulator reads data from the disk, prepares the PIO buffer, marks the PIO read command in progress and interrupts the host meaning the data is ready. After the host gets the interrupt, it starts to transfer the data.

• WRITE MULTIPLE: This command is issued by the ATA driver to write data using the PIO data-out protocol. It specifies the number of sectors to be transferred and the offset on the block disk using the LBA 28-bit addressing mode. The emulator saves the command parameters into the PIO setup, marks the PIO write command in progress, and waits for the host to transfer the data.

• READ AND WRITE DMA: These commands are issued by the ATA driver to read / write using the DMA data protocol and they indicate the first phase of the DMA protocol. The protocol specifies the number of sectors to be transferred and the offset on the block disk using the LBA 28-bit addressing mode. The emulator saves the command parameters into the DMA setup including the direction of the operation (read / write), and marks the DMA transaction as started. Afterwards the host prepares the PRD Table and activates the DMA transaction.

• FLUSH CACHE: This command is a non-data ATA command used by the host to ask the device to flush the write cache. Basically the emulator creates a flush request to the block device which flushes the file descriptor associated with the disk drive.

LISTING 1. ATA LEGACY DRIVER'S LOG

```
atapci0: (ITE IT8211F UDMA133 controller)
port 0x2020-0x2027,0x2028-0x202b,0x170-0x177,
0x376,0x2040-0x204f at device 3.0 on pci0
ata0: <ATA channel> at channel 0 on atapci0
ata1: <ATA channel> at channel 1 on atapci0
ada0 at ata0 bus 0 scbus0 target 0 lun 0
ada0: <BHYVE ATA IDE DISK 1.0> ATA-6 device
ada0: Serial Number 123456
ada0: 16.700MB/s transfers (WDMA2, PIO 65536bytes)
ada0: 8192MB
ada0: (16777216 512 byte sectors: 16H 63S/T 16644C)
ada0: Previously was known as ad0
ada1: at ata0 bus 0 scbus0 target 1 lun 0
ada1: <BHYVE ATA IDE DISK 1.0> ATA-6 device
ada1: Serial Number 123456
ada1: 16.700MB/s transfers (WDMA2, PIO 65536bytes)
ada1: 8192MB
ada1: (16777216 512 byte sectors: 16H 63S/T 16644C)
ada1: Previously was known as ad1
```

Scenarios and Results

Using *-s 3:0,ata-hd,0,./diskdev_ata,./diskdev_ata_slave* input for the bhyve application, the next output is printed by the guest ATA legacy driver (see Listing 1).

We observe that the ATA controller is successfully recognized by the atapci driver and the BAR addresses are properly allocated (BAR1 = 0x2020-0x2027, BAR2 = 0x2028-0x202b, BAR3 = 0x170-0x177, BAR4 = 0x376, BAR5 = 0x2040-0x204f). Also both ATA channels are handled by the ata0 and ata1 drivers. Because both disk drives have been specified in the input string, indicating the first one as master drive and the second one as slave drive, there are two instances of the ada driver—ada0 and ada1. Hence, at the Partitioning phase of the FreeBSD Installer there are two disks on which to install the FreeBSD virtual machine with both of them being supported (see Listing 2). The model implemented by the ATA emulator is an ATA-6 device which supports both PIO and WDMA2 data transfer protocols. It should be emphasized that the speed of 16.700MB/s is not the actual speed of the data transfer between the guest operating system and emulator, but it represents the speed required by the WDMA2 standard.

LISTING 2. FreeBSD INSTALLER—PARTITIONING

```
Select the disk on which to install FreeBSD.
vtbd0     654 MB Disk
ada0      8.0 GB ATA Hard Disk (BHYVE ATA IDE DISK)
ada18.0 GB ATA Hard Disk (BHYVE ATA IDE DISK)
```

LISTING 3. ATA TESTING

```
dd bs=$BLOCK_SIZE count=1
if=/dev/random of=tests/testX
dd bs=$BLOCK_SIZE count=1
if=tests/testX of=/dev/ada1 oseek=$MAX_SECTORS
reboot
dd bs=$BLOCK_SIZE count=1
if=/dev/ada1 of=out/testX iseek=$MAX_SECTORS
diff out/testX tests/testX
```

At the moment, the guest FreeBSD virtual machine can be installed on either ada0 or ada1 and played without any restrictions using the ATA emulator. The only limitation is the size of the disk, which is maximum 128 GB due to LBA 28-bit addressing.

Even though the full installation of the virtual machine on the ATA disk proves the correctness of the ATA emulator, a set of tests for a proper validation have been performed. Using a similar approach to Listing 3, the BLOCK_SIZE and MAX_SECTORS variables have been varied to cover the whole disk with different chunk sizes. All the tests were passed proving the correctness of the implementation.

As we said before, the maximum transfer rate specified by the ATA-6 standard using the DMA Multi-word 2 is 16.7MB/s. However, this value seems smaller than the actual speed of our emulator since it is for hardware devices. In order to get the transfer rates of the ATA emulator, we use the diskinfo tool running inside the virtual machine which has the device emulated. Using the 'diskinfo -t /dev/ada1' command, where /dev/ada1 is the ATA disk emulated by our implementation, we get the following results seen in Listing 4, indicating transfer rates of more than 100MB/s, which is more than enough. The ATA-6 has progressed with Ultra DMA giving data transfer rates up to 100MB/s, which is comparable with our emulator using WDMA2. Hence our implementation is compliant with ATA-6 requirements without a need to develop the Ultra DMA feature.

Even though the ACHI and ATA disk drive controllers are completely different from each other and do not use the same transfer data protocols, it would be interesting to measure the performance of the AHCI module and compare it with our ATA emulator. When we do the

same procedure using the diskinfo tool on a partition controlled by the AHCI driver, we get the following transfer rates displayed in Listing 5.

These results are expected since the AHCI standard uses the UDMA6 data transfer protocol which imposes transfer rates of 300MB/s for hardware devices with more than the 16.7MB/s transfers specified by the WDMA2 data protocol. But the main reason why the AHCI emulation has a better transfer rate is that it uses the LBA 48-bit addressing mode, which makes it possible to transfer more data sectors (65536) per DMA transaction with less overhead in the communication between the host driver and the disk emulator.

LISTING 4. TRANSFER RATES

```
outside: 102400 kbytes in 0.719681 s = 142285 kB/s
middle:  102400 kbytes in 0.796385 s = 128581 kB/s
inside:  102400 kbytes in 0.781721 s = 130993 kB/s
```

LISTING 5. AHCI TRANSFER RATES

```
outside: 102400 kbytes in 0.326701 s = 313436 kB/s
middle:  102400 kbytes in 0.339824 s = 301332 kB/s
inside:  102400 kbytes in 0.348496 s = 293834 kB/s
```

Conclusion and Further Work

In this article we presented a model of emulation for an ATA disk drive controller under the PCI attachment. We managed to integrate it with bhyve and implement the general feature set commands supported by the ATA-6 standard. The implementation was tested for validation and the performance was measured and compared against the AHCI emulation. There is another option as the ATA emulator could be a child of the PCI-ISA bus. The utility of ATA would be hugely improved in legacy operating systems given that the boot loaders would be able to use it since the IO ports and IRQs would be at fixed locations. Future work aims to achieve ATA emulation as a PCI-ISA attachment and merging it to the final code base. ●

ALEX TEACA graduated from the University Politehnica of Bucharest and is currently a second-year master's degree student focused on parallel and distributed computer systems. His departmental research project involved work on the ATA/ATAPI emulation in bhyve with Mihai Carabas.

MIHAI CARABAS is a PhD student at the University Politehnica of Bucharest studying virtualization. He has contributed to the FreeBSD Project and DragonFlyBSD virtualization code.

PETER GREHAN is a FreeBSD committer who has been using BSD-derived operating systems in some form since the days of DEC Ultrix. He co-developed and maintains the bhyve hypervisor with Neel Natu.

References

[1] J. Boyd. Serial ata advanced host controller interface, page 9. Intel Corporation, Hillsboro, Oregon. (2008)

[2] T. Goodfellow. Ata/atapi host adapters standard, page 15. Pacific Digital Corporation, Irvine, California. (2003)

[3] T. Goodfellow. Ata/atapi host adapters standard, page 11. Pacific Digital Corporation, Irvine, California. (2003)

[4] P. T. McLean. Information technology–at attachment with packet interface–6, page 16. American National Standards Institute, New York, New York. (2002)

[5] P. T. McLean. Information technology–at attachment with packet interface–6, page 63. American National Standards Institute, New York, New York. (2002)

[6] P. T. McLean. Information technology–at attachment with packet interface–6, pages 56–57. American National Standards Institute, New York, New York. (2002)

[7] Wikipedia. Parallel ata–wikipedia. http://en.wikipedia.org/wiki/Parallel ATA, 2015. (Online; accessed February 14, 2015)

[8] Wikipedia. Serial ata–wikipedia. http://en.wikipedia.org/wiki/Serial ATA, 2015. (Online; accessed February 14, 2015)