A Messaging Interface to Disks

Manuel Stocker, Mark Nevill, Simon Gerber

April 2011–July 2011

This lab project introduces an AHCI driver and associated ATA primitives to Barrelfish[1]. Interfacing disks is implemented in a library that communicates with a management service. To enable integration of multiple controllers offering access to ATA/ATAPI-based devices, Flounder modifications including a backend for AHCI are proposed. This project also provides a basic analysis of the driver's performance characteristics. To demonstrate usage, a simple testcase, a FAT filesystem implementation and a simple block device filesystem are introduced.

[1] `http://www.barrelfish.org`

Contents

1 Introduction

The Advanced Host Controller Interface (AHCI) is a standard by Intel that defines a common API for Serial ATA (SATA) and SAS host bus adapters. In order to provide backwards-compatibility, AHCI specifies modes for both legacy IDE emulation and a standardized AHCI interface.

AHCI only implements the transport aspect of the communication with devices. Commands are still transferred as specified in the AT Attachment (ATA)/AT Attachment Packet Interface (ATAPI) standards.

1.1 ATA/ATAPI/SATA

The ATA standard specifies an interface for connecting several types of storage devices, including devices with removable media. ATAPI provides an extension to allow ATA to transmit SCSI commands.

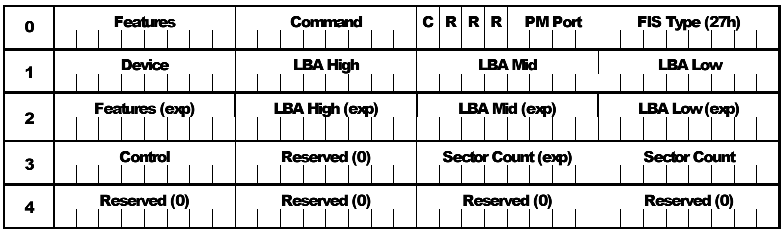
Commands that can be sent to ATA devices are specified in the ATA Command Set (ACS) specifications. Commands in particular interest for this lab project are the `IDENTIFY`, `READ DMA`, `WRITE DMA` and `FLUSH CACHE` commands.

The way these commands are sent to the device is specified in the respective specification, for example the SATA or Parallel ATA (PATA) specifications.

1.1.1 SATA

The SATA standard specifies the layout of the command Frame Information Structures (FISs) that encapsulate traditional ATA commands as well as all the lower layers of the interface to the disk, such as the physical layer.

Figure 1.1 shows the structure of an example FIS. A Host to Device Register FIS can be used to send commands to the disk. The command value is specified by ATA. The FIS contains additional values such as Logical Block Address (LBA) and sector count.



0	Features	Command	C R R R PM Port	FIS Type (27h)
1	Device	LBA High	LBA Mid	LBA Low
2	Features (exp)	LBA High (exp)	LBA Mid (exp)	LBA Low (exp)
3	Control	Reserved (0)	Sector Count (exp)	Sector Count
4	Reserved (0)	Reserved (0)	Reserved (0)	Reserved (0)

Figure 1.1: Host to Device Register FIS [3, p. 336]

1.2 AHCI

1.2.1 Memory Registers

While the PCI base address register 0-4 may contain pointers to address spaces for legacy IDE emulation, Base Address Register (BAR) 5 contains the address of the Host Bus Adapter (HBA)'s memory mapped registers. As shown in figure 1.2, this address space is divided into two areas: global registers for control of the HBA and registers for up to 32 ports. A port can be attached to either a device or a port multiplier. In this lab project, we focus on device handling and ignore port multipliers.

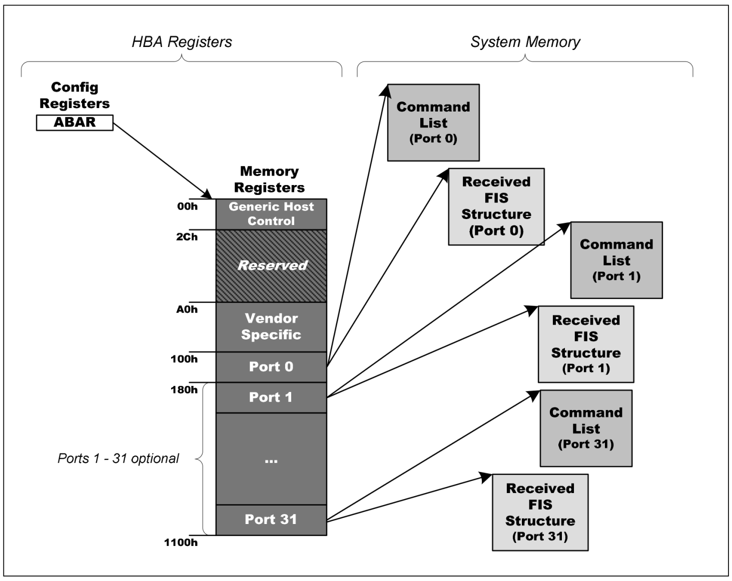


Figure 1.2: HBA Memory Space Usage [2, p. 33]

Every port area (Figure 1.3) contains further control registers and pointers to the memory regions for the command list and receive FIS area. Each of these pointers is a 64-bit value (32-bit for HBAs that don't support 64-bit addressing) stored in two port registers.

1.2.2 Received FIS Area

The received FIS area serves as an area where copies of the FISs received from the device are stored. The HBA will copy all incoming FISs to the appropriate region of the Received FIS (RFIS) area. If the HBA receives an unkown FIS it is copied to the Unknown FIS region if it is at most 64 bytes long. If the HBA receives an unknown FIS that is longer than 64 bytes, it will be considered illegal.

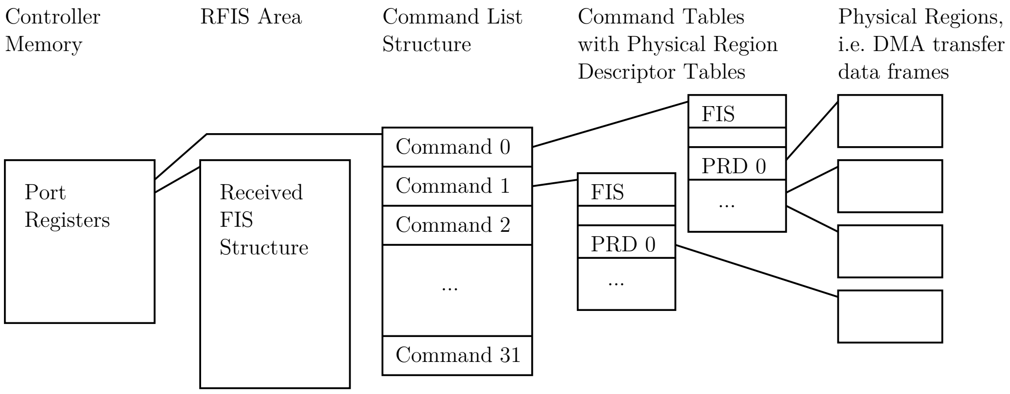


Figure 1.3: Port System Memory Structure adapted from [2, p. 34]

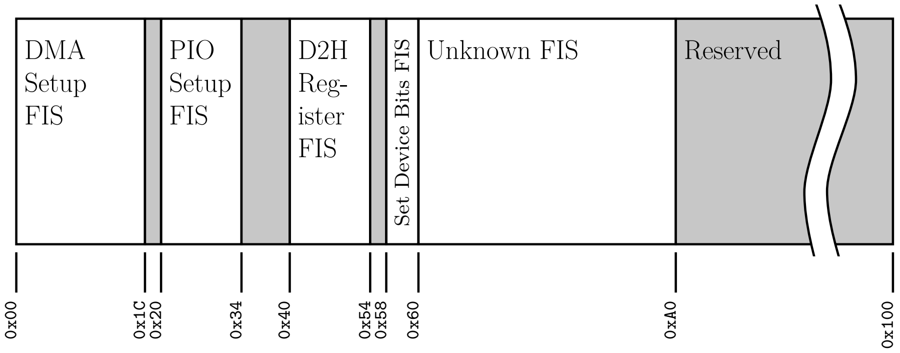


Figure 1.4: Received FIS Organization, adapted from [2, p. 35]

1.2.3 Commands

A command list (Figure 1.5) contains 32 command headers, which each contain the metadata for a single command.

Commands can be issued to the device by constructing a command header containing a reference to a command table and further metadata for the command to be issued.

The command table (Figure 1.6) contains the command FIS itself and an optional number of physical region descriptors specifying chunks of main memory in form of a scatter-gather list.

Commands are issued by setting the corresponding bit in the command issue register. Upon command completion, the bit is cleared and if enabled, an interrupt is triggered.

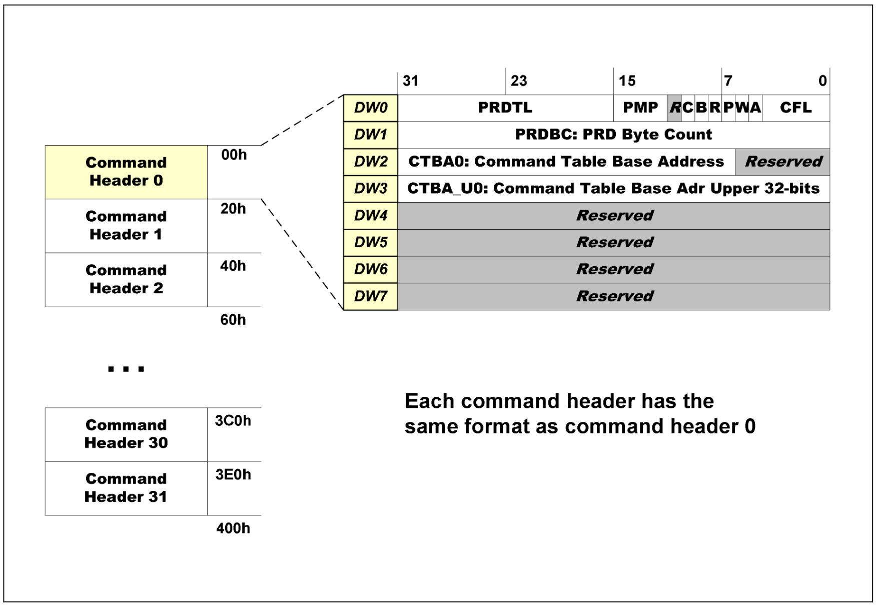


Figure 1.5: Command List Structure [2, p. 36]

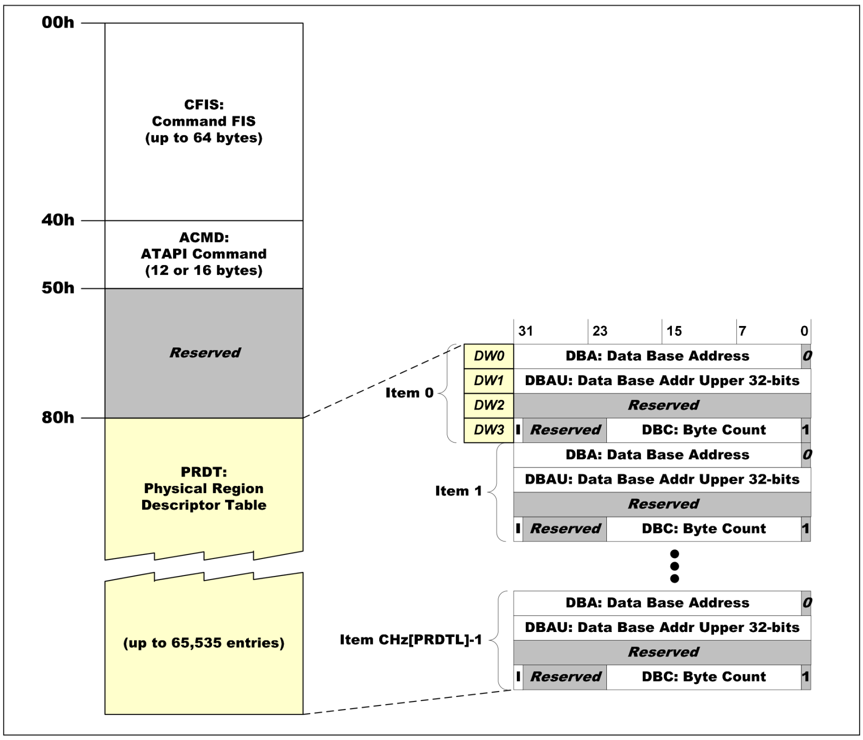


Figure 1.6: Command Table [2, p. 39]

2 Related Work

2.1 Other OSes

Most Unix-derived Operating Systems (Linux, BSD flavours, OpenSolaris, etc) integrate their AHCI subsystem into a larger disk subsystem with support for IDE disks, SATA disks (via AHCI), CDROM/DVD drives, and also floppy drives.

This larger disk subsystem often utilizes a general buffer layer which the OS kernel provides to its subsystems. Furthermore most Unix derivates – due to their essentially monolithic nature – couple the different layers of their disk subsystems (transport layer, message format and disk commands, e.g. AHCI, SATA and ATA respectively) using function pointers and most of them have in-kernel structures that describe commands that are issued to the disk in a message format and transport agnostic way. This makes those systems relatively easy to extend by adding a new layer implementations (e.g. when AHCI was first implemented a few years ago, it was as simple as providing a new transport layer implementation for disks attached to AHCI controller).

2.1.1 FreeBSD

FreeBSD employs the Common Access Method (CAM)[1] framework to seperate implementation of the driver for the I/O bus from the device driver for the attached device. Therefore, FreeBSD's AHCI driver is realized as a SCSI Interface Module (SIM) handling the I/O operations needed to get an ATA or SCSI command to the device (transparently using the packet interface of ATAPI). Other aspects of the storage system, such as filesystem or disk driver do not have to be modified.

2.1.2 Linux

Linux handles access to ATA devices with libATA[2] which unifies interfacing with SCSI and ATA based devices and adapters in a common API. libATA can translate SCSI commands to ATA and vice-versa or simulate a certain command if there is no translation possible. Drivers for adapters only need to implement hooks for basic device operations and communication.

[1] FreeBSD SCSI Documentation can be found under `http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/scsi-general.html`

[2] The libATA Developer's Guide can be found under `http://www.kernel.org/doc/htmldocs/libata.html`

3 Design

3.1 Design Options

In order to be able to register as a PCI device driver, some sort of management process for receiving the interrupts is necessary. A management process is also useful for device detection and initialization, providing the system with an overall view of what devices are available.

Consumers of AHCI-related interrupts must register with the management process so that interrupts may be forwarded. To provide clients with access to different SATA devices, it makes sense to grant access to invidual HBA ports, and similarly to forward all interrupts for a port to any clients registered to that port.

However, in choosing the method of accessing the ports, a trade-off must be made between security and performance. For example, by setting a suitable address, another domain's memory can be written to disk, and then read back, violating domain separation. To stop this from occuring, a central location must check that all Physical Region Descriptors (PRDs) reference memory which the client may access.

In such a design, all port memory access, including issuing of commands, would happen via Flounder messages to the central daemon. The central daemon would have to ensure that the client does not modify the command list or command tables after they are checked, so the all these areas would have to be copied into a memory area not accessible to the client.

To achieve optimal performance at the cost of security, each client must be given full access to the port memory. Because this is usually within the same page as the HBA memory, clients are able to access not just their own port's registers, but all other ports' and the HBA's registers as well. Also, as described above, the client can access all memory with suitable DMA commands.

3.2 General Architecture

As shown in in Figure 3.1, our message passing to disk has two main parts: a management part and a communication part. The management part, *ahcid*, provides a system-wide authority over an AHCI controller. The communication part, consisting of libahci (a low-level abstraction for using an AHCI port) and the ATA message specification and translation layer using Flounder, is used by all user-level code that wants to access a disk to send and receive messages from a disk.

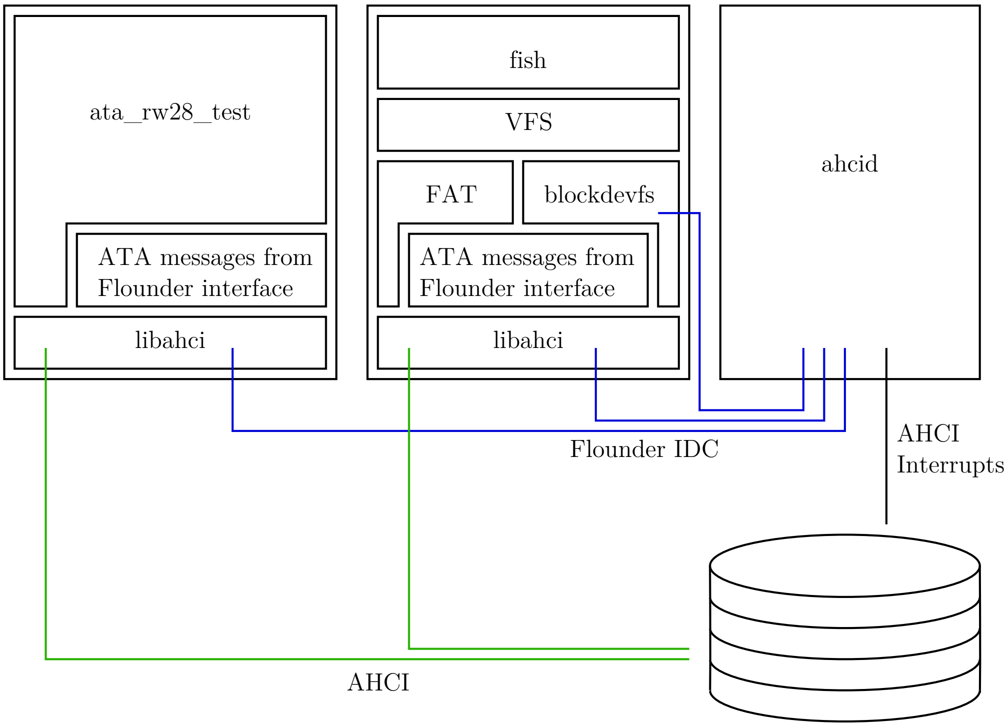


Figure 3.1: Barrelfish AHCI subsystem architecture

3.3 ahcid

ahcid exists as a central point of authority over an AHCI HBA. It is responsible for HBA initialization, Interrupt handling and access mediation.

3.3.1 Operation

ahcid ensures only one user can access an AHCI port at a time. Users can open a port by sending an IDC message to ahcid. If no other user currently owns the port, ahcid will provide a memory capability to the port's memory registers. The user is then able to use the port exclusively. Interrupts generated by the port are handled by ahcid and dispatched to the associated user via an IDC message. ahcid registers itself as PCI device driver for certain AHCI chipsets.

3.4 libahci

libahci tries to hide the necessary bit-twiddling and DMA buffer management for sending and receiving messages to a disk. Its interface resembles a Flounder-generated interface and makes use of Barrelfish's waitsets.

3.5 Flounder Backend

This lab project also contains a Flounder backend and Flounder modifications in order to be able to specify ATA commands as message definitions. The resulting interface can be used similarly to performing IDC.

3.6 Implemented ATA Commands

For our purposes (designing a messaging interface to disks) it was sufficient to implement commands for reading and writing blocks to and from disks using DMA (ATA commands `READ DMA`, `WRITE DMA`), inspecting the disk (`IDENTIFY`) and flushing the cache (`FLUSH CACHE`). Due to the Flounder layer in our system (specifically the AHCI backend) adding new ATA commands is quite easy: just add the command you want in the ATA interface specification.

4 ahcid

4.1 Introduction

4.1.1 Public IDC Interface

ahcid's design is modeled after netd. It has a small IDC interface that facilitates user access to a port: when registering for a port, the user is given the capability for the port registers. Interrupts are forwarded via IDC messages. Currently the interface also provides access to the `IDENTIFY` data of all available disks. This is useful to determine the type of device and total disk space without having to open the port.

```
interface ahci_mgmt "AHCI Management Daemon" {

    rpc list(out uint8 port_ids[len]);
    rpc identify(in uint8 port_id,
                 out uint8 identify_data[data_len]);

    rpc open(in uint8 port_id, out errval status,
             out cap controller_mem, out
             uint64 offset, out uint32 capabilities);
    rpc close(in uint8 port_id, out errval status);

    message command_completed(uint8 port_id,
                              uint32 interrupt_status);
};
```

Listing 4.1: ahci management Flounder interface

4.2 Initialization

ahcid registers itself as a driver for the AHCI device class. Once the init procedure is called, ahcid consults the received base address registers to find the memory region used for the HBA's registers. Although the AHCI specification requires the AHCI memory region to be in BAR 5 (BARs 0 to 4 are used for legacy IDE mode) the QEMU AHCI emulation incorrectly uses BAR 0. Because of this, ahcid consults both BAR 0 and BAR 5 to find the HBA's memory mapped I/O region.

As a first step, the HBA is reset in order to get to a known state. The HBA is also put into AHCI mode. After the initial reset, ahcid discovers the number of ports and detects which of them are implemented and have a disk connected. Discovered disks are assigned a system-wide unique port id and are registered with the skb. For every disk, an ATA `IDENTIFY` command is sent to determine the disk's parameters. A copy of the `IDENTIFY` response is cached in ahcid for later use.

After all attached disks are initialized, ahcid exports the ahci management interface, which clients can then use to register themselves for a single port.

4.3 Interrupt Handling

ahcid registers itself as an interrupt handler for the AHCI HBA controller when calling `pci_register_driver_irq`. The interrupt handler extracts the current interrupt state of the controller from the device memory and decides if the interrupt was triggered by the HBA. If the interrupt was triggered by the HBA, the handler loops over all ports and checks which ports received an interrupt and clears the port's interrupt register. The HBA's interrupt register is cleared after all port interrupt registers have been cleared. At last, if a client is registered for a port that has received an interrupt, ahcid sends a `command_completed` message (Listing 4.1) using the established `ahci_mgmt` interface. Clearing the interrupts before delivering the completion messages to the respective users ensures that we do not miss interrupts that would be triggered as a consequence of any commands issued in the command completion handler. Missing any interrupts for further completions could deadlock a user since we do not poll the port's state if no interrupts have been triggered.

5 libahci

5.1 Introduction

5.1.1 Purpose

The intent behind `libahci` is to provide an easy-to-use low-level interface to a single AHCI port. The main reason why such a library is desirable is to be able to send arbitrary ATA commands via AHCI without having to bother with the AHCI specification details.

5.1.2 Design

`libahci` abstracts the low-level AHCI operations such as the writing to memory mapped control registers of the HBA. It exposes an interface similar to that of Flounder-generated interfaces to offer a familiar environment for Barrelfish developers. The library is also used for the AHCI specific layer of the Flounder AHCI backend. It acts as a central point for interfacing AHCI controllers.

Apart from handling the sending of AHCI formatted ATA messages, `libahci` also provides memory management for DMA regions.

5.2 DMA Buffer Pool

As all data transfers with AHCI as transport are done via DMA, we need a mechanism to manage data buffers that are mapped non-cached. Because Barrelfish does not have memory reclamation for raw frame allocation, we must manage these buffers ourselves and have therefore implemented our own memory subsystem in the form of a DMA buffer pool, which allows for DMA buffer allocation and freeing.

The user has to call `ahci_dma_pool_init` to initialize the DMA buffer pool. After that, calls to `ahci_dma_region_alloc` and `ahci_dma_region_alloc_aligned` allocate buffers of the given size rounded up to 512 bytes, and the latter aligns the base address such that `base % alignment_requirement == 0`. `ahci_dma_region_free` returns the region it gets passed to the pool.

Additionally the buffer pool provides helper functions that facilitate copying data in and out of a buffer (`ahci_dma_region_copy_in` and `ahci_dma_region_copy_out`).

```
struct ahci_dma_region {
    void *vaddr;
    genpaddr_t paddr;
    size_t size;
    size_t backing_region;
};
```

Listing 5.1: DMA region handle

5.2.1 Design

The pool memory is organized in regions which are allocated and mapped using `frame_alloc` and `vspace_map_one_frame` respectively. The virtual and physical addresses of each of these regions are stored in the fields `vaddr` and `paddr` of `struct dma_pool` (c.f. Figure 5.1). The DMA buffer pool uses a doubly linked free list for maintaining the free chunks of the memory belonging to the pool. A pointer to the first free chunk of each backing region of the pool is stored in the pool metadata. Additionally pointers to the first and last free chunk are stored.

When processing an allocation request, the free list is scanned from the front for a sufficiently free chunk (first-free policy), which is returned in its entirety if it is at most 512 bytes larger than the requested size or split otherwise. If the chunk is split, the request is taken from the end of the chunk and the beginning of the block is left in the free list. If the entire chunk is returned, it is removed from the free list and the appropriate metadata pointers (`first_free`, `last_free`, and `pool.first_free[backing_region]`) are updated, if necessary.

If there is no block large enough to satisfy the allocation request, the pool is grown. This is done in steps of 8 megabytes at a time. Growing the pool involves resizing the metadata arrays (`virt_addrs`, `phys_addrs`, and `first_free`) and allocating and mapping memory for the new backing region.

Returning a block to the pool is similar: using the info in `pool.first_free`, a suitable point in the free list is found, and the block is inserted into the free list.

5.2.2 Implementation

`ahci_dma_region_alloc` searches through the free list linearly and stops at the first free chunk that meets the condition `request_size <= chunk_size`. If no free chunk meets that condition `grow_dma_pool` is called to increase the pool size by eight megabytes and the free list traversal continues with the new memory regions. When a sufficiently large free chunk is found, `get_region` is called. That function checks if the free chunk will be split or not (a chunk is split if the remaining free chunk will be at least 512 bytes), allocates and constructs a `struct ahci_dma_region` for the buffer that will be returned, including computing the virtual and physical addresses of the buffer, and shrinks the free chunk or removes it from the free list (according to the chunk-splitting decision).

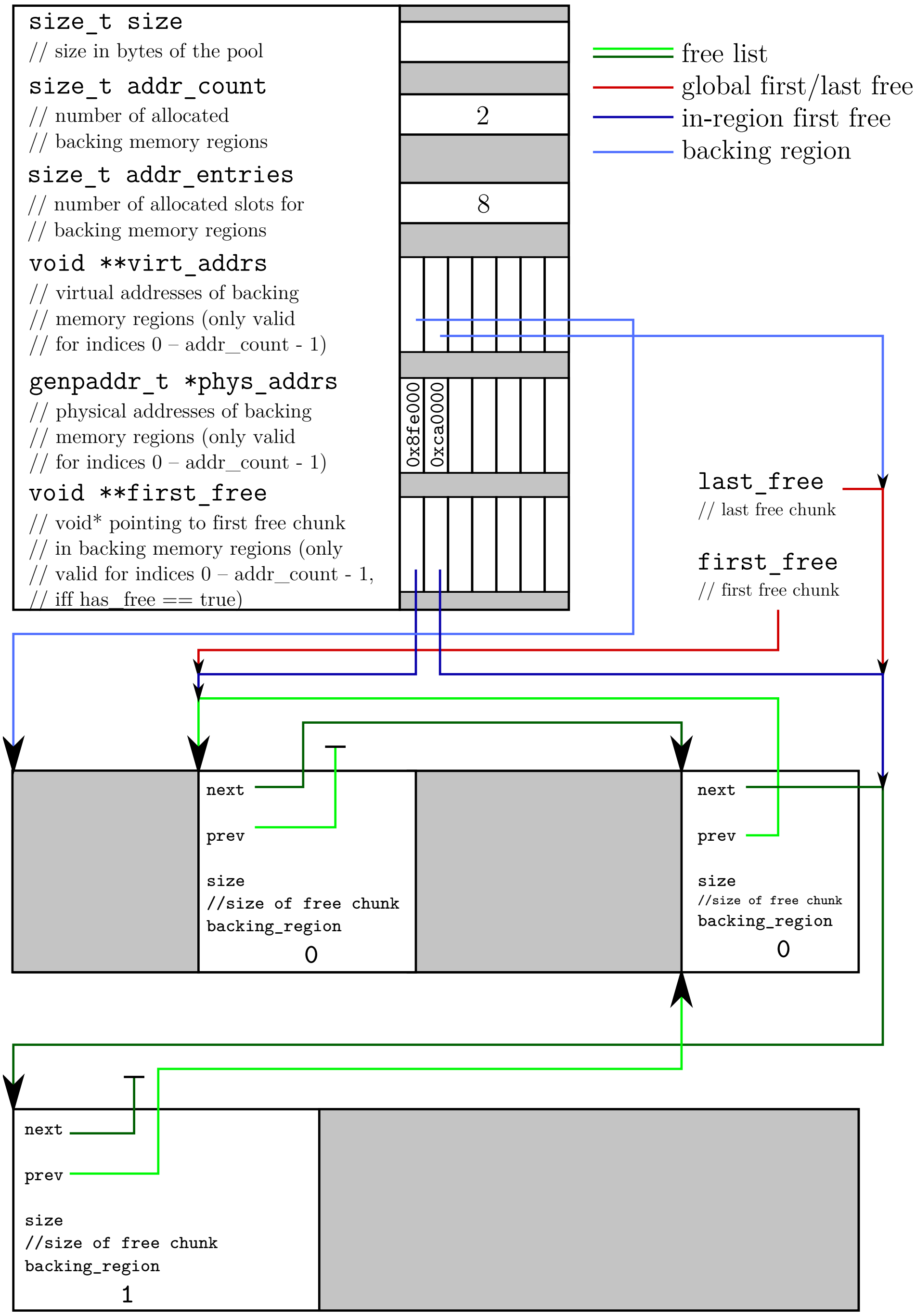


Figure 5.1: DMA Buffer Pool Design

ahci_dma_pool_init calls grow_dma_pool with the requested initial pool size rounded up to BASE_PAGE_SIZE.

ahci_dma_region_free calls return_region on the passed struct ahci_dma_region. That function inserts the region into the free list. Inserting the region into the free list can take different forms according to the state of the free list before inserting the chunk.

After inserting the newly freed chunk into the free list, return_region tries to merge the chunk with its predecessor and successor in order to prevent excessive fragmentation of the buffer pool memory. After calling return_region, the struct ahci_dma_region is freed.

The last two functions (ahci_dma_region_copy_in and ahci_dma_region_copy_out) are implemented as static inline and take a struct ahci_dma_region, a void* data buffer, a genvaddr_t offset (into the DMA region), and a size_t size. These functions just calculate the source (for ahci_dma_region_copy_out) or destination (for ahci_dma_region_copy_in) pointer for the memcpy and then copy the data.

5.3 libahci Interface

5.3.1 ahci_issue_command

ahci_issue_command is the main function of libahci and takes a void* tag with which the user can later match the command completed messages to his issued commands, a FIS and FIS length, a boolean flag is_write which indicates if DMA takes place to or from the disk, and a struct vregion* data buffer and associated length.

First off ahci_issue_command calls ahci_setup_command which allocates a command slot in the port's command header. After that, ahci_setup_command allocates a command table for the new command that has enough entries to accomodate $\lceil data\_length/prd\_size \rceil$ PRDs. Then ahci_setup_command inserts the newly allocated command table into the reserved slot in the port's command header and sets the bit to indicate the DMA direction (according to is_write) and also sets the FIS length in the command header slot. Finally, the FIS is copied into the newly allocated command table and the int *command output parameter is assigned the command slot number of the new command.

After completion of ahci_setup_command, ahci_issue_command saves the user's tag into the command slot metadata and proceeds to call ahci_add_physical_regions. This function takes the command slot number (int commmand) and a data buffer, partitions the data buffer into physical regions and inserts those regions into the command slot indicated by command. The size of the physical regions is specified as at most 4MB and must be an even byte count. However, due to hardware-related problems when using physical regions larger than 128kB we artificially cap the physical region size at 128kB. Memory addresses have to be word aligned. If a constant and predictable physical region size is desired, one can define AHCI_FIXED_PR_SIZE and PR_SIZE to enforce a specific size for physical regions.

Finally ahci_issue_command sets the issue command bit for the command slot in which the new command is stored and calls the user continuation, if any.

5.3.2 Command Completed Callback

The command completed callback is called when the AHCI management daemon receives a interrupt targeted to the AHCI port which is coupled with the associated `struct ahci_binding`. The command completed callback can be adjusted by user code in order to post-process (cleanup, copy-out of read data, etc.) a completed AHCI command.

The management command completed callback in `libahci` (which is called from ahcid when the port associated with the current libahci binding receives an interrupt) reads the commmand issue register of the port and calls the user-supplied command completed callback for each command slot which is marked `in_use` in libahci but which has the corresponding bit in the command issue register cleared.

The user-supplied command completed callback takes a `void *tag` as its only argument; these tags are also saved in libahci, and should uniquely identify their correpsonding AHCI command.

5.3.3 `ahci_init`

`ahci_init` is the first function a user of `libahci` calls. `ahci_init` initializes the `struct ahci_binding` for the connection and if the connection to *ahcid* has not yet been established, tries to bind to *ahcid*. The initalization of `libahci` continues when the bind callback that was specified in the call to *ahcid* executes.

On the first call to `ahci_init`, the bind callback sets up the function table for the management binding and then calls `ahci_mgmt_open_call__tx` to request the port specified by the `uint8_t port` parameter of `ahci_init` from *ahcid*. The initialization finishes when the ahci management open callback executes.

On later `ahci_init` calls `ahci_init` updates the *ahcid* binding to know about the new `libahci` connection and directly calls `ahci_mgmt_open_call__tx`.

The open callback checks if the open call succeeded, and if so, the memory region containing the registers belonging to the requested port is mapped in the address space in which `libahci` executes. After that the receive FIS area and the command list are set up, a copy of the `IDENTIFY` data is fetched from *ahcid*, the port is enabled (the *command list running* flag is set to one) and all port interrupts are enabled.

5.3.4 `ahci_close`

The purpose of `ahci_close` is to release the port by calling the close function of *ahcid* (c.f. Listing 4.1). This needs to be done, as otherwise *ahcid* will return `AHCI_ERR_PORT_BUSY` on subsequent open calls for the same port.

5.3.5 `sata_fis.h`

This header contains definitions dealing with SATA's FIS that are used for sending commands over AHCI. While the ATA command specification defines what registers exist for each FIS type and how they are used, the SATA specification defines the binary layout of these registers.

While it might initially seem that a mackerel specification for these structures would be sufficient, complexity introduced through optional ATA features makes a custom API preferable. As an example, consider the layout of 28-bit and 48-bit LBAs: for 28 bit LBAs, the lower 24 bits are placed in registers lba0 through lba2, while the upper 4 bits are placed in the low bits of the device register. However, for 48-bit LBA, the device register is not used, and the upper 24 bits are placed in register lba3 through lba5, which are separate from the lower 3 lba registers.

5.4 Error Handling

A mandatory part of an AHCI driver is to check if the HBA signals any errors on command completion. libahci does check the relevant registers, but the only error handling implemented right now is to dump the registers specifying the error and then aborting the domain that received the error.

In order to comply to the AHCI specification, the software stack (i.e. libahci) should attempt to recover. Errors signaled by one of the HBFS, HBDS, IFS or TFES interrupts are fatal and will cause the HBA to stop processing commmands. To recover from a fatal error, the port needs to be restarted and any pending commands have to be re-issued to the hardware or user level code has to be notified that these commands failed.

Errors signaled by the INFS or OFS interrupts are not fatal and the HBA continues processing commands. In this case the software stack does not have to take any action.

6 Flounder AHCI Backend

6.1 Introduction

6.1.1 Purpose

The goal of the AHCI Flounder backend is twofold: first, it should allow specifying ATA messages declaratively, making adding messages easier and reducing the amount of code potentially containing bugs. Second, sending such ATA messages to a disk should behave just like general-purpose inter-dispatch messaging, enabling transparent proxying of messages should no direct connection be available between a dispatcher and a suitable I/O controller.

6.1.2 Design

However, our use of Flounder also represents a major extension to its purpose. So far, Flounder transports have simply been responsible for transferring data, i.e. marshalling, packaging and transmission. Our backend differs in that it must understand the purpose of each data item: depending on that purpose, it must be formatted in a particular way, other actions may be necessary, and certain restrictions (in particular on the size and type of the data) may apply.

What this means for our project is that we must extend Flounder's syntax with message meta-data, i.e. parameters providing additional information about a message definition but not contributing to the runtime message payload. An example of our current syntax can be seen in figure 6.1.

6.2 Discussion

6.2.1 Targeting: Compiler vs. Topic

As seen in the syntax example, a set of meta-parameters is targeted using the "@" notation. Two possibilities exist for the interpretation of the target specifier:

- The target may specify a compiler name (e.g. `@AHCI_Stubs`), with each compiler receiving only the meta-parameters targeted to it. Among other things, this requires an additional step between parsing and compiling, thus a compiler no longer receives the interface definition's full AST.

  Also, compilers are unable to share such parameters, e.g. if backends exist for different ATA command transports, parameters related to formatting of ATA commands must be repeated for each backend's compiler.

```
interface ata_rw28 "ATA read & write with 28-bit LBA" {

    @ata(command=0xC8, dma_arg=buffer, dma_size=read_size,
         lba=start_lba)
    rpc read_dma(in uint32 read_size, in uint32 start_lba,
                 out uint8 buffer[buffer_size]);

    @ata(command=0xC8, dma_arg=buffer, dma_size=512, lba=lba)
    rpc read_dma_block(in uint32 lba, out uint8 buffer[buffer_size]);

    @ata(command=0xCA, dma_arg=buffer, is_write=1, lba=lba)
    rpc write_dma(in uint8 buffer[buffer_size], in uint32 lba,
                  out errval status);

    @ata(command=0xEC, dma_arg=buffer, dma_size=512)
    rpc identify_device(out uint8 buffer[buffer_size]);

    @ata(command=0xE7)
    rpc flush_cache(out errval status);
};
```

Figure 6.1: Example ATA message definitions

- The target may specify a generic “topic”. This does not require the extra preprocessing step and allows sharing of meta-parameters, but requires compilers to match their interpretation of shared parameters. Nonetheless, the sharing of parameters (also between header and stub compilers of the same backend) may make this solution preferable.

While the initial implementation used the former solution, this was replaced and we now use the second option.

6.2.2 Parameter Analysis

Extracting information from parameters and meta-parameters can be done in various ways. Their presence, absence, type and (for meta-parameters) value can all be used as sources of information. The question is therefore how best to handle this information, as demonstrated in the following examples:

- The presence of an output parameter `status` of type `errval_t` may imply that it should be used for a status result. But what if the type is different, or there is a `errval_t`-typed parameter with a different name?

- The size of a buffer may be extracted from its type if that is an array typedef. If it is a dynamic array, the may be the dynamic length parameter. In either case, the size might also be specified as a meta-parameter. Which of these information sources should be accepted?

6.3 Generated Interface

6.3.1 Initialization

Initialization is done with *if_name*`_ahci_init`. The client must first initialize libahci, at which point the target device is specified with the `port` parameter of `ahci_init`. Also, the client must allocate a suitable *if_name*`_ahci_binding`.

6.3.2 Binding Type

The *if_name*`_ahci_binding` type extends the generic binding type, allowing the generated AHCI bindings to be used anywhere the generic binding type is used. In particular, the RPCClient can be wrapped around AHCI bindings, greatly simplifying their usage.

Additionally, an AHCI binding contains a libahci binding, used internally for communication with the library.

6.3.3 Interface methods

Because AHCI Flounder bindings use the generic binding as a base, the generic messaging methods can be used, with the *if_name*`_ahci_binding` cast to the generic *if_name*`_binding`.

6.4 Implementation

6.4.1 Command Completion

To generate Flounder responses, the AHCI backend must associate command completion callbacks from libahci with information from the original command. This is done using libahci's command tags; before issuing a command, a `completed_rx_st` "command completion" struct is allocated and filled. The address of this struct is sent as the command tag for `issue_command`. Upon command completion, the tag is cast to a `completed_rx_st`, and the `completed_fn` callback function pointer is called.

The `completed_rx_st` contains information for the message-specific completion handler. Currently, this consists of the *if_name*`_ahci_binding` and the region used for DMA. If the message is supposed to perform a DMA read, the data must be copied out of the DMA region into an allocated buffer to be passed to the client. If DMA is used at all, the region must also be freed.

Finally, the `completed_rx_st` is used by the `issue_command_cb` for freeing the allocated FIS and calling the message send user continuation.

6.4.2 DMA Handling

When parameter analysis indicates a DMA transfer is expected, the AHCI backend must generate code to setup DMA regions, copying in TX data before issuing the command, and copying out RX data after the command completes.

To perform DMA, a `dma_arg` must be specified in the `ata`-targeted message meta-arguments. When the AHCI backend detects this argument, it expects the value to be an identifier that corresponds to one of the formal message arguments. The DMA direction is then the direction of that message argument; `in` is a transmit, `out` is a receive.

Because the DMA region must be allocated before the command is issued, if a DMA argument is present, the size of the DMA must either be specified in the interface file, or must be determinable upon receiving the rpc call from the client. The size of the DMA may therefore be specified with any of the following means:

- If `dma_arg` is a dynamic array, its size argument is used. (transmit only)
- If `dma_arg`'s type is an array of fixed size, that is used.
- If a meta-argument `dma_size` is present and is an integer, that is used.
- If a meta-argument `dma_size` is present and is an identifier, the RPC must have an `in` argument with that name, the value of which is used.

Finally, the DMA data must be copied in and out of suitably mapped regions, managed using libahci's `ahci_dma_region` API. This is necessary because flounder semantics require that the client owns buffer memory.

6.4.3 FIS Setup

To issue a command over AHCI, the AHCI backend must first set up a suitable FIS. This is done using the `sata_fis` API in libahci.

7 Driver Usage Example

This lab project contains a new testcase ata_rw28_test to test the Flounder-generated interface for ATA in LBA28 addressing mode. This chapter walks through its code to demonstrate the steps needed to access disks using the Flounder backend.

The application first initializes the necessary bindings and RPC client. It then uses the RPC wrapper around the Flounder-based ATA interface geared towards LBA28 addressing mode. The test itself is performed by writing 0xdeadbeef in multiple 512 byte blocks and verifying that the data is actually written to disk by reading it back and checking the contents. The test concludes with releasing the port.

7.1 Datastructures

To be able to perform RPC calls to read from or write to the disk, an ahci_binding as well as an ahci_ata_rw28_binding and an ata_rw28_rpc_client are necessary. ata_rw28_test defines these as global variables out of convenience:

```
struct ahci_ata_rw28_binding ahci_ata_rw28_binding;
struct ata_rw28_rpc_client ata_rw28_rpc;
struct ata_rw28_binding *ata_rw28_binding = NULL;
struct ahci_binding *ahci_binding = NULL;
```

The required header files are:

```
#include <barrelfish/barrelfish.h>
#include <barrelfish/waitset.h>
#include <if/ata_rw28_defs.h>
#include <if/ata_rw28_ahci_defs.h>
#include <if/ata_rw28_rpcclient_defs.h>
```

7.2 Initialization

First, we need to initialize the DMA pool which is used to manage frames that are mapped uncached and are therefore suitable for DMA transfers. We initialize the pool to be 1MB in size:

```
ahci_dma_pool_init(1024*1024);
```

Next, we need to initialize libahci and specify which AHCI port we want to use. For simplicity, we use port 0 which is the first device detected. To achieve blocking behaviour, we enter a spinloop and wait for the callback from ahcid:

```
err = ahci_init(0, ahci_bind_cb, NULL, get_default_waitset());
if (err_is_fail(err) ||
    err_is_fail(err=wait_bind((void**)&ahci_binding))) {
    USER_PANIC_ERR(err, "ahci_init");
}
```

The callback `ahci_bind_cb` simply sets the global `ahci_binding` and `wait_bind` waits for this global to be set:

```
static void ahci_bind_cb(void *st,
    errval_t err, struct ahci_binding *_binding)
{
    bind_err = err;
    if (err_is_ok(err)) {
        ahci_binding = _binding;
    }
}

static errval_t wait_bind(void **bind_p)
{
    while (!*bind_p && err_is_ok(bind_err)) {
        messages_wait_and_handle_next();
    }
    return bind_err;
}
```

The RPC client can be constructed by first initializing the `ata_rw28` binding and then building an RPC client on top of it. The pointer to the binding is stored for convenience as it is used frequently:

```
err = ahci_ata_rw28_init(&ahci_ata_rw28_binding, get_default_waitset(),
    ahci_binding);
if (err_is_fail(err)) {
    USER_PANIC_ERR(err, "ahci_ata_rw28_init");
}

ata_rw28_binding = (struct ata_rw28_binding*)&ahci_ata_rw28_binding;

err = ata_rw28_rpc_client_init(&ata_rw28_rpc, ata_rw28_binding);
if (err_is_fail(err)) {
    USER_PANIC_ERR(err, "ata_rw28_rpc_client_init");
}
```

RPC calls can now be made to perform operations on the disk.

7.3 Data Manipulation

write_and_check_32 is the function used to write 0xdeadbeef to the disk and verify that writing succeeded. It accepts arbitrary 32 bit patterns that are written to disk. First off, we need to calculate some values, allocate a buffer and fill this buffer with the pattern:

```
static void write_and_check_32(uint32_t pat, size_t start_lba,
    size_t block_size, size_t block_count)
{
    errval_t err;
    size_t bytes = block_size*block_count;
    uint8_t *buf = malloc(bytes);
    assert(buf);
    size_t step = sizeof(pat);
    size_t count = bytes / step;
    assert(bytes % sizeof(pat) == 0);
    for (size_t i = 0; i < count; ++i)
        *(uint32_t*)(buf+i*step) = pat;
```

The actual writing is very simple. We issue the write_dma RPC call, pass it the binding, the buffer, the number of bytes to write, the LBA on the disk where we want to write to and do some basic error handling:

```
    printf("writing data\n");
    errval_t status;
    err = ata_rw28_rpc.vtbl.write_dma(&ata_rw28_rpc, buf, bytes,
        start_lba, &status);
    if (err_is_fail(err))
        USER_PANIC_ERR(err, "write_dma rpc");
    if (err_is_fail(status))
        USER_PANIC_ERR(status, "write_dma status");
```

Reading data is equally simple:

```
    size_t bytes_read;
    err = ata_rw28_rpc.vtbl.read_dma(&ata_rw28_rpc, bytes,
        start_lba, &buf, &bytes_read);
    if (err_is_fail(err))
        USER_PANIC_ERR(err, "read_dma rpc");
    if (!buf)
        USER_PANIC("read_dma -> !buf");
    if (bytes_read != bytes)
        USER_PANIC("read_dma -> bytes_read != bytes");
```

At the end, we do a simple verification and free the allocated buffer.

7.4 Cleanup

To return ownership of the port and clean up resources, a simple call to `ahci_close` suffices:

```
ahci_close(ahci_binding, NOP_CONT);
```

8 Blockdevice Filesystem

Barrelfish offers a simple VFS layer for accessing different filesystems. blockdevfs adds a further layer to facilitate exporting of file-like objects to the filesystem layer. There is not restriction on the nature of these files, apart from having to be of a fixed size.

The backends of blockdevfs can expose an arbitrary number of filenames. The filenames from different backends are combined to form the root directory of the blockdevfs filesystem. VFS calls are mapped to the corresponding backend. The filesystem only consists of a single directory with no nested directories. Files cannot be created nor deleted or truncated.

8.1 Datastructures

blockdevfs keeps a very simple doubly-linked list of directory entries. These entries contain a file name, file position, file size, backend type and backend handle. blockdevfs does not enforce any kind of order in this list. Therefore, enumerating the contents of the blockdevfs root directory will yield the files registered by blockdevfs backends in the order they were added to blockdevfs. When routing VFS calls to the right backend, the number stored in backend type is used as an index into the `backends` array holding function pointers to the backend's operations.

Figure 8.1 shows how the directory structure looks like with two entries. `prev` and `next` are used to implement the linked list. `path` holds a pointer to the filename. `size` contains the size of the file in bytes. `type` is either 0 for the *libahci* backend or 1 for the Flounder-based backend. `backend_handle` points to an internal handle private to the backend. `open` is a boolean value indicating if the file has been opened already.

blockdevfs backends must use the `blockdev_append_entry` function to register files they export.

8.2 Backend API

blockdevfs only exports `blockdev_append_entry` which can be used by backends to register their exported files. A backend can choose the `backend_handle` freely. This handle will be passed as an argument to all VFS related functions.

For standard VFS operations, backends need to provide these four functions:

- `open(void *handle)` to open an exported file. The backend does not have to check or manipulate any blockdevfs-specific structures. blockdevfs ensures that only one client has a file open concurrently.

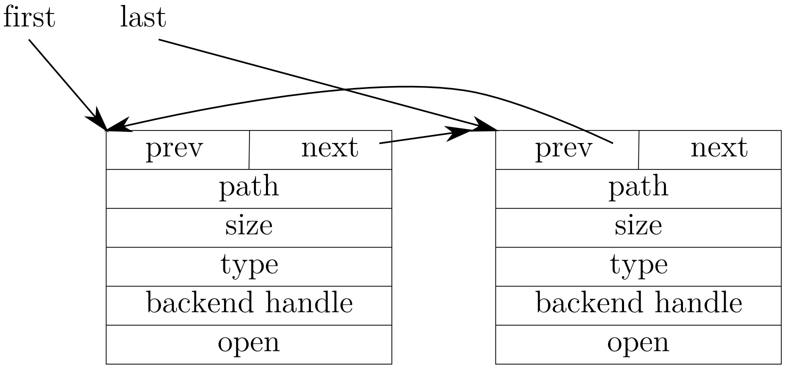


Figure 8.1: Directory entries of blockdevfs

- `close(void *handle)` to close a previously opened file. As with open, blockdevfs takes care of manipulating its structures.
- `read(void *handle, size_t pos, void *buffer, size_t bytes, size_t *bytes_read)` to read from the file corresponding to the handle.
- `write(void *handle, size_t pos, void *buffer, size_t bytes, size_t *bytes_written)` to write to the file corresponding to the handle.
- `flush(void *handle)` to flush all data of the file correpsonding to the handle to persistent storage.

All functions are supplied with the backend-handle associated with the corresponding file.

8.3 Usage

blockdevfs can by mounted by issuing `mount mountpoint blockdevfs://` and does not accept any further parameters.

Upon mounting, blockdevfs initializes its backends which in turn populate the list of directory entries. Listing the directory contents will yield any attached disk drives and report their sizes.

8.4 Backends

Currently the block device file system has two backends. One backend uses libahci standalone and the other backend uses the Flounder-generated ATA interface. The backends are named the *ahci* and *ata* backend respectively.

As both these backends expose the same devices (namely any SATA disks attached to the AHCI controller), the file names for the devices are composed of the backend name and the device's unique id, e.g. *ahci0* and *ata0* for the device with unique id 0. Keep in mind that *ahcid* prevents concurrent access, therefore you can't open the respective *ata* and *ahci* devices at the same time.

8.4.1 AHCI Backend

The AHCI blockdevfs backend implements the open and close commands by calling the corresponding functions in libahci (`ahci_init` and `ahci_close`) and implements read and write by allocating a DMA buffer using `ahci_dma_region_alloc`, constructing an appropriate FIS and calling `ahci_issue_command`. The read implementation updates the `rx_vtbl.command_completed` pointer to point to `rx_read_command_completed_cb`. That function then uses `ahci_dma_region_copy_out` to copy the read bytes from the DMA buffer to the user buffer, frees the DMA buffer, and calls the user continuation. The write implementation copies the bytes that need to be written to the DMA buffer (using `ahci_dma_region_copy_in`) and updates the `rx_vtbl.command_completed` pointer to point to `rx_write_command_completed_cb` which frees the DMA buffer and calls the user continuation. Flush is implemented by issuing the `FLUSH CACHE` ATA command which flushes the on-disk cache to the harddisk proper.

8.4.2 ATA Backend

The ATA blockdevfs backend implements the open command by initializing an RPC client to the `ata_rw28` Flounder AHCI interface. The close command just calls `ahci_close` so that a subsequent open-call on the same blockdevfs file is successful. The read, write and flush commands are easy to implement using the RPC client to the Flounder AHCI interface by just calling the `read_dma`, `write_dma` and `flush_cache` functions in the RPC function table.

8.5 Restrictions

As blockdevfs is only intended to provide a simple way for VFS aware applications (e.g. fish) it has several restrictions:

- The size of the files should not change. Although a backend might change the size stored in the handle dynamically, blockdevfs is not geared towards this.
- Subdirectories are not supported.
- Only one client can have a file open.
- Files cannot be removed, neither by the user nor by the backend.

8.6 VFS adaptation

In order to ensure that data written to a block device really gets written to the hard disk, we added a new VFS call, namely `vfs_flush`, which is used to flush the hard disk's volatile cache. vfs_flush returns VFS_ERR_NOT_IMPLEMENTED for VFS backends that have no handler for flush in their struct vfs_ops table.

9 FAT Filesystem

9.1 Overview

The layout of the FAT16 and FAT32 filesystems can be seen in Figures 9.1 and 9.2 respectively. The File Allocation Table (FAT) itself is simply a linked list, where the value of a cell indicates the index of the next cell, and special values indicate unused, bad and list-terminating cells. The data area is split into clusters with sizes a multiple of the sector size. The cluster corresponding to a FAT entry is simply the cluster with the same index, i.e. for an index i the FAT entry is $fat\_start + i \cdot entry\_size$ and the cluster entry is $clusters\_start + i \cdot cluster\_size$.

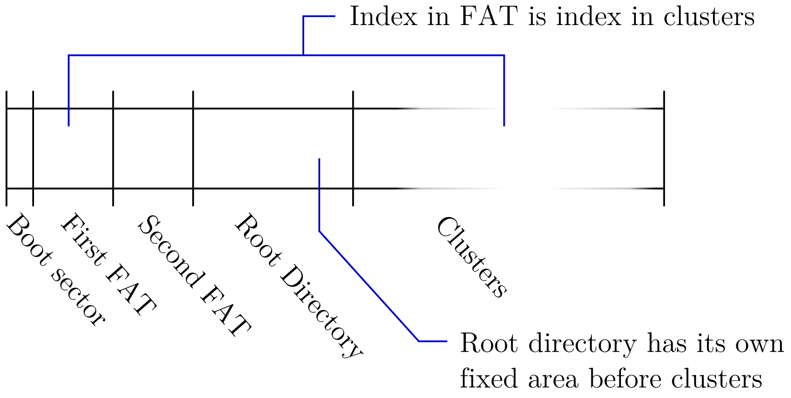

Index in FAT is index in clusters
Boot sector
First FAT
Second FAT
Root Directory
Clusters
Root directory has its own fixed area before clusters

Figure 9.1: FAT16 Layout

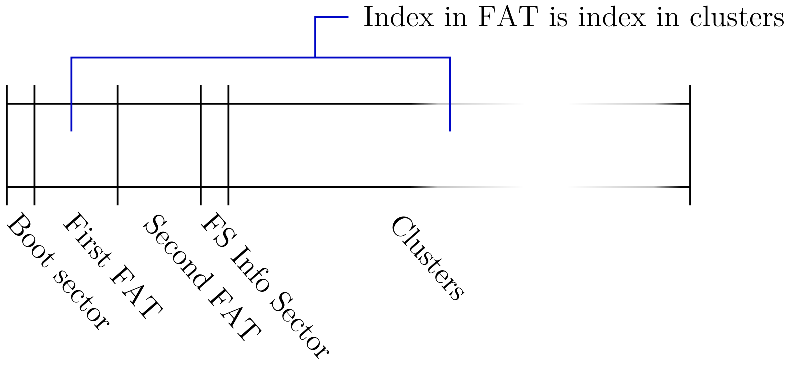


Figure 9.2: FAT32 Layout

FAT16 (and FAT12) have the particularity that the root directory is not like other directories, but is instead inside its own area preceding the start of the "clusters" area. This also implies that the maximum number of entries in the root directory is fixed when formatting. FAT32 removes this limitation, and adds an additional Filesystem Information Sector (FSIS) containing dynamic information about the state of the filesystem, e.g. the amount of allocated/free space.

9.2 Implementation and Limitations

We have implemented read-only support for FAT16 and FAT32. However, because the example `ata_rw28` interface only has the 28-bit `READ DMA` and `WRITE DMA` commands, we can only access the first 128GB of a disk (with 512-byte sectors).

9.2.1 Unicode

While FAT 8.3 filenames are 8-bit strings, FAT long filenames use UTF-16. Barrelfish does not have any concept of Unicode, so our FAT implementation replaces non-ASCII characters with a question mark in directory listings, and does not support opening files with non-ASCII filenames.

9.2.2 BSD conv Functions

To generate 8.3 filenames in the first place, we have adapted various conversion functions from OpenBSD's msdosfs. However, our current implementation still compares filenames case-sensitively.

9.3 Caching Layer

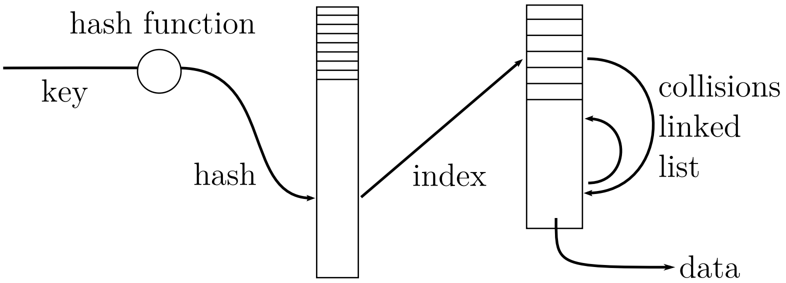


Figure 9.3: Cache Design

The FAT code uses a cache layer as a global block and cluster store, simplifying the code and improving performance. The cache is implemented as a fixed-size hashmap from keys to indices into a backing array. The backing array uses doubly linked lists to handle collisions, the free list, and a list of unused cache entries that can be freed if space is required. Clients must acquire a reference to a cache entry, either using fs_cache_acquire if the entry is already present, or fs_cache_put when creating a new entry. When the entry is no longer used, the client must call fs_cache_release. If the reference count for an entry sinks to zero, it is appended to the aforementioned list of unused entries, which can be seen as an LRU queue. Thus when fs_cache_put is called and the cache is at its maximum capacity, it can pop the front entry from the unused list, free its data, and use the entry for the new cache item.

The caching API consists of the following methods:

- fs_cache_init and fs_cache_free, for cache setup and teardown. The initialization method takes the maximum capacity of the backing array and the hashmap size. Both values must be powers of two.
- fs_cache_acquire, for getting a reference to an existing entry.
- fs_cache_put, for adding an item to the cache. This also increments the reference count as if fs_cache_acquire had been called.
- fs_cache_release, for releasing a reference to an entry.

9.4 VFS Interaction

The mount URI for FAT has the format fat<version>://<port>[+<startblock>], e.g. fat32://0+63, where version is is either 16 or 32, port is the AHCI port of the device, and the optional startblock specifies the offset the first sector of the filesystem (the boot sector).

Unlike Barrelfish's ramfs, our FAT implementation does not share state between multiple mounts using IDC, so with the current VFS implementation mounting a FAT filesystem gives the mounting domain exclusive access to the filesystem and the whole disk. An alternative that would avoid code duplication would be for the VFS to allow part of its directory structure to be exported as a service, creating a Barrelfish-internal system conceptually similar to NFS.

10 Running the AHCI Driver

This chapter details the ways the AHCI driver can be run and elaborates on the adjustments needed to be able to run the driver on real hardware.

10.1 QEMU

Since version 0.14, QEMU contains an emulation layer for an ICH-9 AHCI controller[1]. To define a disk, the QEMU command line needs to be extended with:

```
-device ahci,id=ahci -device ide-drive,drive=disk,bus=ahci.0 \
-drive id=disk,file=/path/to/disk.img,if=none
```

QEMU emulates an ICH-9 controller sufficiently well that little special code is required. A first workaround is needed for finding the AHCI PCI BAR; since the QEMU AHCI emulation layer does not provide the legacy IDE compatibility mode, the AHCI MMIO region is found in BAR 0 instead of BAR 5. Another workaround is necessary when receiving the response for an `IDENTIFY`, which is a PIO command but is delivered as a Device to Host Register FIS by QEMU.

10.2 Physical Hardware

Running Barrelfish on real hardware, one can run into several issues. In order to be able to test our AHCI implementation, we had to adjust several aspects of Barrelfish outside the scope of the AHCI driver infrastructure. This chapter details any additional modifications.

10.2.1 PCI Base Address Registers

Barrelfish's system knowledge base is not able to handle addresses above 32 bit correctly. Fixing this issue would require extensive modifications in the SKB which are out of the scope of this lab project. As a consequence, *ahcid* will receive zero BARs on hardware where the AHCI memory regions are mapped in memory above 4GB and thus be unable to access the memory mapped I/O region to control the HBA. For the same reason, the code produced by this lab project has not been tested for 64bit addresses in memory mapped I/O regions. Still, once the issues in the HBA have been fixed, the driver should properly recognise and handle the devices in question.

[1] The QEMU 0.14 changelog is available at `http://wiki.qemu.org/ChangeLog/0.14`

10.2.2 PCI Bridge Programming

Because of a bug in PCI bridge programming, Barrelfish sometimes does not program PCI BARs correctly if PCI bridges are present. For this lab project, we introduce a workaround that will retrieve the original BARs in case no reprogrammed BARs can be found in the SKB.

This is achieved in the `device_init` function of `pci.c`, by querying the SKB for the original `bar(...)` facts of the device if no reprogrammed ones can be found:

```
error_code = skb_execute_query(
  "findall(baraddr(BAR,Base,0,Size),bar(addr(%u,%u,%u)"
  ",BAR,Base,Size,_,_,_), BARList),"
  "sort(1, =<, BARList, L),"
  "length(L,Len),writeln(L)",
  *bus, *dev, *fun);
```

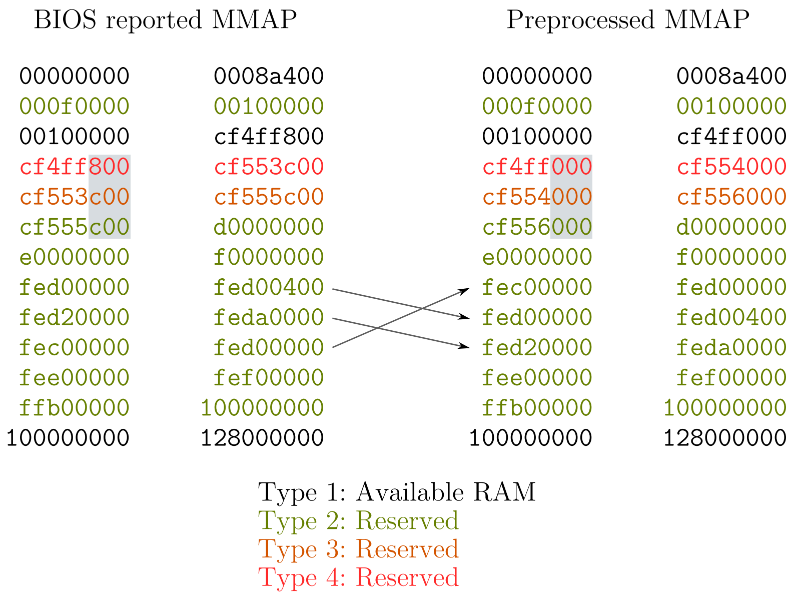
The result of this prolog expression has exactly the same form as `pci_get_implemented_BAR_addresses` therefore the surrounding code is exactly the same as in the usual case.

10.2.3 BIOS Memory Maps

On x86 architectures, the BIOS memory map can be retrieved to determine the layout of memory. Some BIOSs report a memory map that is not sorted by increasing base address or even might return overlapping and conflicting memory regions. This lab project contains modifications to the code that creates capabilities to physical memory in `startup_arch.c` such that the memory map is preprocessed to eliminate conflicts and ensure ascending addresses.

As the preprocessed memory map might be larger due to the case where one memory region completely contains another and thus is split into three new regions, we first need to copy the map into a larger buffer. The memory map is then sorted with a simple bubblesort. To remove conflicts, overlapping regions are given to the region with the higher type or merged if they are both of the same type. At the end, regions are page-aligned as Barrelfish can only map whole pages.

Figure 10.1 shows the memory maps seen on a DELL Optiplex 755 workstation. Several regions are not aligned to the pagesize and the region at `0xfec00000` does not appear in ascending order. After preprocessing, memory addresses appear in ascending order and are page-aligned. Note that higher types take precedence, therefore page alignment does not necessarily round down.



BIOS reported MMAP		Preprocessed MMAP	
00000000	0008a400	00000000	0008a400
000f0000	00100000	000f0000	00100000
00100000	cf4ff800	00100000	cf4ff000
cf4ff800	cf553c00	cf4ff000	cf554000
cf553c00	cf555c00	cf554000	cf556000
cf555c00	d0000000	cf556000	d0000000
e0000000	f0000000	e0000000	f0000000
fed00000	fed00400	fec00000	fed00000
fed20000	feda0000	fed00000	fed00400
fec00000	fed00000	fed20000	feda0000
fee00000	fef00000	fee00000	fef00000
ffb00000	100000000	ffb00000	100000000
100000000	128000000	100000000	128000000

Type 1: Available RAM
Type 2: Reserved
Type 3: Reserved
Type 4: Reserved

Figure 10.1: Memory map transformation

11 Performance Evaluation

All benchmarks have been executed on a DELL Optiplex 755 Workstation. Disks used are a Seagate Barracuda as well as a Western Digital Caviar Green. Benchmarks have been run on Barrelfish and Linux.

11.1 Hardware Details

11.1.1 System

The DELL Optiplex 755 desktop features an Intel Q35 Express Chipset [1] which contains an Intel 82801IR I/O controller (ICH9R-family). This is the same AHCI controller that QEMU emulates. Remaining characteristics are:

CPU	Core 2 Q6700 2.66 GHz
RAM	4 GB

11.1.2 Disk: Seagate Barracuda

The Seagate Barracuda 7200 rpm disk is a SATA II disk with 512 byte sectors [4]. Disk characteristics:

Capacity	160GB
RPM	7200
Max. sustained data transfer rate	78 Mbytes/sec
Avg. seek time read	<11ms
Avg. seek time write	<12ms
Cache	8MB

11.1.3 Disk: Western Digital Caviar Green

Although the Caviar Green disk reports a 512 byte sector size, the disk in question uses Western Digital's Advanced Format technology and thus internally uses 4096 byte sectors [5]. Disk characteristics:

Capacity	1.5TB
RPM	5400 - 7200
Max. sustained data transfer rate	110 Mbytes/sec
Avg. seek time read	<8.9ms
Avg. seek time write	<8.9ms
Cache	64MB

11.2 Software Details

11.2.1 Linux

We use a Ubuntu 11.04 live cd booted via PXE. Benchmarks are performed on the raw /dev/sdX files via native file handles. After writes, an `fsync` call is performed prior to time measurement to ensure changes are written to disk.

11.2.2 Barrelfish

Barrelfish is booted via PXE. All benchmarks are performed on the files exported in the blockdevice filesystem. Benchmarking workloads use the VFS API to access files.

11.3 Workloads

We implemented three benchmark workloads. `fillbench` sequentially writes a number of blocks to the disk, filling each block with its number. After all blocks are written the timer is stopped and the blocks are read back and their contents verified. `randbench` reads all blocks in pseudo-random order with one timer, then writes all blocks in pseudo-random order with a second timer, and measures the throughput for read and write. The sequence of locations is determined by a linear feedback shift register. `randbench_time` is the same workload as `randbench` but measures response time.

Algorithm 11.1 Sequential Benchmark

```
open disk; start timer
for $i = 0 \rightarrow count$ do
    write block $i$
end for
close disk; stop timer;
open disk; start timer
for $i = 0 \rightarrow count$ do
    read block $i$ and verify
end for
stop timer; close disk
```

Algorithm 11.2 Shuffle Benchmark

```
open disk; start read timer
for $i = 0 \rightarrow count$ do
    $r_i := rand()$; read block $r_i$
end for
stop read timer; start write timer
for $i = 0 \rightarrow count$ do
    $w_i := rand()$; write to block $w_i$; flush disk
end for
stop write timer; close disk
```

11.4 Sequential Access Results

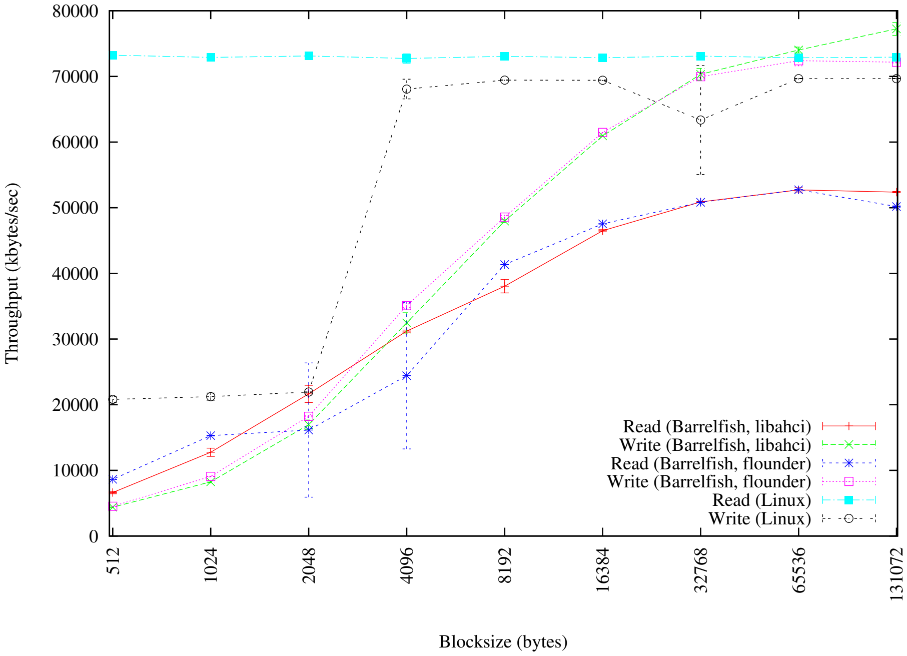


Figure 11.1: Sequential read/write performance (Seagate Barracuda 7200.10)

- The similarity of the lines from direct libahci calls and calls through the flounder-generated layer show that the flounder layer has no significant performance impact.

- Linux performance hints at the underlying architecture: even with a flush reads and writes go through the cache, so are always at least one cache page. This impacts the performance of Linux's write for small blocks, where a whole cache page must be written every time.

- Barrelfish read performance is bad for small blocksizes because we do not prefetch data for sequential reads. The read performance for large blocksizes might be worse than Linux because the command completion notification has to go through IDC. Other possible reasons may involve scheduling and `memcpy` performance.

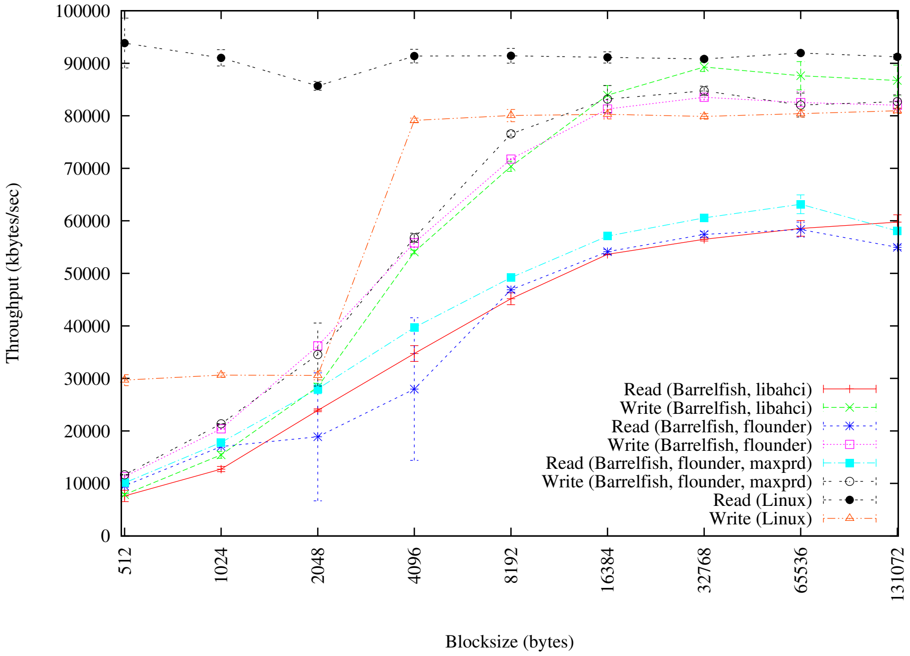


Figure 11.2: Sequential read/write performance (WD Green EARS)

- Measuring sequential performance using a Western Digital disk gives qualitatively the same results as with the Seagate disk.
- We also see that 4K sectors don't have a large impact on both Linux and Barrelfish performance.

11.5 Random Access Results

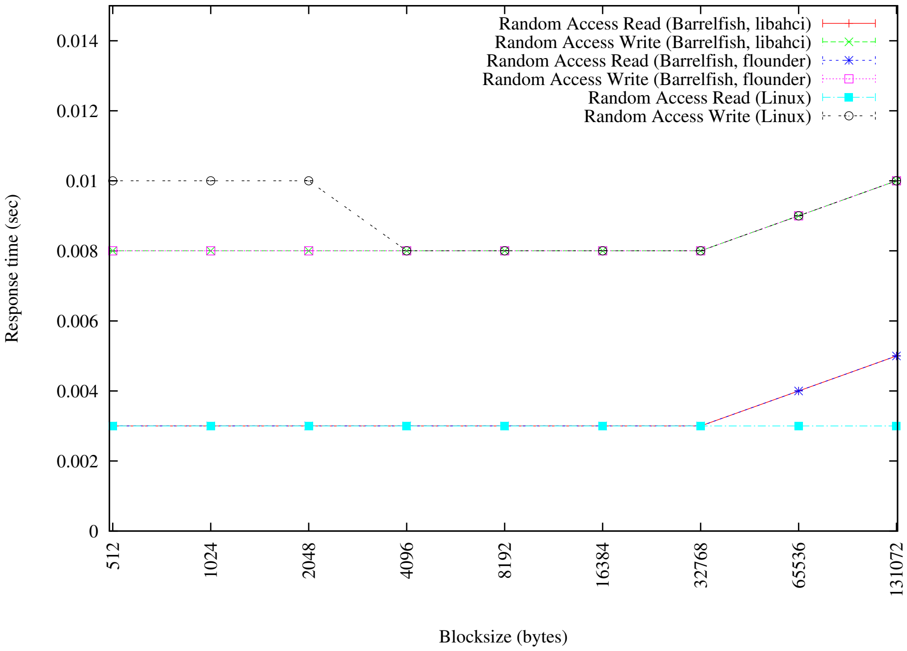


Figure 11.3: Random read/write response times (Seagate Barracuda 7200.10)

- We see that for random accesses our implementation performs as good as a current Linux implementation.
- Also software overhead is not noticeable as we are comfortably below the disk's specified maximum seek time.

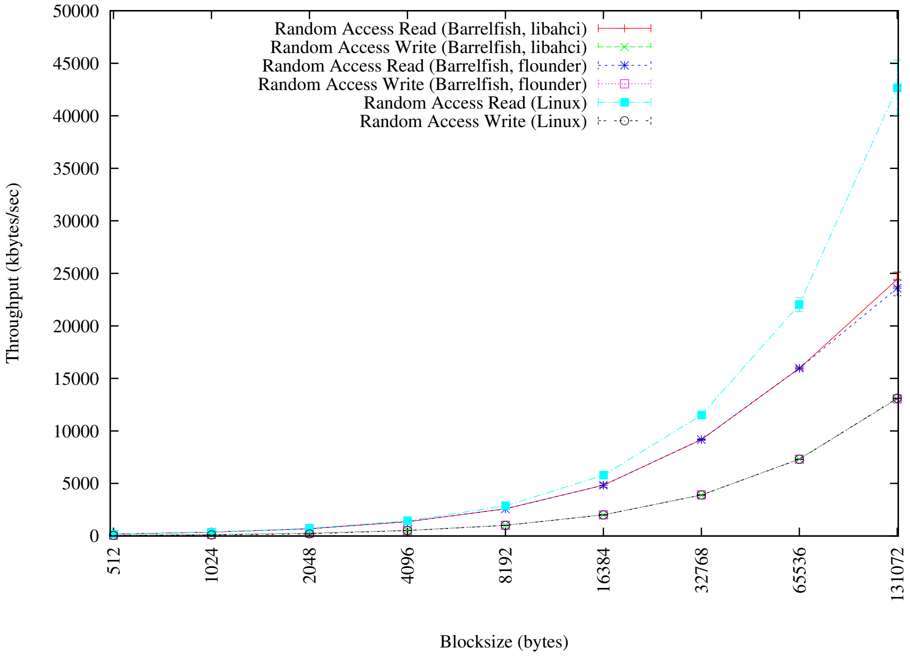


Figure 11.4: Random read/write throughput (Seagate Barracuda 7200.10)

- The random access throughput measurements show again that our read performance is worse than Linux. What we also see is that write performance is equal for Linux and Barrelfish

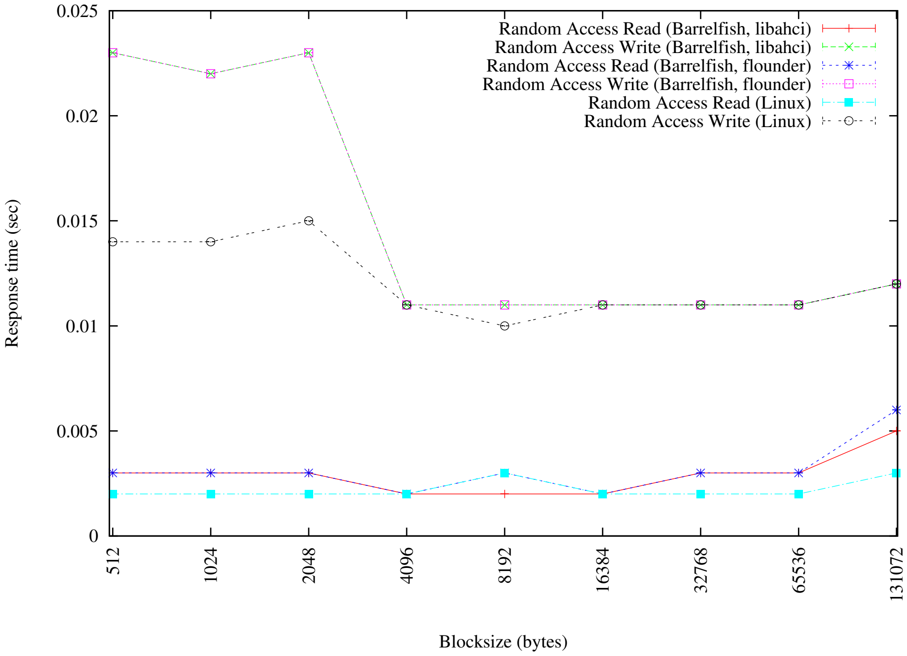


Figure 11.5: Random read/write response times (WD Green EARS)

- In the random write performance we clearly see the impact of the 4K sectors on writes with blocksize < 4096 bytes. The reason for the doubled response time is that the disk needs to read the whole sector, update it and write it back.
- We also see that our implementation is on par with Linux for read response times.

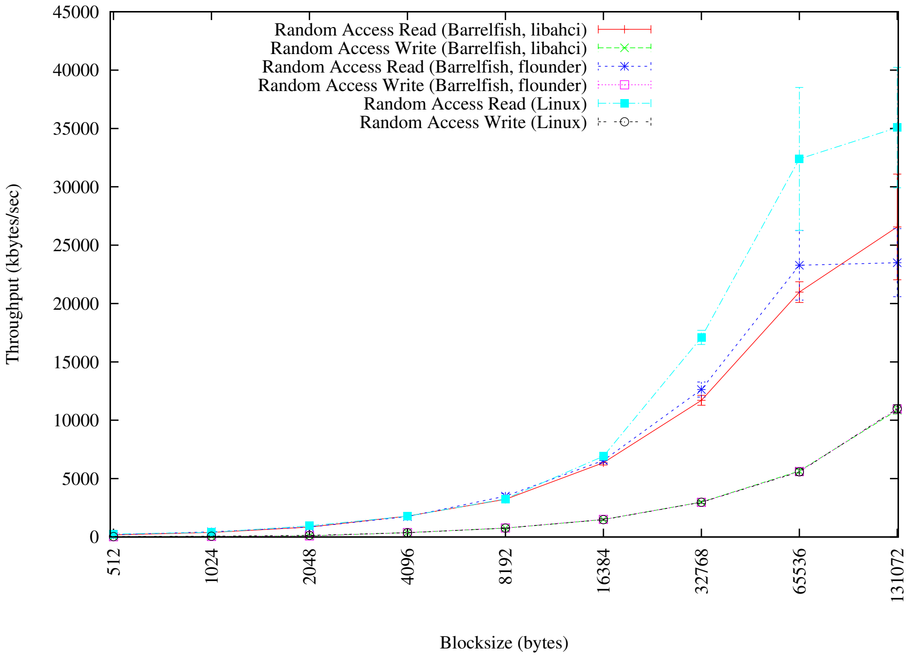


Figure 11.6: Random read/write throughput (WD Green EARS)

- Once more, we see that the read throughput of our implementation is not as good as Linux read throughput.
- Write throughput is equal for Linux and Barrelfish as in previous results.

12 Future Work

12.1 ATA Messages

Since this lab project was geared towards designing and implementing a message passing interface to disk, only very few message types have been defined to showcase the interface. SATA supports a wider range of devices which can benefit from more aspects of the ATA command set, such as `TRIM` on solid state drives. Adding further commands to the flounder-based approach is as simple as adding a further message definition.

12.2 Integration with the System Knowledge Base

As we have mainly focused on getting message passing to disk to work, we have taken a few shortcuts concerning the integration of our subsystem into Barrelfish as a whole. For instance, we do not really use the SKB to uniquely identify the disks attached to an AHCI controller. Neither do we use the SKB to store additional data, e.g. serial number and size, of the attached disks. Adding that kind of data would simplify discovery of disks and acting appropriately, for example automatically mounting a volume or similar.

12.3 Handling multiple AHCI controllers at the same time

Currently our management daemon (see chapter 4, ahcid) successfully exits from the initialization code as soon as a AHCI controller has been found. It would be prefereable if on systems with multiple controllers attached all of these could be used. One consideration in this case would be whether to have one management daemon per controller or a global management daemon that controls all available AHCI controllers.

12.4 Support for advanced AHCI/SATA features

Some of the features of AHCI/SATA we did not look at are Port Multiplication and Native Command Queueing (NCQ).

However, our system design, including a management daemon that presents each port to the rest of the system as a separate entity, makes accomodating multiplied ports (multiplying ports is actually done in hardware and the AHCI host controller has a register for each port which contains the port multiplication status for that port) relatively easy as the only parts that have to be changed are the management daemon and libahci. Also, NCQ could be implemented almost entirely in libahci, if desired.

We also do not handle hotplug of devices. Addition of devices could implemented relatively easy by extending ahcid's interrupt handler and performing the initialization steps once the link to the device has been established. Removal however is more challenging. Outstanding requests have to be completed with an error, the user notified and memory resources reclaimed.

12.5 Further Controllers

The modular nature of the Flounder-based approach allows to add additional backends for other controllers. Since this lab project only examined AHCI-compliant controllers, support is limited to realtively new controllers for SATA. In reality, there are still a lot of use cases where one might have to access PATA-based devices, such as older CDROM drives. Therefore, backends for more chipsets should be developed, most importantly one for a widespread PATA controller such as the PIIX family or the ICH controllers before the introduction of AHCI.

13 Conclusion

In the course of this lab project we successfully implemented an AHCI driver and supporting code for data storage and retrieval.

13.1 Flounder Modifications

The extensions added to flounder provide a very simple and extensible way to interface with disks. The overhead incurred is acceptable in the trade-off for simplicity and modularity. The seperation of interface definition for ATA from implementation of command dispatching to the device allows simple addition of further ATA transports, such as additional PATA/SATA controllers.

13.2 Security

The AHCI driver demonstrates the trade-off when dealing with DMA. If a domain is allowed full control over the configuration of DMA aspects, it can obtain full read/write access to physical memory. To mitigate this problem, the management service would have to check and validate any memory regions supplied before allowing a command to execute. If only trusted domains are allowed to bind to the AHCI driver, these checks are not neccessary. This is a valid assumption, as filesystems and blockdevice-like services are the only ones that should be allowed raw access to disks.

13.3 Performance

Performance is in the same order of magnitude as seen on Linux for large blocksizes and random access. There is some bottleneck during read operations that could relate either to interrupt dispatching or memcopy performance. To achieve high throughput on sequential workloads with small blocksizes, a prefetcher of some sort is necessary. A possible solution would be to have a cache that stores pages or larger chunks of data. A read operation would then have to read multiples of the cached size if the data is not present in the cache. If data is cached, the request can be completed much faster without needing to consult the disk.

Bibliography

[1] DELL. *Optiplex 755 Technical Specification*, 1.0 edition, 01 2008. http://www.dell.com/downloads/global/products/optix/en/opti_755_techspecs.pdf.

[2] Intel Corp. *Serial ATA Advanced Host Controller Interface (AHCI) 1.3*, 06 2008. http://download.intel.com/technology/serialata/pdf/rev1_3.pdf.

[3] SATA-IO. *Serial ATA Revision 2.6*, 02 2007. https://www.sata-io.org/developers/purchase_spec.asp.

[4] Seagate. *Barracuda 7200.10 Product Manual*, 12 2007. http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.10/100402371k.pdf.

[5] Western Digital. *Caviar Green Series Specifications*, 05 2005. http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701229.pdf.

Acronyms

ACS	ATA Command Set
AHCI	Advanced Host Controller Interface
ASCII	American Standard Code for Information Interchange
ATA	AT Attachment
ATAPI	AT Attachment Packet Interface
BAR	Base Address Register
CAM	Common Access Method
DMA	Direct Memory Access
FAT	File Allocation Table
FIS	Frame Information Structure
FSIS	Filesystem Information Sector
HBA	Host Bus Adapter
ICH	I/O Controller Hub
IDC	Inter-Domain Communication
LBA	Logical Block Address
LRU	Least-recently used
NCQ	Native Command Queueing
PATA	Parallel ATA
PCI	Peripheral Component Interconnect
PIIX	PCI IDE ISA Xcelerator
PRD	Physical Region Descriptor
RFIS	Received FIS
RPC	Remote Procedure Call
SATA	Serial ATA

SCSI	Small Computer System Interface
SIM	SCSI Interface Module
SKB	System Knowledgebase
VFS	Virtual Filesystem