OpenBSD SCSI Evolution

David Gwynne
The University of Queensland, Australia

Kenneth R. Westerback
Westerback Software Foundry Inc., Canada

Abstract

The SCSI protocol has become the lingua franca of block oriented I/O. OpenBSD has always supported SCSI devices, but recently the OpenBSD SCSI stack has been significantly enhanced to improve stability, performance and scalability.

This paper presents the architectural details of the recent changes. It describes the state of the SCSI stack (a.k.a. SCSI midlayer) at the beginning of the changes, identifies issues driving those changes, describes current state and lays out some near term goals for the SCSI stack.

1 Introduction

In the beginning SCSI (Small Computer System Interface) was one of many protocols used to communicate with peripheral devices such as disk drives. It co-existed with other standardized protocols such as IPI (Intelligent Peripheral Interface) and ESDI (Enhanced Small Disk Interface), along with a plethora of vendor specific attachment technologies.

To the kernel it was one of several exit points for the buffer cache I/O generated by various file systems. The OpenBSD SCSI stack was a minimalistic layer that translated between buffer cache I/O's and requests of hardware devices.

The basic design can be summarized as:

(a) At system startup all attached SCSI buses would be probed for attached devices. Each device discovered would have a fixed limit of concurrent operations. This limit was called the number of 'openings' available for the device.

(b) The buffer cache would submit block I/O requests via the SCSI device's strategy routine. eg sdstrategy() for disk devices (sd(4)).

(c) The strategy routine would convert the block I/O requests into scsi_xfer requests and issue the scsi_xfer requests to the hardware adapter. This was often done in interrupt context as part of the completion processing of a previous I/O.

(d) The strategy routine would NEVER issue more commands than the hardware could handle, ie, the 'openings' limit would always be respected. If the openings limit was exceeded then I/O's would error out, reporting EIO to the buffer cache code.

SCSI got popular, crowding out competitors. It got its own standards committee, and accumulated various vendor interest groups. There followed in quick succession Fast, Wide, SCSI-II, SCSI-III, SCSI-IV, Ultra, Ultra-2, Ultra-3, Ultra-4, U160, U320, 32 bit parallel buses, serial buses, fibre buses, etc. New attachment technology such as USB and ATAPI used significant chunks of SCSI design and implementations.

The increasing demands of the SCSI protocol were dealt with in OpenBSD almost completely within the hardware adapter code for each new SCSI adapter. Outside of bug fixes and code cleanup there were few changes in the actual SCSI midlayer. This was the advantage of the minimalistic design: there wasn't much to change.

Today SCSI is everywhere, it is all around us. Even now, in this very room.

2 History

This paper does not attempt to describe the entirety of SCSI and its history. Instead it describes the state of the OpenBSD software stack and supported hardware around 2005 when major changes began to be make. The description will provide only enough detail to understand the issues driving the changes that are the topics of later sections.

2.1 Hardware

The OpenBSD SCSI midlayer provided the normal SCSI device abstractions to the rest of the kernel. ie disk (sd(4)), cdrom (cd(4)), tape (st(4)), media changer (ch(4)), scanner (ss(4)), and enclosure (ses(4)) devices.

Of these only sd(4), cd(4), and st(4) were in common use and likely to work with actual hardware. Only sd(4) had the requirement or normally attempted to execute more than one SCSI command at a time. This paper will focus on sd(4) devices.

The SCSI adapters supported were varied, and ranged from devices where the SCSI bus transitions were directly controlled (eg trm(4)), to RAID controllers which provided interfaces that closely resembled the SCSI command set (eg arc(4)).

The SCSI adapters could be connected via hardware busses such as ISA, EISA, or PCI.

2.2 Software

At startup the kernel would probe each SCSI bus found, attempting to contact every possible SCSI target and lun. Each bus would allocate a fixed set of resources, usually sufficient to control 256 I/O's. The number of openings that would be given to each device discovered was calculated by dividing the the set of allocated resources by the maximum number of targets. A narrow bus had 8 possible targets and a wide bus had 16. Thus a device discovered on a 'wide' bus would receive resources allowing 16 I/O's to be started at one time. If only 1 device was found the other resources would remain unallocated and unused.

The normal path for a disk I/O was

(a) sdstrategy() was invoked with a pointer to a struct buf describing an I/O. The buf was placed into a sorted queue of I/O requests.

(b) sdstart() was invoked from sdstrategy() to process the queue. If the destination device had unused 'openings', requests were removed from the queue and sent to the device via scsi_scsi_cmd() until either the openings or the request queue were exhausted.

(c) scsi_scsi_cmd() created a scsi_xfer struct that described the desired I/O in SCSI terms, and submitted the scsi_xfer to the scsi_cmd() entry point provided by the hardware adapter.

(d) scsi_done() was invoked by the hardware adapter in interrupt context to return the I/O result via the original buf.

(e) scsi_done() invoked sdstart() to process the queue of I/O requests.

There were a number of known issues, all irritating rather than crippling. They included

(a) 'openings' were proving a porous limit, and hardware adapters were getting more I/O's than they could handle. This resulted from SCSI commands being injected by scsi_scsi_cmd() calls from outside the normal block I/O path described above, eg via ioctl calls. As a result I/O's could randomly fail.

(b) many hardware adapters had created their own internal queuing to handle excessive I/O requests, leading to significant code duplication, and confusion about who was really responsible for the I/O request queue.

(c) static allocation of resources was wasteful of scarce kernel memory when so few devices were normally present on any one bus.

(d) equal distribution of static resources to all possible, as opposed to actual, targets meant that the system could not provide more resources to busy devices when required, even though the resources were available.

(e) SCSI busses could support many more devices than the 8 or 16 originally envisioned.

The irritation began to edge into annoyance in 2005. In March, bioctl(8) was committed, in August safte(4) and ses(4) were committed. In July 2006 a major rewrite of the ch(4) device was committed. With these, and improvements to the sensors framework for reporting system information, the number of SCSI commands being injected outside of the block I/O path protected by the openings limit began to be so significant that problems were encountered with some regularity.

The first reaction was to reserve some of the available openings for these alternative paths. This was done by setting openings to a value less than the number of I/O's the adapter code could handle. This alleviated most but not all of the problems with failing I/O's. And it exacerbated the growing disconnect between the openings values and the hardware adapter resources that the openings were supposed to represent.

The second reaction was the first architectural change to the SCSI midlayer in many years. This was the introduction at h2k6 at the end of 2006 of a new mechanism to allow the scsi_cmd() entry points of the hardware adapters to reject I/O's, with sdstart() gaining the ability to re-queue the rejected I/O's.

This mechanism, termed NO_CCB after the new result code, was largely adapted from NetBSD.

Over the course of 2008 and 2009 NO_CCB allowed the removal of reserved openings, and the elimination of most of the internal queueing in the hardware adapter

code. In itself it proved insufficient to solve all the problems. But it had removed the psychological barrier to considering significant changes to the SCSI midlayer.

The floodgates had been opened.

3 SCSI Moving Forward

SCSI hardware has evolved and devolved at the same time. Once upon a time SCSI meant one type of hardware interconnect with a well known number of targets and possible luns. These days the SCSI bus is the lingua de franca of storage. Even ATA, despite being extremely common in its own right, is being placed behind SCSI emulation layers. Hardware RAID controllers used to provide their own command sets, but the trend with modern RAID controllers is to simply provide a SCSI interface and translate it inside its own firmware.

The agnosticism of the SCSI command set towards the transport mechanism is likely responsible for this success.

It has evolved toward very complicated ”enterprise” solutions featuring high availability with multiple paths between the host and storage, high speed interconnects etc. It has devolved in the case of USB. The concepts and solutions found in the high end are making their way down into things like iSCSI and SAS and are becoming more common.

3.1 Software

Throughout this period the OpenBSD SCSI stack did generally keep up with support for the newer and varied hardware. These changes included the widening of SCSI buses, ie, allowing more than 8 or 16 targets on a bus, increasing the lun depth from 0 or 8 to an arbitrary value, working hotplug (but perhaps not safe hotplug), and support for modern command sets (eg, SBC2 and SBC3 for larger disk support)

However, while the stack did support most new developments, and did indeed support newer physical hardware, it lacked the ability to scale and adapt. Newer physical hardware features much wider buses and deeper luns, larger command queues, and out of band command handlers, all of which strain at the original midlayer's constraints and assumptions. We could use modern hardware, but we had to compromise to use it. The semantics of the original SCSI midlayer did not allow us to make effective or fair use of the resources available to us.

Furthermore, as multipathing nears the mainstream with the advent of iSCSI, the original architecture acted as an obstruction to proper multipath support in OpenBSD.

The rest of this paper will discuss changes made in the last 2 years to solve some of the architectural issues we inherited.

4 Probing Adapters For Devices

The SCSI midlayer was written with the assumption that adapters were simply transports for SCSI commands. There was no state stored in an adapter about any devices attached to it, it just transported the commands. The midlayer probed for devices by issuing INQUIRY commands at every possible address on the bus and if it got a response it proceeded to try and attach a device.

This assumption was flawed because of how SCSI is mapped on top of various transports. For example, OpenBSD now emulates SCSI on top of some ATA controllers. The emulation code needs to know what type of ATA device is present so it can provide the appropriate SCSI emulation for it before the midlayer talks to it.

Previously adapters that needed to allocate state for a device used to snoop the commands going past their SCSI command handlers and look for things it knew the midlayer would request at certain points in the device's lifecycle. Because there was no recognizable command at the end of a device's life (usually its life ends because its been removed), the adapters were rarely able to properly free up state associated with a device when it went away. This wasn't a huge problem for things like USB since the bus and adapter go away at the same time, but it was becoming an issue when trying to support hotplug busses and RAID volumes which need different state saved for different targets.

We added adapter handlers so the midlayer could ask the adapter if there might be a valid device at that address and give the adapter a chance to modify the device before the midlayer talks to it. For example, you can attach ATAPI SATA devices to SAS mpi(4) controllers, but by default mpi(4) assumes anything attached to it is a normal SCSI device. The probe adapter handler in mpi(4) will query the firmware to see if a device is ATAPI so it can set the appropriate flags for cd(4) to use once it is attached, so it in turn will generate appropriate commands for the device.

Once the midlayer has finished with a device (or it found no device during its probe) it calls another adapter handler to clean up any state the adapter did allocate.

5 Fixing The SCSI Command API

During the f2k9 hackathon hosted in Sweden in September 2009 an attempt was made to implement SCSI multipathing.

SCSI multipathing involves collecting all the physical paths to a single SCSI device and presenting a single abstracted instance of the device for use in the kernel. As

physical paths appear and disappear, the abstracted device remains constant and accessible via these backend paths.

The actual implementation of this functionality involved having the SCSI midlayer detect and steal paths to devices that could support multipathing. These paths were then used behind the scenes by a virtual SCSI bus called mpath(4). mpath would then take requests from a single virtual device and issue another SCSI command request on its behalf to a physical device.

The executive summary of this is that the SCSI command interface was called twice for every SCSI operation to a multipathed device, the first going to the virtual device and the second for the command going to the physical device. This exposed an annoying "feature" of the SCSI Command API.

5.1 The Original SCSI Command API

All requests used to go via a single function called scsi_scsi_cmd:

```
int scsi_scsi_cmd(
    struct scsi_link *link,
    struct scsi_generic *cdb,
    int cmdlen,
    u_char *data_addr, int datalen,
    int retries, int timeout,
    struct buf *bp, int flags);
```

One of the biggest problems with this API was that it was very hard to remember what the arguments meant when it was being used or read. When reviewing code it was difficult to recognize the meanings of some parameters because of the length of the argument list. Invocations tended to be copied without much thought about whether some of the values were appropriate in the context of the new code.

The problems that really bit us when implementing multipath support was that it isn't possible to submit a command for asynchronous completion without supplying a struct buf, and it isn't possible to inspect the result of the SCSI transfer directly.

Hardware SCSI controllers generally complete SCSI operations via interrupts, ie, you submit a SCSI command and then go do something else until it completes and generates an interrupt to let you know. scsi_scsi_cmd wouldn't let you do something else while waiting for the interrupt unless you were doing block I/O represented by the buf you pass to it. Without a buf scsi_scsi_cmd would poll or sleep while waiting for the adapter to complete the command before return to the caller.

If you were using a buf and you didn't ask scsi_scsi_cmd to poll for the commands completion, then scsi_scsi_cmd would return immediately. This would allow you to queue multiple block I/O operations until you ran out of command slots to fill and then go do something else while the hardware actually processed all those queued operations. As these block I/O operations were completed and returned to the SCSI midlayer, it would call a per driver callback to finalize the block I/O operation and finally call the block layer directly on the devices behalf to complete the I/O request.

Because the SCSI command interface only allowed asynchronous completions for block I/O requests, all the SCSI device drivers that wanted asynchronous completions faked block requests for their common I/O paths. For example, SCSI scanners forced their operations into bufs so they could perform asynchronous completions for requests. Other drivers required kernel threads to perform their actions, allowing scsi_scsi_cmd to sleep while waiting for the transaction to complete rather than polling with busy-waits on hardware state.

To effectively proxy requests from a virtual device, mpath(4) should be able to queue multiple transfers onto a physical path, but this was impossible without it creating a request of its own with a buf. Providing a fake buf without any storage associated with it and generating useless calls into the block layer was considered but quickly rejected.

In either the asynchronous or synchronous case scsi_scsi_cmd did not let you inspect the SCSI transfer it executed on your behalf to see what the hardware responded with. The midlayer would translate the transfer's state into a single int error value. Which it reported via the return code from scsi_scsi_cmd or as an error to the block layer in the buf struct.

mpath(4) needs to be able to look at the SCSI transfer directly to make appropriate decisions about what to do with the command it was proxying. For example, if the physical adapter marks a transfer as RESET or SELTIMEOUT in response to a path disappearing, mpath(4) might choose to send the transfer down another physical path rather than report the failure up to the actual disk device driver. The midlayer would translate the very precise state in the SCSI transfer to overlapping and therefore useless error codes before reporting it to the wrong software layer.

Lastly, because scsi_scsi_cmd allocated the actual resources needed to perform the SCSI transfer inside itself, it was difficult for devices to coordinate and guarantee access to those resources without doing their own locking. Drivers tended to do this locking around their common I/O path, but it was haphazardly applied to all the other paths in the same drivers that could generate SCSI commands.

While it would have been technically possible to (ab)use the interfaces provided it would have made a well performing multipath implementation extremely fragile and difficult to comprehend and maintain. The imple-

mentation written during f2k9 was simple from a code point of view, but wasn't ideal because it effectively caused I/O to multipathed devices to be issued one at a time and polled for their completion. It confirmed the opinion held by the SCSI developers that the command API was too complicated, restrictive, and rife with subtle caveats and side effects.

5.2 The Replacement SCSI Command API

After the issues with the original API became painfully apparent the following API was designed and largely implemented during f2k9.

```
struct scsi_xfer *scsi_xs_get(
    struct scsi_link *,
    int flags);
void scsi_xs_exec(
    struct scsi_xfer *xs);
int scsi_xs_sync(
    struct scsi_xfer *xs);
void scsi_xs_put(
    struct scsi_xfer *xs);
```

The goal of the API is to allow consumers to allocate a SCSI transfer (struct scsi_xfer) and set it up directly instead of through arguments to a function that would do it on your behalf. A scsi_xfer struct allocated by scsi_xs_get is preset with sane or commonly used defaults on parameters wherever possible, thereby simplifying the use and readability of a lot of code.

The scsi_xfer structure was extended so each request had its own completion routine which the consumer specified (ie, a per command completion handler rather than per a device completion handler), which would be called when the adapter completes the transfer. This completion routine would have full access to the scsi_xfer structure so it could examine its state according to the adapter.

If you wish to asynchronously complete a command, you must specify a completion handler and submit it using scsi_xs_exec. If you wish to sleep while waiting for a command to complete you submit it using scsi_xs_sync, but you must not specify a completion handler for that command as the API provides its own internal completion handler. It is assumed that you will do any completion work on the transfer once you're woken up and scsi_xs_sync returns.

There is no special behaviour inside this API dependant on whether a buf structure has been provided or whether it is a block I/O request or a simple device inquiry command. Lastly, there is no implicit free of the scsi_xfer structure by the midlayer or calls into other software layers in the kernel. Drivers that do handle block I/O requests from the buffer cache are now responsible for calling biodone() with the requests rather than having the midlayer do it on its behalf. What happens to the transfer once it completed is the responsibility of the driver that set the transfer up, it can either free the resource or use it again. Again, the midlayer no longer implicitely frees scsi_xfer structures on a drivers behalf.

The API was designed so you could use scsi_xs_put as the completion handler for a transfer so it was freed immediately after completion. In practice this has proved unnecessary.

The new API is a lot simpler to use and understand in practice because it does so much less than the previous scsi_scsi_cmd() interface. This simplicity does come at a cost. The actual device drivers have to do more work to replace the functionality that was removed from the midlayer. However, while the driver code is larger, it is more explicit and obvious what actions are taken when, and is therefore readable and maintainable.

The old and the new APIs co-existed as devices were converted, with the old API removed in July of 2010.

6 Fixing Adapter's scsi_cmd Handlers

The complexity of the scsi_scsi_cmd internals as described above extended beyond the midlayer itself and into the API it expected SCSI adapters to implement. Commands submitted to the midlayer via scsi_scsi_cmd and now scsi_xs_exec had to end up on the hardware at some point. While each instance of an adapter provides its own function, we can generally refer to these as the scsi_cmd() handler.

Once a scsi_cmd handler has been given a scsi_xfer to deal with, it obviously has to signal its completion to the midlayer. This used to be done by one of two method. The state field of the scsi_xfer could be modified and scsi_done called to return it to the midlayer, Or, if the adapter was still in the call to its scsi_cmd handler it could return COMPLETE to the midlayer which would then take on the responsibility of finishing the scsi_xfer on the adapters behalf.

So, not only did the midlayer do work on the SCSI device's behalf, like calling biodone for block I/O requests, it also tried to do work on the adapters behalf. This unclear distribution of responsibility led to fragile code at best.

This was a fairly simple semantic to understand but it was hard to reliably implement, especially in combination with the need to handle polled SCSI requests. Polled SCSI requests required scsi_cmd handlers to return COMPLETE to the midlayer, but special casing POLLED handling in the depths of an adapters command handling far from the scsi_cmd call led to very complicated code.

During the replacement of the scsi_scsi_cmd API, the adapter scsi_cmd handlers were updated to be void functions, rather than functions returning COMPLETE or SUCCESSFULLY_QUEUED by an int return type to the midlayer. This in turn meant that all scsi_xfers had to be given back to the midlayer via a call to scsi_done.

Of all the changes made to the SCSI subsystem, this was the most error prone simply because there are so many SCSI adapter drivers, roughly 80 across the entire OpenBSD tree. It was impossible to sufficiently test this change across so many different architectures and drivers because the developers did not have access to the entire range of hardware. The decision was made to commit the changes at a point in the development cycle that would allow as much testing by the community as possible before a release was made.

The most common problem falling out of these changes was discovering an adapter calling scsi_done() on the same scsi_xfer multiple times, or not at all. Apart from a few notable exceptions (ciss(4) and gdt(4) in particular), fixes for these problems were trivial.

Tracking the ownership of a scsi_xfer (ie, device to midlayer to adapter and back again) is a lot easier now as obvious API calls act as gates between these layers. For example, it is now trivial to identify when an adapter finishes processing a scsi_xfer by looking for the scsi_done calls. Any handling of a scsi_xfer after a call to scsi_done can be considered a bug (similar to a use after free) and must be fixed.

Yet again, by removing code from the midlayer for implied handling of a scsi_xfer, it was able to be simplified. in turn this enabled great improvements in the readability and maintainability of both the midlayer itself and the adapter drivers. The clarification of responsibilities led to much more robust code in general.

7 Command Allocation and Scheduling

The way the SCSI subsystem allocated and distributed openings for commands between the devices on a bus has already been discussed. An adapter would generally allocate a set number of command slots and then evenly distribute them between all possible devices on a bus. The example before described an adapter with 256 openings and a bus width of 16 targets which would carve the openings up so each target would get 16 of these openings.

The number of openings an adapter advertises to the midlayer is a contract. If the adapter says it can handle 16 openings on a device, it must be able to accept and process that number of concurrent commands. If the adapter fails to process a command and returns an error to the midlayer, an error will be reported further up the stack. Failing block I/O can lead to the block layer believing a filesystem is inconsistent and therefore corrupt.

There were several problems with the way openings were distributed between devices.

Firstly, the SCSI midlayer considers devices on luns as equal consumers of openings on an adapter. If you really wanted to evenly distribute an adapters openings amongst all possible devices on a bus you would take luns into consideration in addition to targets.

Some SCSI hardware consumes a target id on its bus, eg, traditional parallel SCSI adapters occupy the address slot at target 7 on a bus, so a device cannot exist there and should therefore not be counted as a possible consumer of the adapters resources.

Secondly, allocating resources for possible devices is obviously naive. Despite the increases in bus widths and lun depths, the number of command slots hasn't grown at the same rate, so modern disks are being allocated less openings to use compared to their historic counterparts. To compensate you can violate the principal of being conservative in dividing adapters openings up between targets, but how optimistic to be is hard to judge. Either you over-allocate and allow I/O failures to occur, or you're still too conservative and fail to effectively utilize the adapters resources.

Thirdly, even you could correctly foresee the number of actual devices and allocate openings accordingly, the number of actual devices doesn't reflect the number of devices you're using at any point in time. To explain, if you have multiple disks in a server you may dedicate one to the operating system and software, and use another disk for a database. Generally the disk with the database on it will get more I/O requests than the disk the operating system is on, therefore the openings dedicated to the operating system disks are wasted when the database disk could make use of them.

It was not possible for the midlayer to redistribute openings between devices after a bus had been probed because the midlayer itself had no visibility on what resources the adapter actually had. So far we have been discussing adapters that have a pool of openings that any target could use, but some controllers have a pool of commands for each target which cannot be redistributed to other targets. This is true of ATA controllers such as ahci(4) and sili(4). The SCSI to ATA translation maps their ports to SCSI targets, but each port on these ATA controllers has their own set of registers and command slots which cannot be used by other ports. Also, because device drivers (eg, cd(4)) reduce their openings below what the adapter allocates to them, and adapters cannot easily determine which drivers are attached where, it was not possible for the adapter to redistribute their resources either.

Fourthly, many modern controllers have non-SCSI

command paths that the kernel can use to talk to the chip. For example, RAID controllers such as mpi(4), mpii(4), mfi(4), and ami(4) allow the kernel to query the state of volumes by using a custom firmware command. These custom commands use the same adapter resource that a normal SCSI operation would consume, which further adds to the contention for openings. The traditional solution to this problem was to permanently allocate one of the adapters openings for use with custom commands and have the adapter use a lock around it to avoid further contention on it.

In summary, the SCSI stack was too conservative when distributing adapter resources between possible targets and at the same time extremely optimistic about the number of devices that could appear. The midlayer was very simplistic at scheduling access to an adapters resource by dedicating openings to devices rather than granting access to them based on need.

The first attempt to solve this problem was called NO_CCB.

NO_CCB was an additional return code (the others being COMPLETE, SUCCESSFULLY_QUEUED and TRY_AGAIN_LATER) that an adapter's scsi_cmd handler could return. NO_CCB caused scsi_scsi_cmd to return a new code to its caller. ie, it was a signal from an adapter that it did not have an opening to handle the transfer which the midlayer propagated up to the scsi_scsi_cmd caller. It was named NO_CCB because adapters generally named their internal command handling structure as a "ccb". The total number of openings an adapter gave to attached devices was supposed to be less than or equal to the number of ccbs it had allocated.

SCSI devices (sd(4), etc.) were then modified to look for that return code and requeue, at the head of the queue, the I/O they were attempting to issue to the adapter.

This solution was applied to many adapter drivers and definitely did allow better utilization of an adapters resources. Unfortunately there were a few caveats to the NO_CCB solution that became apparent as it became more widely used.

One was that NO_CCB results caused the midlayer to immediately return to the caller, without retrying the I/O. Callers were now expected to do the retrying. This expectation was correct for the normal block I/O path through strategy routines. It was not met by other paths, which resulted in a much higher rate of failure on these paths than had been previously experienced.

Another issue with NO_CCB was that it did not provide a mechanism to notify code waiting for adapter openings that one had become available. You cannot rely on further calls to the strategy routine as the block layer may be waiting on a block that the device has queued before waking up a process to generate more I/O requests.

The best that a strategy routine could do on receiving NO_CCB, was to schedule a timeout to run in the near future that would trigger queue processing if queue processing was not triggered sooner by a new I/O being queued by the strategy routine.

The same was true of the other paths. In all cases there was no guarantee that any retry of the I/O would succeed, leading to unnecessary retries.

A third issue with NO_CCB was relaxing the openings limit, as several adaptors did in an attempt to make more efficient use of the resources available by relying on the back pressure mechanism of NO_CCB to not lose I/O requests.

It was now possible for a busy device to monopolize the adapter. If you have a system with two disks, and one disk is busy, it will be using a lot of the adapters openings. If it continues to have work queued on it, it will continually reuse the openings it currently has to service that queue. However, if work is queued on that second disk, it will be starved of the adapters openings because the first disk only ever asks itself if there is outstanding work to do.

Lastly, the NO_CCB implementation only dealt with helping devices attached to the midlayer, it did not try to guarantee that the adapters use of its own ccbs was guaranteed. Adapter drivers could have introduced the sleeping semantic for ccbs that the midlayer used while waiting for a devices openings, but this would have led to a lot of code duplication and therefore possible places for bugs and variation

After the replacement of the scsi_scsi_cmd API with the scsi_xs_get family of functions, there was sufficient impetus to keep working and provide a solution to the allocation and distribution problem once and for all.

The requirements for this solution were:

- A device should be able to use all the resources available to it at any point in time.
- When there is contention for an adapters resources a single consumer should not monopolize said resources by reusing them immediately. Another consumer should be given the opportunity to use them instead.
- A consumer must be able to be notified when a resource has become available for it to use. It should not have to schedule a timeout to give it a go later on.
- An adapter must be able to contend alongside SCSI devices for its own resources.
- Devices should continue to respect their own number of openings. This is so devices that limit their own openings (eg, if it knows it is talking to a

buggy device that cannot handle multiple outstanding commands) but still have long queues of I/O cannot erroneously consume adapter resources.

The solution was the development of iopools and the scsi_ioh and scsi_xsh APIs.

```
void scsi_iopool_init(
    struct scsi_iopool *iop,
    void *cookie,
    void *(*io_get)(void *cookie),
    void (*io_put)(void *cookie,
      void *io));
void scsi_iopool_destroy(
    struct scsi_iopool *iop);
void scsi_link_shutdown(
    struct scsi_link *link);

void *scsi_io_get(
    struct scsi_iopool *iop,
    int flags);
void scsi_io_put(
    struct scsi_iopool *iop,
    void *io);

void scsi_ioh_set(
    struct scsi_iohandler *ioh,
    struct scsi_iopool *iop,
    void (*cb)(void *, void *),
    void *cookie);
void scsi_ioh_add(
    struct scsi_iohandler *ioh);
void scsi_ioh_del(
    struct scsi_iohandler *ioh);

void scsi_xsh_set(
    struct scsi_xshandler *xsh,
    struct scsi_link *link,
    void (*cb)(struct scsi_xfer *));
void scsi_xsh_add(
    struct scsi_xshandler *xsh);
void scsi_xsh_del(
    struct scsi_xshandler *xsh);
```

The basic premise of iopools is that adapters provide the midlayer with direct access to their own ccbs so it can manage and schedule access to them. All access to those ccbs must be via the midlayers API to these resources so it can properly arbitrate access to them. In the iopool API an adapter's resource is referred to as an io.

An adapter sets up an iopool by calling scsi_iopool_init. The adapter is responsible for providing the memory used by the iopool, which means that an adapter can provide it as part of its own soft state structure, rather than expecting the iopool API to successfully allocate it itself. The adapter provides a get function for the iopool API to take one of the adapters ios, and a put function for returning that io once a consumer has finished with it. The cookie argument to scsi_iopool_init is provided so the adapter can provide a unique identifier for which pool that is being accessed.

The get and put functions must not sleep as they could possibly be called at interrupt time. The iopool API will call the get function to see if an io is available. If the adapter has no io available then it must simply return NULL from its get handler. Any sleeping an io consumer may need while waiting for an io is done by the iopool API on behalf of the adapter.

This semantic greatly simplifies the adapters code since it generally just has to test if a ccb is available on a free list. It does not have to bother itself with moving ccbs to sleeping processes, iopools takes that responsibility on itself.

Internally an iopool is simply a list of consumers that are waiting for an io. A consumer is represented on that list as a callback function with a cookie thats passed to the callback function along with an io. If an io is returned from the adapters get function then the consumer is dequeued and the callback is called. Once an iopool is initialized it is then available for use by both the adapter and midlayer to grant access to the adapters resources.

An adapter may request an io by calling scsi_io_get. Once it has finished with the io it returns it to the iopool via scsi_io_put. If the caller cannot sleep and no io is available, scsi_io_get will return NULL. If the caller can sleep then a list entry is allocated on the stack and put on the iopools wait queue. The callback for this list entry will simply move the io back to the callers process and wake it up. When an io is returned, the iopool will check if there are any consumers on the wait list, dequeue it and call it with the now available io.

However, if the adapter wants to use an io but is not able to sleep for it, it is able to register its own entry for the iopools wait list with the scsi_ioh API.

The scsi_ioh API was modelled after OpenBSDs timeout(9) API. It is the adapters responsibility to allocate a scsi_iohandler and initialize it using scsi_ioh_set before calling scsi_ioh_add. If the adapter decides at some point that it no longer wants to get an io it must remove it using scsi_ioh_del.

An example of scsi_iohandler usage can be found in the mpi(4) and mpii(4) drivers. They receive notification that a device has been removed in their interrupt handlers. To correctly detach a device they must cancel and outstanding transfers against this device, which requires the use of a ccb so it can issue such a requests to the firmware. Because they are in an interrupt context they cannot sleep while waiting for a ccb, so instead they register a scsi_iohandler so they will get a callback when an io becomes available.

Before iopools became available this code was optimistic about getting hold of a ccb. If a ccb wasn't avail-

able the code simply dropped the event on the floor. The alternative was to implement the functionality now in iopools in each and every driver that wanted reliable access to their own resources.

An adapter provides a device access to an iopool by configuring the template scsi_link structure with it, or it may specify it during the midlayers call to its device probe routine.

This allows a lot of flexibility in how many iopools an adapter may provide to the midlayer for devices to use. For example, an adapter like mpi(4) has a single pool of ccbs for all devices to use, so it may simply configure the template scsi_link structure. However, ATA adapters have a separate pool of ccbs for each port, so they allocate and configure iopools during the call to its probe handler for each target.

SCSI device drivers are more interested in using scsi_xfers rather than an adapters ccbs directly, but we were interested in providing similar semantics for access to these scsi_xfers. The scsi_xshandler API effectively mirrors the scsi_iohandler API, but deals with scsi_xfers instead of ccbs. Internally the scsi_xshandler APIs also properly account for the use of a devices own openings, and implements a wait queue in front of them. Once a scsi_xshandler has got an opening it then goes through the motions of getting an io from the adapter via the scsi_ioh API.

This means that the availability of scsi_xfers for devices to use directly maps to the availability of the adapters io resources. If the adapter is unable to provide an io then the device will wait until one becmes available. One an io becomes available it is associated with the scsi_xfer so that when it is executed by the adapter it is guaranteed that it will succeed. Once again we return to the semantic where if you were allocated an opening you were guaranteed that the adapter has the resources to actually execute it.

The scsi_xs_get API was rewritten to use scsi_xshandlers internally if it needed to sleep while waiting for resources to become available.

The SCSI device drivers were rewritten to use a single scsi_xshandler to service I/O requests. When the block layer queues an I/O, that xsh is added. If an opening or adapter io is not available for it, the I/O will remain on the devices queue and the block layer will be able to queue further I/O. Further calls to scsi_xsh_add will not affect the handlers current position on the queues.

As soon as the resources become available to the I/O paths scsi_xshandler, the callback that dequeues the I/O and prepares it for a call to scsi_xs_exec. Once that I/O has been executed, the driver checks to see if there are further I/O requests pending. If so it calls scsi_xsh_add again to say it has more work to do.

Because a call to scsi_xsh_add puts a handler on the end of the wait list, multiple devices servicing their own I/O queues will continually put themselves behind the other. This effectively means all the devices schedule themselves and cooperate so they have round-robin access to the adapters resources.

A single device will always be putting its xsh on an empty list, so it will immediately get access to any available resources, but as soon as another device wants access to the same resources they will get round-robin access.

iopools successfully solve the problems with the command allocation, distribution and scheduling that the original midlayer featured, and the problems that became apparent with NO_CCB.

The scsi_shutdown_link function is provided to wake up scsi_xs_get calls that are sleeping while waiting for resources on a device that is going away. If scsi_xs_get is sleeping and a device goes away, a call to scsi_shutdown_link will cause it to get woken up and return NULL. Therefore it is important to always check for a NULL return value from scsi_xs_get, even if you have process context and always expect a scsi_xfer.

Similarly, scsi_iopool_destroy performs the same cleanup for any scsi_io_get calls that are waiting for an io.

8 Fine Grained Locking

Because we were effectively rewriting the SCSI command handling and scheduling, it was relatively easy to write the code using fine grain locks. The midlayer and device drivers are largely SMP safe now along with several adapter drivers. However, the effectiveness of these changes is hard to test because the OpenBSD kernel is still under a Big Giant Lock.

9 Hotplug

Recent effort has gone into providing APIs for adapters to call when luns, devices, or entire busses of devices go away, rather than having to implement that functionality themselves (and inconsistently):

```
int scsi_probe_bus(
    struct scsibus_softc *link);
int scsi_probe_target(
    struct scsibus_softc *link,
    int target);
int scsi_probe_lun(
    struct scsibus_softc *link,
    int target, int lun);

int scsi_detach_bus(
    struct scsibus_softc *link,
    int flags);
```

```
int scsi_detach_target(
    struct scsibus_softc *link,
    int target, int flags);
int scsi_detach_lun(
    struct scsibus_softc *link,
    int target, int lun, int flags);

int scsi_req_probe(
    struct scsibus_softc *link,
    int target, int lun);
int scsi_req_detach(
    struct scsibus_softc * link,
    int target, int lun, int flags);

int scsi_activate(
    struct scsibus_softc *link,
    int target, int lun, int act);
```

The scsi_probe_, scsi_detach_ and scsi_activate functions are simply wrappers around the kernels own device autoconf routines, except that they also update the midlayer's state. Just like the kernels attach and detach routines, you must be in a process context to call the scsi_attach_ or scsi_detach_ routines.

The scsi_req functions provide a way for adapters that detect a topology change in an interrupt context to request a call to scsi_attach_ or scsi_detach_ from a process context later on. Currently these attach and detach requests are run out of the systems workq.

The current midlayer requires that a device be deactivated by a call to scsi_activate with DVACT_DEACTIVATE passed as the activity before the call to scsi_detach.

The SCSI layer has long supported the ability to add and remove devices at runtime but was quite optimistic about the state of in flight transactions when removing a device. When detaching a device the midlayer now waits until all the outstanding operations on the device have completed before allowing the softwares state to be deallocated.

Once upon a time the adapter and midlayer would grant devices a number of outstanding operations it was allowed to queue, referred to as openings. The only accounting done for operations on a device was to decrement the openings as they were used until it hit zero, and as operations completed the number of openings was incremented again. The problem with this is you don't know how many openings you had so you cannot tell if you're using any or not. The midlayer was changed to leave the openings value alone and count the number of pending operations separately. When detaching the driver the midlayer will sleep on a device until all the outstanding commands have completed, thereby avoiding a lot of potential use after frees.

This does put a lot of responsibility onto adapter drivers to clean up any commands associated with a device thats going away. Previously it was possible for the transfers currently on the hardware to a detached device to stay in that state and never complete, the device driver was happily able to detach and the system would continue running.

In this situation the hardware must return all the commands that are outstanding for the device before the driver is able to detach. Because the attach and detach routines are generally run out of the systems workq, unless all outstanding operations have been completed the system will unable to continue processing tasks in that same queue.

10 Sparse LUNs

The SCSI midlayer used to keep pointers to the devices that were attached to it in an array called sc_link that was sized by the number of targets and luns the adapter could support. For example, traditional SCSI controllers supported a bus width of 8 with 8 possible luns at each target, so the midlayer would allocate a two dimensional array of 8 by 8 pointers for sc_link. If a device was found at target 1, lun 0, a pointer to it would be stored in sc_link[1][0], target 6, lun 5 would go in sc_link[6][5].

However, time has moved forward and bus widths on some adapters (especially)SAS and Fibre Channel) can be as high as 512, and in modern SCSI standards the lun is a 64bit value. It is impractical to allocate an array of pointers of this size since it would exhaust the physical memory of a machine, and it is rather pointless as busses of that size are relatively sparsely populated.

The midlayer was modified to store the location of child devices in a list rather than the array described above. If something needs to check if a device exists at a particular address on the bus it simply iterates over this list. This is a very rare operation and therefore doesn't need to be as highly optimized as the array implementation was.

11 Remaining Work

Now that the changes described above are part of the OpenBSD SCSI stack, the main work remaining is updating all 80 SCSI hardware adapter drivers to properly use the new mechanism.

At the moment only a few of the adapter drivers have been updated to use all of the new features. There is a lot of testing on interesting hardware remaining, and volunteers are actively sought to help.

12 Results

The OpenBSD SCSI stack is significantly more robust, scalable and maintainable as a result of the changes implemented over the last two years.

As part of the effort of updating the hardware adapter drivers, an interesting accident of history was brought to light, This was the decision to emulate SCSI on top of hardware RAID controllers. Generally other operating systems implement a separate (but minimal) stack for hardware RAID controllers that allows the adapter to process block I/O requests directly, but OpenBSD has always emulated SCSI on top of these controllers. It is our belief that this has been beneficial since the improvements to the SCSI midlayer have avoided the need to port similar improvements to an alternative stack for RAID devices. It is this belief that has lead to the development of a SCSI to ATA translation layer so ATA hardware can also inherit from improvements to the SCSI subsystem.

13 Acknowledgments

We would like to thank the OpenBSD developer and user communities for their support in implementing and testing these SCSI changes.

In particular we want to acknowledge the work done by Matthew Dempsky (matthew@openbsd.org) who, from a standing start, made immediate and significant contributions to this work.

14 Availability

All the changes made the to SCSI subsystem, ie, its midlayer, SCSI device drivers, and adapter drivers, have been committed and are a part of OpenBSD right now.