USENIX Association

Proceedings of BSDCon '03

San Mateo, CA, USA
September 8–12, 2003

The WHBA Project: Experiences "deeply embedding" NetBSD

Jason R. Thorpe
Allen K. Briggs
Wasabi Systems, Inc.

Abstract

Traditionally, use of BSD in an embedded application has been limited to "stand-alone" environments such as server appliances, wireless access points, and the control plane of routing equipment. Recently, Wasabi Systems was given the opportunity to use NetBSD in a more "deeply embedded" application, as the software on a PCI storage adapter card. This paper will introduce our specific application and describe the requirements of the storage adapter environment, the challenges presented by this type of environment, and the solutions we have developed to address these challenges, along with the results of our effort and some outstanding issues that warrant further study.

1. Introduction

BSD has been used in embedded systems for several years in applications such as server appliances and network infrastructure equipment. These applications tend to take advantage of BSD's traditional strengths, namely networking and file systems. However, these applications present environments not all that different from the traditional stand-alone UNIX system.

Recently, Wasabi Systems, Inc. was given the opportunity to use NetBSD as the firmware on a PCI storage adapter card. This "deeply embedded" application is very different from traditional BSD embedded applications.

The goal of the WHBA (Wasabi Host Bus Adapter) project is to produce a design for a PCI add-in card that can be used to offload a variety of processing chores from the host system, which is typically a server. The initial application is that of an iSCSI target HBA. An iSCSI target HBA performs TCP/IP, IPsec, and iSCSI protocol processing, then passes the SCSI command and data payload up to the host system. Once the host system has processed the SCSI command, the HBA encapsulates the status code and any payload or sense data as necessary, and then sends the result back to the initiator. This is very similar to a parallel SCSI or Fibre Channel HBA operating in target mode.

Other potential applications for the WHBA include iSCSI initiator, IPsec offload processor, RDMA processor, TCP offload engine (TOE), and RAID controller.

2. WHBA requirements

The requirements of an HBA application are much different from those in a traditional embedded BSD environment. Here we will discuss the requirements of our application, as well as the general requirements and challenges of the HBA environment.

2.1. Overview of iSCSI

iSCSI is a transport layer for the Small Computer System Interface command set that uses TCP/IP to connect initiators and targets over standard IP networks. There are several usage models for iSCSI, including:

- Expandable direct-attached storage for consumer and SoHo environments
- At-distance connectivity to disk mirrors and tape volumes
- IP-based Storage Area Networks (iSANs)

Each of these usage models has different characteristics and performance requirements (described below).

The main advantage of iSCSI is its ability to run on any IP network, including LANs and WANs. Since IP-capable networks are ubiquitous, the cost of deploying iSCSI is low, especially compared to the cost of Fibre Channel; in addition to the cost savings from not having to deploy a different type of network (Fibre Channel), every server and client system on the market today comes with an Ethernet port. In environments where a SAN is not needed, iSCSI still has the advantage of not requiring the storage to be located within a few meters of the system to which it is attached.

iSCSI's basic unit for data transfer is a *protocol data unit*, or PDU [1]. There are several PDU types, but they all share the same basic structure:

- A *basic header segment* (BHS) that contains the PDU's opcode, the LUN, task tag, CDB, or other opcode-specific fields
- Zero or more *additional header segments* (AHS)
- An optional *header digest* (CRC32C)
- An optional *data segment*
- An optional *data digest* (CRC32C)

The maximum PDU size is negotiated when an iSCSI session is established. All data transfers are broken down into one or more PDUs. Since a common PDU size is 8KB, it is likely that any given SCSI data transfer will require multiple PDUs.

In addition to the overhead of encapsulating into or de-encapsulating data out of PDUs, the iSCSI transport has an optional error recovery mechanism, known as *markers* [2], that requires additional processing. Markers are inserted into the data stream at regular intervals, and allow an iSCSI target or initiator to re-synchronize the stream in the event a PDU with an invalid header digest is received (the length fields in the basic header segment can not be used in this case). These markers are orthogonal to PDUs, and thus are not accounted for in length fields or the header or data digests.

The combination of TCP/IP processing, PDU encapsulation and de-encapsulation, CRC32C generation and verification, and marker insertion and removal make a compelling argument for including an offload component in an iSCSI solution.

2.2. Usage model requirements

Each of iSCSI's usage models has its own set of characteristics and performance requirements. There are several variables to be factored in, including:

- Target audience and what they expect from a storage system.
- Acceptable cost of the storage system.
- Need for remote or local connectivity.

In order to understand the requirements of the HBA, it is helpful to have a basic understanding of some of iSCSI's usage models.

2.2.1. Consumer / SoHo

As homes around the world become "wired", the need for high-capacity storage increases. Consider the amount of space required to store music, pictures and movies, etc. Not only is the typical consumer PC limited in the amount of disk drives that can be directly attached, but the typical home PC user is not a computer hardware expert, and usually does not wish to open up his or her computer to install new disk drives.

A similar situation exists with the SoHo environment. Data stored on an office storage system is often business-critical, and tends to be stored for long periods of time. Therefore, as time goes on, more storage is needed. Because of the business-critical nature of this data, the downtime and risk associated with migrating to new, higher-capacity disks is unacceptable.

From the perspective of storage, consumer and SoHo environments are not generally high-performance environments. Low-end PCs and laptops, which are generally tuned for reduced power consumption, are common. The expectation of the user is that their files and pictures will be available on-demand, and their music and movies will play smoothly. The performance achievable with 100Mb/s Ethernet is adequate for audio applications. Gigabit Ethernet is becoming increasingly common on consumer PCs, and the iSCSI performance that can be achieved on Gigabit will likely be comparable to, if not better than, what the user would experience with disk drives directly attached to the computer.

In this usage model, cost is a dominant factor. In order for an offload solution to work in this market, a small number of low-cost parts should be used on the HBA.

2.2.2. At-distance backup

At-distance backup is a scenario where storage resides in one physical location and the capacity to back up that storage resides in another, possibly far away, location. There are a number of reasons why this strategy might be used, including cost-effectiveness (backup systems can be expensive) and disaster recovery (losing your data and its backup in the same fire would render the backup useless).

This environment is constrained by the characteristics of wide-area networks. The user will not expect low latency or high throughput access to storage, but will expect the data to stream effectively, and for software on the initiator (client) side to behave as though the storage were local, to the extent possible [3].

This usage model is less concerned with cost than the low-end consumer/SoHo market. However, small businesses are likely to be a major part of this market, and so keeping the cost down is still important.

2.2.3. Storage Area Networks

Storage Area Networks, or SANs, are typically deployed in environments where multiple systems require access to pooled storage, or where redundant connections to storage are required in order to ensure uptime.

SANs are currently dominated by Fibre Channel. However, Fibre Channel has a high total cost of ownership [4]:

- Fibre Channel is not commonly used for IP networks, and thus is always added cost.
- Fibre Channel equipment is expensive.
- IT staff generally needs additional training in order to manage and maintain a Fibre Channel SAN.

Because iSCSI runs over any IP network, it can utilize existing networking infrastructure and expertise, thus reducing the total cost of ownership.

Even though Fibre Channel is currently faster than Gigabit Ethernet (Fibre Channel is available in 1Gb/s and 2Gb/s speeds), the reduced cost of iSCSI still makes a compelling argument in favor of its use. However, users of Fibre Channel have come to expect low latency and high throughput from their SANs, largely due to the fact that Fibre Channel is a fairly low-overhead protocol and Fibre Channel HBAs offload all of the necessary processing.

Even an expensive iSCSI HBA is likely to be less expensive than a Fibre Channel HBA; current iSCSI HBAs retail for approximately $500US, and Fibre Channel HBAs retail for $800US and up. Combined with the cost savings associated with the use of commodity Gigabit Ethernet networking equipment, iSCSI presents a much lower-cost alternative for SAN deployment. As such, the cost of an iSCSI HBA is less of a factor in this usage model.

2.3. Host communication

An HBA plugs into a host system and must communicate with it. HBA applications based on intelligent I/O processors generally use a *messaging unit* and a DMA controller to implement host communication.

A messaging unit is a device in an I/O processor that provides support for notifying the host and the HBA when new messages are available. A typical messaging unit is comprised of two or more *doorbell* registers and two or more sets of *queue pointers*.

The queue pointers in the messaging unit are used to implement a ring of message descriptors. These message descriptors reside in host memory. The DMA controller is used to copy the host-resident descriptors to HBA local memory. The DMA controller is also used to move payload data between the HBA and host.

We looked at several examples of message passing APIs used by storage controllers. Most were either too simplistic (early BusLogic and Adaptec), too complicated to implement in a short period of time (I2O), or too tied to their specific hardware platform and application (recent Adaptec). Since one of our goals was to produce a solution that could offload different types of workloads, we decided to borrow ideas from the early BusLogic [5] and the I2O [6] APIs and create our own.

The WHBA message passing API is based on two circular queues; one for host->HBA messages and one for HBA->host messages. The general data flow is that the host sends a message to the HBA and at some time in the future, the HBA responds to that message, although there are a few exceptions to this rule. Messages and replies use the same data format.

Each message is 64 bytes long and begins with a standard header:

```
uint32_t msgid_high;
uint32_t msgid_low;
uint16_t msg_type;
uint16_t hdr_reserved;
```

The `msgid_high` and `msgid_low` fields are not processed by the WHBA, but are copied into the corresponding fields of message replies, where they may be used by the host's driver software to identify the context associated with original message sent by the host to the HBA. The `msg_type` field defines the remaining data fields after the header. The `hdr_reserved` is reserved for future expansion.

There are currently four groups of messages defined in the WHBA message passing API:

- General messages
- iSCSI common messages
- iSCSI initiator messages
- iSCSI target messages

Some of these messages are self-contained, and others contain pointers to additional message data. This data is then transferred using the DMA controller from the host's memory to the HBA's memory. Either the message or the additional message data may also contain pointers to payload data associated with the message.

2.4. HBA hardware

Two hardware platforms, both based on Intel's XScale™ processor core, were used during the development of the WHBA:

- The Intel IQ80321 reference platform for the i80321 I/O Processor
- The ADI Engineering *i*/HBA (which was designed and built expressly for the purpose of demonstrating the WHBA)

The IQ80321 was used for early prototyping and development. This board consists of the i80321 I/O Processor, an Intel i82544 Gigabit Ethernet controller, an IBM PCIX-PCIX bridge, a DIMM slot, 8MB of flash memory, and a UART. The i80321 contains the XScale™ core, PCI controller, DMA controller, timers, and an I2O-compatible messaging unit.

The ADI *i*/HBA was the target platform for the WHBA. The *i*/HBA consists of an i80200 CPU, an Intel i82545 single-port or an i82546 dual-port Gigabit Ethernet controller, an Intel i21555 non-transparent

PCI-PCI bridge, 128MB or 512MB of on-board SDRAM, 8MB of flash, and a Xilinx Virtex-II FPGA containing the companion chip, known as HBACC.

The HBACC is a descendant of the BECC ("Big Endian Companion Chip") used on ADI's BRH XScale™ development platform. The HBACC contains a high-performance memory controller, the PCI controller, a DMA controller with CRC32C offload capability, timers, and a UART. The i21555 non-transparent PCI-PCI bridge contains an I2O-compatible messaging unit.

Initial tests on each of these platforms showed raw TCP throughput of 300-400Mb/s, with the IQ80321 being on the low end of that range and the *i*/HBA being on the high end. Since a high-performance host could easily saturate either of these platforms, it is necessary to utilize the hardware available on the WHBA as effectively as possible.

The XScale™ processor is an implementation of the ARM5TE architecture. This processor has an MMU and virtually-indexed/virtually-tagged instruction and data caches. While the MMU provides some support for address space identifiers, the semantics of these identifiers and the requirements of the application preclude their use. As a result, process context switches carry a significant penalty: changing address spaces requires a complete cache clean and TLB invalidation.

In addition to context switch overhead, the memory buffer used to stage data to and from the host must be managed. The means of moving data in and out of this buffer is through a DMA controller, so extracting the physical address of any given region of the buffer is a time-critical operation.

Finally, the latency of application notification when a messaging unit doorbell or queue pointer register is written is critical.

2.5. System start-up and shutdown

System start-up and shutdown in the HBA environment are very different from start-up and shutdown in the desktop and server environments where BSD is traditionally found. These differences present some interesting challenges.

Most BSD systems rely on the system's firmware to bootstrap the hardware and read in a loader program from a storage device, such as disk or CompactFlash. In the HBA environment, there are no disks. While our prototyping systems had the RedBoot [7] bootstrap and debug environment in ROM, it is unlikely that any final product sold to end-users will include it. This puts the onus of hardware bootstrap entirely on the operating system, which must be loaded into ROM at the reset vector.

Furthermore, BSD systems have historically not been very fond of unexpected shutdowns, which are often associated with long restart times (due to file system checks) and lost data (due to data not being flushed out to disk, as would happen in a clean shutdown). In contrast, unexpected shutdowns are a part of normal operation in an HBA environment. These can be caused by a number of events on the host system including reboot, driver load/unload, and power management activity.

3. The Wasabi Embedded Programming Environment™

In order to meet the challenges before us, we concluded a traditional BSD operating model would not be appropriate for the following reasons:

- A good portion of our application needs to reside in the kernel in order to be able to interact with messaging and DMA hardware effectively.
- A large chunk of physically contiguous SDRAM is required for efficiency.
- We needed to address the start-up and shutdown issues associated with the HBA environment.

We observed that by placing the entire application into the kernel, user space became completely superfluous. Eliminating support for user space would free up a large chunk of space in the virtual address map that we could be used to contiguously map SDRAM. With no user-land, there is little need for a root file system, and eliminating the root file system goes a long way towards addressing start-up and shutdown issues.

Unfortunately, our existing iSCSI software was a pthreads application that ran in user space. What we needed was an application environment that would allow an application to run either in user space or the kernel. So, we developed one, and called it the *Wasabi Embedded Programming Environment*™, or WEPE for short.

WEPE is actually a combination of three things:

- An API for applications that provides portability to both user space and kernel WEPE environments
- A configuration management framework that eliminates the need for configuration files
- A set of modifications to the NetBSD kernel that removes support for processes running in user space and provides a simple, extensible debugging console

While there are many applications that can benefit from the simplified single address space model that WEPE provides in the kernel environment, it is important to recognize that there are significant trade-offs associated with using this model.

Perhaps the most significant of these is the lack of memory protection; a badly behaving application can corrupt data structures in another application.

Secondly, since all applications are linked together into a single ELF image, applications, as well as the kernel itself, must be careful with regard to symbol namespace.

Finally, the WEPE API must contend with the fact that the semantics of calling system facilities is quite different in the user space and kernel environments, and global state, such as `errno` cannot be used in the kernel environment. While thread-specific data can be used to work around many of the awkward POSIX API issues, such schemes can have problems with efficiency that vary from platform to platform. For this reason, WEPE does not provide a POSIX API, requiring that applications be ported to the WEPE API.

3.1. The WEPE API

In addition to the trade-offs listed above, there are some functions available in the user space environment that do not map directly to equivalent functions in the kernel. It is therefore necessary to provide applications with a compile-time indication of whether they will be run in user space or in the kernel environment. This indication is provided by the presence of a C preprocessor macro:

- `_WEPE_MODEL_SINGLE` indicating the single address space kernel environment
- `_WEPE_MODEL_VIRTUAL` indicating a full user space virtual memory environment

WEPE API elements are logically grouped by function, and have naming rules that help to identify their function and grouping. For example, functions whose names begin with `wepe_sys_` map to system calls. The following sub-sections provide an overview of these functional groups.

3.1.1. File I/O functions

WEPE file I/O functions are made available by including the `<wepe_fileio.h>` header file. This functional group provides WEPE versions of several Unix file-related system calls, such as *open(2)*, *read(2)*, and *close(2)*. All of these functions return 0 on success or an error code to indicate the reason for failure. Functions that would return a file descriptor or a byte count in a POSIX environment take an additional pointer to that returned value in WEPE.

3.1.2. Socket I/O functions

WEPE socket I/O functions are made available by including the `<wepe_sockio.h>` header file. This functional group provides WEPE versions of several Unix socket-related system calls, such as *socket(2)*, *bind(2)*, and *listen(2)*. Like the file I/O group, all of these functions return 0 on success or an error code to indicate the reason for failure, and functions that would return a file descriptor in a POSIX environment take an additional pointer to that returned value in WEPE.

As in POSIX, any file descriptor returned by a WEPE socket I/O routine may be used with file I/O functions.

3.1.3. Thread functions

WEPE thread functions are made available by including the `<wepe_thread.h>` header file. This functional group includes a threads API similar to, but simpler than, POSIX threads, including thread creation, mutexes, read/write locks, and condition variables. The thread group also includes some basic process management functions.

3.1.4. Networking functions

WEPE networking functions are made available by including the `<wepe_net.h>` header file. This functional group provides WEPE versions of several networking-related functions, such as *getifaddrs(3)*, *getaddrinfo(3)*, and *getnameinfo(3)*. In the kernel environment, this group also provides function calls that allow applications to configure network interfaces, set routing table entries, and access the kernel's DHCP client.

3.2. WEPE configuration management

While the configuration management framework used by WEPE is actually independent from WEPE, it bears mentioning in this context since WEPE relies on its functionality. The main problems it attempts to solve are as follows:

- In embedded systems, management and storage of multiple configuration files can be problematic, especially if there is no file system in which to store multiple files.
- Configuration files have differing syntaxes, and thus make it difficult to provide a consistent configuration interface.

The MIB model used by the BSD *sysctl(8)* tool is very attractive from a consistency point of view.

However, it does not define the storage mechanism for the configuration data.

The solution that WEPE employs, both in the kernel and in user space environments, is known as *wctdb* ("Wasabi Config Tool Database"). The database consists of simple key-value pairs. The keys are hierarchical MIB names, similar to those found in the *sysctl(8)* interface. The API is made available by including the `<wctdb.h>` header file.

The implementation of the back-end database is completely hidden from the user of the API. In user space, *btree(3)* is used. In the kernel environment, a simple key-value list designed for storage in memory-mapped flash is used.

3.3. WEPE kernel modifications

The WEPE kernel modifications fall into four categories:

- Replacement of several functions in the kernel proper in order to remove the user address space
- Implementation of the WEPE API
- Implementation of the WEPE debug console **kshell**
- Kernel environment debugging tools

In this section we will discuss the replacement functions, the **kshell** debug console, and the kernel environment debugging tools.

3.3.1. Replacement functions

One of our goals was to make WEPE as non-invasive to the rest of the kernel as possible. However, there are several places in the kernel proper that access the user address space, using routines such as `copyin()` and `copyout()`. By replacing these functions, we are able to use all of the system call functions in the kernel to provide WEPE API support without modifying them.

There is a slight problem with this approach, however. The semantics of these user address space access functions dictate that the data be copied. This is not strictly necessary in the kernel environment, since all tasks share a single address space. Unfortunately, many parts of the kernel assume that the arguments provided in a system call are "owned" by the code that implements the call, and thus may modify the arguments while executing the system call. For this reason, we must accept a certain amount of this data copying.

There is one other function that is replaced, although for a different purpose. The `start_init()` routine, which normally takes care of starting *init(8)*, is replaced with the entry point for the **kshell** debug console.

3.3.2. The kshell debug console

The **kshell** debug console provides a command-line interface to the WEPE environment. Its primary purpose is for debugging, but it is extensible and could be used to provide a command-line based management interface in an appliance application. The **kshell** is also responsible for initializing all of the applications that are linked into the kernel image.

The **kshell** provides several API components available only in the kernel environment. These API components are made available by including the `<wepe_kshell.h>` header file.

Kshell takes the place of *init(8)* as process #1. When it starts, its first task is to initialize the console device in a way that mimics the behavior of a normal tty; the initial (and only) session is created, the console device is set as its controlling terminal, and backspace is set to ^*H*.

Once the console is initialized, **kshell** calls the initialization function provided by each application linked into the image. Applications register themselves using a C preprocessor macro: `KSHELL_APPINIT()`. This macro causes a pointer to the application's initialization function to be placed into a special section in the final ELF image. The **kshell** then traverses this special section, calling through each function pointer. This scheme not only allows applications to be linked into the image easily, but also facilitates binary-only distributions to licensees. Applications may perform any start-up tasks they require using this mechanism, including creating additional processes and/or threads.

Once all of the application initialization tasks have been performed, the **kshell** enters a command loop. The following built-in commands are provided: *about*, *help*, *ddb*, *reboot*, *halt*, and *poweroff*, the latter three being the equivalent of the user space commands of the same names.

The **kshell** also allows applications linked into the image to add commands to the command line interface. Applications register these commands using the `KSHELL_COMMAND_DECL()` C preprocessor macro. This macro causes a pointer to a structure describing the command to be placed into a special section in the final ELF image. This section is consulted by the **kshell** command loop when a command is entered on the command line interface. The command description includes the command name, a usage string, and a pointer to the function that implements the command.

The **kshell** exports some support functions that help application programmers to implement command extensions. These support functions include a *getopt(3)*-like routine and a paged-output routine.

3.3.3. Kernel environment debugging tools

Debugging tools for Unix systems are generally used in a user space environment; if your program crashes, you run *gdb(1)* on it to figure out why. Similarly, if you want to profile your application, you compile it for profiling, run it, and use *gprof(1)* to analyze the profiling data.

NetBSD also provides support for using debugging and profiling tools on the kernel: in-kernel (DDB) and remote (KGDB) debuggers are available, and the standard BSD kernel profiling feature *kgmon(8)* is supported.

Unfortunately, the standard BSD debugging and profiling tools were insufficient for the WEPE kernel environment for the following reasons:

- DDB is not a source-level debugger, and thus has limited usefulness for debugging a complex application.
- KGDB requires a dedicated serial port; the WHBA target hardware only has one, which is used for the console.
- Kgmon requires a user space application to control kernel profiling and to extract the profile data.

The debugging problem was addressed by enhancing the in-kernel DDB debugger to interact better with KGDB. When an event that traps into DDB occurs (either a special sequence on the console, the *ddb* command in **kshell**, or a fatal trap or kernel panic), DDB is entered as normal. If the user wishes to use KGDB to debug the problem, the new *kgdb* command is entered at the DDB command prompt. This causes the console port to become the KGDB port, disabling normal console output. The user may now interact with the application using KGDB. When the user is finished using KGDB, he or she simply disconnects the debugger and the port returns to console mode, presenting the DDB command prompt once again. At this point, any DDB command, including *continue*, may be issued.

The profiling problem required a more complex solution. We started by implementing a *kgmon* application for the kernel environment. This optional application is linked into the **kshell** command line interface using the standard **kshell** extension facilities, and includes the same functionality as the user space *kgmon(8)* utility. This allowed us to perform all of the necessary control operations for kernel profiling.

Once we were able to control kernel profiling, we needed a way to examine the profile data. Since our *gprof(1)* tool is cross-capable, we decided to add a TFTP client to the kernel in order to dump the profiling data over the network.

4. Implementation of the HBA application

The iSCSI application for the HBA breaks down into several components: startup code, the PCI interface, the iSCSI engine, and some glue that stitches the pieces together.

4.1. HBA startup

The start-up component of the HBA application varies from platform to platform. For example, the IQ80321 platform requires that bootstrap software configure the memory controller so that SDRAM is available, whereas the *i*/HBA's HBACC performs this task.

The primary challenge of HBA start-up is that the application requires certain PCI resources that must be configured by the host system. Meanwhile, the host must not configure the PCI resources until the HBA has completed its initial setup, such as setting of the size of the PCI base address registers that the host must program. This issue is handled differently on our two hardware platforms.

The IQ80321 uses a semi-transparent PCI-X bridge and the host system programs the i80321's PCI-X configuration space directly. The bridge may be set into a mode such that configuration cycles issued from the primary (host) side of the bridge will be locked out and retried by the host until software on the HBA releases the bridge. However, if the bridge locks out configuration cycles from the host system for too long, the host system may fail its boot-time configuration and power-on self-test (POST). Since the i80321's memory controller must be programmed and an ECC scrub performed, PCI BAR sizes must be programmed by reset vector code. Fortunately, the RedBoot bootstrap environment for this board configures the i80321's PCI BARs in a way that is compatible with our HBA application, which allowed us to skip this task in our start-up code.

The *i*/HBA, on the other hand, uses the i21555 non-transparent PCI bridge. This bridge is similarly configured to lock out PCI configuration cycles, allowing our application code to configure the bridge. Fortunately, the *i*/HBA does not require an ECC scrub, and thus can start the HBA software rather quickly.

The HBA software is stored in flash memory on both of the platforms we used. In order to minimize the flash footprint of the HBA software, a special loader called *gzboot* was developed to allow the software to be stored in compressed form. *Gzboot* can be prepended onto a flat binary image of a NetBSD kernel that has been compressed with *gzip(1)* and the resulting image can be written to flash memory. When started, *gzboot*

will then decompress the kernel into a pre-determined location in SDRAM and jump to it. Code to bootstrap SDRAM and configure PCI BARs can be placed into *gzboot*'s start-up code.

During development, the RedBoot bootstrap environment was used on both boards to provide initial start-up support. On the IQ80321, a RedBoot script is used to perform an unattended start-up of the HBA software. On the *i*/HBA, the RedBoot reset vector code will automatically jump to the HBA software written into flash, but to aid development, it will drop into the RedBoot command prompt if a test point on the board is shorted to ground at hardware reset.

4.2. HBA PCI interface

As mentioned, the PCI configuration is somewhat different between the *i*/HBA and the IQ80321. For development, though, we wished to minimize the differences visible to the application layer. In this section, we'll go into a bit more detail on how we work with the PCI interface on each card and how we create an abstraction to keep the application code itself away from the differences.

4.2.1. IQ80321 PCI interface

The host system sees the IQ80321 as two devices: the IBM PCI-X bridge and the i80321 I/O processor (on the secondary side of the bridge). Depending on the configuration of the board, the host may also see other PCI devices on the secondary side of the PCI-X bridge.

The IQ80321 has four inbound PCI BARs. The first BAR is set to its minimum size, 4KB, and maps the i80321's Messaging Unit (MU) into the host's PCI space. The next two BARs are disabled. The last BAR maps the IQ80321's local memory onto the PCI bus, providing a window for PCI devices under the i80321's control to access the entire range of the IQ80321's SDRAM.

Since the PCI-X bridge on the IQ80321 is semi-transparent, it is possible for the host system to access the PCI devices under the i80321's control. The IQ80321 has a set of switches that allow the on-board PCI devices to be "hidden" (by making its IDSEL line unavailable) from the host.

The DMA controller on the i80321 can address the PCI bus directly; no address translation is required. This means that the HBA software can directly use the PCI addresses provided by the host system. Unfortunately, the DMA controller does not support true scatter-gather; there is a single chain of buffer descriptors, each descriptor containing a source address, destination address, and length. Since the length applies to both the source and destination in that descriptor, the source and destination DMA segments must map 1:1. It is not very likely that this would ever happen with host-provided scatter-gather lists during normal operation, so it is necessary to buffer data to/from the PCI bus in a physically contiguous memory region.

4.2.2. *i*/HBA PCI interface

The host sees the *i*/HBA as a single device: the i21555 PCI bridge. On the secondary side of the bridge is a completely independent PCI bus with its own address space. There are mapping windows on both the primary side and secondary side of the bridge that allow devices on either side to access devices on the other. Transactions that pass through the bridge go through an address translation process.

While the DMA controller on the HBACC can address the PCI bus directly (like the i80321), the HBACC's DMA controller does not operate on the same address space as the host (unlike the i80321). This requires the HBA software to translate PCI addresses provided by the host.

Further complicating PCI access on the *i*/HBA is the fact that the i21555's upstream (secondary to primary) memory window is not large enough to map all of the memory that might be installed on the host system. The i21555 provides a look-up table that sub-divides the upstream memory window into *pages* and remaps those individual pages onto the host's PCI address space. Note that the HBACC DMA controller and the bridge's look-up tables are distinct, which requires us to put some knowledge of the look-up tables into the HBA application in the *i*/HBA case.

One fixed BAR on the i21555 is part of the first downstream memory window. The first 4KB of that BAR maps the local registers on the i21555, and it is through this BAR that the host accesses the doorbell registers to signal the *i*/HBA. Another 4KB BAR is configured for access to a page of HBA local RAM that is used for extra register space, as there are only a few general purpose "scratch" registers on the i21555. The upstream BARs and look-up tables are configured on the fly.

Wasabi was consulted numerous times during the design of the HBACC, and as a result, we had a fair amount of input on the design of the HBACC DMA controller. The HBACC DMA controller fully supports independent scatter-gather lists for both the source and destination, and thus does not require a physically contiguous staging area. The DMA controller also supports the iSCSI CRC32C algorithm, and can compute a full or partial CRC32C when copying data or in a read-only mode (where no destination is used).

4.2.3. Messaging Unit API

While the messaging units on our two hardware platforms are both I2O-compatible, the interface is somewhat different both from the host's and HBA software's perspective. In order to minimize the impact of these hardware differences on the HBA software, we developed a general framework for interfacing with messaging units called **muapi**.

Muapi is a session-oriented mechanism for sending and receiving messages through different types of messaging portals found in a messaging unit. The basic architecture of **muapi** consists of *back-ends*, *portals*, *sessions*, and *messages*. Back-ends present portals to the **muapi** framework. Each portal has an associated data type:

- *Doorbell* portals can indicate a small number of specific events, such as "new message available" or "error occurred", represented by a bit mask.
- *Mailbox* portals can pass single integer messages. Mailboxes can be used to implement handshake protocols for initialization, among other things
- *Message queue* portals consist of *head* and *tail* pointers. The message producer advances the head pointer, and the message consumer advances the tail pointer. The pointers are indices into a ring of message buffers. These message buffers are usually accessed using the DMA controller.

Clients of the API create a session associated with a specific portal. Clients send messages by calling `muapi_put_message()`. Clients may receive messages either asynchronously (using a callback) or synchronously (using `muapi_get_message()` or `muapi_sync()`, which waits for the callback to be invoked). Once a client has processed the message, it acknowledges the messaging unit by calling the `muapi_release_message()` function, which may clear a mailbox or advance a tail pointer.

4.2.4. Data mover API

As with the messaging unit, our two hardware platforms used different DMA controllers with different programming interfaces. We also wanted a single framework capable of moving data between SDRAM and the PCI bus, moving data from one region of SDRAM to another, and handling CRC32C offload.

The solution used in the HBA application is called **dmover**. **Dmover** provides a session-oriented mechanism for queuing data movement operations. The API is fully asynchronous, and clients are notified by a callback when an operation completes.

The basic architecture of **dmover** consists of *back-ends*, *algorithms*, *sessions*, and *requests*. Back-ends present algorithms to the *dmover* framework. Clients of the API create a session to perform a specific algorithm, at which times a backend is chosen and assigned to the session. The client then allocates requests, fills in the necessary information, and queues them with the session.

In order to provide maximum future expandability of the interface, algorithms are named using strings. Back-ends and clients that wish to interoperate must agree on a name for a given algorithm. For example, the main **dmover** algorithms used by the HBA application are *pci-copyin* (copy from PCI to SDRAM) and *pci-copyout* (copy from SDRAM to PCI).

Since the address spaces used by the CPU running the HBA application and the PCI bus are separate, **dmover** must also handle multiple data representations. The **dmover** API currently supports linear buffers (mapped into the kernel virtual address space), UIOs, CPU physical addresses, and 32-bit and 64-bit scatter-gather lists. The following usages are common in the HBA application:

- *pci-copyin* with a source buffer type of *sglist32* and a destination buffer type of *paddr*.
- *pci-copyout* with a source buffer type of *paddr* and a destination buffer type of *sglist32*.

4.3. HBA iSCSI engine

The iSCSI engine is largely untouched from the user space application, but we did rework a couple of routines to adapt them to the kernel environment.

A number of the functions used in the iSCSI application were already wrapped to allow the kernel and user space implementations to be different. For example, all uses of `malloc()` and `free()` in the iSCSI code were already using `iscsi_malloc()` and `iscsi_free()`. Calls to these wrapper functions were replaced with calls to the corresponding WEPE API functions.

The most obvious place the iSCSI engine required retooling, however, was in how it handled network I/O. The iSCSI engine code uses `struct uio` and the `readv()`/`writev()` family of functions to read and write vectors of data. While this works well in user space, and was possible to do in the kernel environment using WEPE, it is much more natural and efficient in the NetBSD kernel to use *mbuf* chains in those places. This was achieved by adding some data type abstraction in the iSCSI application code.

Since our iSCSI software had heretofore been targeted at stand-alone appliances, it included a full SCSI command processing engine and assumed direct access to the backing-store. Since SCSI command processing is performed by the host in the HBA case,

our iSCSI engine was changed so that a distinct boundary exists between the transport and command processing components, in order to allow either piece to be replaced. This has benefits beyond enabling the HBA application: it enables our appliance software to support other SCSI transports, such as Fibre Channel.

4.4. HBA glue

In addition to code that parses messages from the host system, the HBA application requires a certain amount of "glue" to interface to the iSCSI application. This glue layer forms the SCSI command processing half of the iSCSI engine.

The interface is designed so that the HBA handles as much iSCSI-specific processing as possible. The interface between the host and the HBA is therefore largely a SCSI interface with a couple of hooks to notify the host of new iSCSI connections and sessions.

After each new session is established or a session disconnects, the HBA notifies the host with a unique session identifier. Apart from those messages, the bulk of the interface is simply data movement.

When a new command is received, the HBA sends a message to the host with the SCSI CDB. The host responds with one of several commands:

- *SendDataIn*, indicating that the target is responding to a SCSI "data-in" command or phase. In this case, the message contains a scatter-gather list pointing to the data in host memory that should be sent to the iSCSI initiator.
- *ReceiveDataOut*, indicating that the target is ready for the data in a SCSI "data-out" command or phase. In this case, the message contains a scatter-gather list pointing to the space in host memory where the data from the iSCSI initiator should be placed.
- *ScsiCommandComplete*, which, when sent in reply to a new CDB, usually means an error condition was encountered. This message contains the status and response codes as well as any sense data that should be sent to the iSCSI initiator.

The principal job of the HBA glue code is to marshal these requests from the host with the data requested and provided on the iSCSI session. It also handles moving the data to and from the host, using the scatter-gather lists provided by the host and the **dmover** facility.

5. Results

We had several goals in embarking on this project. Our primary goal was to build a workable system for tackling these "deeply embedded" types of applications with NetBSD running on an XScale™-based platform. Secondary goals included efficient use of the hardware, maximum performance, and adequate tools to measure different aspects of system performance.

While it is difficult to fully quantify, we believe that we have met our primary goal. We have functional iSCSI target HBA prototypes built around our modified NetBSD and a public proof-of-concept demonstration was given at Storage Networking World in April 2003 using an IQ80321-based iSCSI target HBA prototype, in cooperation with Intel and DataCore Software. Unfortunately, we have been less successful in meeting the secondary goals.

After we established the basic functionality of the prototype system, we set out to look at performance. For testing the performance of the iSCSI application, we required two systems: the initiator and the target. The target system included a prototype HBA, driven by a pre-release version of DataCore's SANsymphony™ software with support for iSCSI. This software was running on a dual 2.4GHz Xeon system with 512MB of RAM running Microsoft Windows Advanced Server 2000. The iSCSI initiator system was a Dell Dimension 4500 running with the Intel Pro/1000 IP Storage adapter under Windows XP Professional. On the initiator system, we ran an I/O test application called *Iometer* [8]. The default Iometer workload settings were used.

With the settings we used, iSCSI traffic to and from the target peaked at approximately 7.5MB/s using both 100Mb/s and Gigabit Ethernet networks. As the speed was effectively constant at different wire speeds, we had to conclude that the limiting factor was not in the network path but either in the test structure or in the HBA application itself.

The entire HBA application is relatively complex. To get a clearer picture of the system behavior, we began looking at just the raw TCP/IP performance. For this test, we used the *i*/HBA mounted in the PCI slot on an ADI Engineering BRH board and the Xeon system above running NetBSD with a uniprocessor kernel. The test application used here was *kttcp*, a TCP performance testing tool similar to *ttcp*, except the kernel provides the data source and sink, thus enabling us to test performance in an environment similar to the HBA. On the Xeon system, the *kttcp* version in NetBSD's pkgsrc system in ***pkgsrc/benchmarks/kttcp*** was used; on the *i*/HBA, the same was ported to the **kshell** environment. While the *i*/HBA's low-level diagnostics are able to send raw Ethernet frames at near-line-rate speeds in external loop-back tests, we were only able to achieve a bit above one-third of that speed in basic intersystem testing (as noted in section 2.4). Obviously, we would expect some overhead for TCP/IP processing, but we

did not expect to see such a substantial difference in performance.

After profiling the *i*/HBA side of the *kttcp* test both with time-based profiling and with profiling based on the hardware-resident performance monitoring counters, there is no one clear cause for the poor performance.

Clearly, for this system to progress from "workable" to "commercially viable" on this class of hardware, the performance issues will have to be addressed. However, considering the short amount of time we had in which to produce a working demo, we feel that the effort was a success.

6. Conclusions

In this paper we have described an embedded application outside the realm of traditional BSD embedded applications. We have shown the requirements and challenges of the HBA environment, and how we adapted the NetBSD kernel to deal with them. We have also provided results that show that NetBSD is a viable platform for these types of applications, but that more work is needed in order for it to reach its full potential.

We also believe that WEPE has the potential to enable NetBSD to be used in several applications that were previously off-limits, including system firmware and on MMU-less platforms.

7. Areas for future study

The semantics of Unix system calls make it difficult to write extremely high-performance applications. In particular, the implied ownership of a data buffer before, during, and after a system call is problematic. Furthermore, in the presence of a flat address space, there is no protection boundary to cross during a system call, complicating the use of existing zero-copy data movement schemes. Asynchronous APIs with explicit data ownership should be employed to address these issues.

The memory footprint is still somewhat larger than we would like for the HBA application. Some work should be done to identify dead code and eliminate it. Similarly, we should be able to improve the cache footprint of the software significantly in the kernel environment.

The extant profile analysis tools for Unix assume that time units are counted by the profiling subsystem [9]. This can result in confusing profile reports when using other types of profiling triggers, such as branch prediction misses, data cache misses, or TLB misses. Knowledge of other types of profile events should be added to these tools so that more meaningful reports can be generated.

Some performance gains could be achieved by offloading more processing onto hardware. For example, the Intel i82544 Gigabit Ethernet MAC can perform "TCP segmentation offload", which allows the host to provide a large data buffer, along with a template TCP header, to the MAC, which then performs the task of breaking up the large buffer into appropriately sized TCP segments. The NetBSD TCP/IP stack should be adapted to take advantage of such features.

The basic threading model in the kernel is different from that provided by **libpthread** in user space. Assumptions in one model or the other may adversely affect performance in the other space. It should be possible to either mitigate the adverse effects or develop tools to better measure the effects of scheduling on performance of an application.

8. References

[1] J. Satran et al. "iSCSI", Internet Draft draft-ietf-ips-iscsi-20.txt, pp. 107-108.

[2] J. Satran et al. "iSCSI", Internet Draft draft-ietf-ips-iscsi-20.txt, pp. 198-200.

[3] J. Hufferd. "iSCSI: The Universal Storage Connection", pp. 31-33. Addison-Wesley Publishing Company (2003)

[4] J. Hufferd. "iSCSI: The Universal Storage Connection", p. 25. Addison-Wesley Publishing Company (2003)

[5] "BusLogic Multi-Master Ultra SCSI Host Adapters for PCI Systems Technical Reference Manual", BusLogic, Inc. (1996)

[6] "Intelligent I/O (I2O) Architecture Specification, Version 2.0", I2O Special Interest Group (1999)

[7] RedBoot: *http://sources.redhat.com/redboot/*

[8] Iometer: *http://sourceforge.net/projects/iometer/*

[9] S. Graham, P. Kessler, M. McKusick. "gprof: A Call Graph Execution Profiler", proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6., pp. 120-126 (1982)

9. About the authors

Jason R. Thorpe is the Chief Science Officer of Wasabi Systems, Inc. A contributor to the NetBSD

Project since 1994, he has been involved in developing high-performance storage and networking systems using NetBSD for over eight years. Mr. Thorpe is a past participant in the IETF's TCP implementation working group, and also contributes to the GNU GCC, Binutils, and GDB projects.

Allen K. Briggs started working with BSD shortly after the release of 386BSD 0.0 by contributing to a BSD port to Apple's m68k-based Macintosh systems, which evolved into NetBSD/mac68k. Mr. Briggs has continued to work with NetBSD in his spare time, and in late 2000, joined Wasabi Systems, Inc., where he has been involved in a number of NetBSD-based embedded systems projects.