Abstract—This paper describes Groundhog, an open-source SATA host bus adapter (HBA) for FPGAs. This system makes it easy for FPGA-based applications to directly interact with permanent storage devices. This allows reconfigurable computing devices to be used in new applications that require bulk storage and presents additional opportunities to increase performance, reduce power consumption and improve system integration. In addition to standard disk sector read/write commands, this framework also supports more advanced concepts such as native command queuing (NCQ) introduced with SATA II. We test the system with latest-generation SSDs and demonstrate the potential performance advantages and trade-offs of direct hardware access to bulk storage devices.

Keywords—FPGA, SATA, I/O, Xilinx, Virtex-5, open-source

I. INTRODUCTION

Several benefits arise from directly connecting FPGAs to permanent storage devices (i.e. hard disks and solid-state drives (SSDs)) creating an interesting opportunity for new system design approaches.

1) Performance. Having hardware located close to the storage device allows for optimizations to be applied in parallel and early in the data path. For instance, the IBM Netezza 1000 [5] (formerly Twinfin) is a data warehouse appliance, where data filtering and data compression are performed within the storage nodes (containing FPGAs). This reduces data movement within the appliance.

2) Power consumption. Storage servers built using general-purpose processors require significant power to operate and properly cool. FPGAs can be used to build more efficient low-frequency, high-throughput interface and management logic for storage devices. Key portions of Groundhog were used in [4].

3) Integration. To build a self-contained storage appliance, the FPGAs must be able to directly access bulk storage devices.

Serial ATA (SATA) is a standard interface for accessing hard disks and SSDs. This connection requires a device known as a host bus adapter (HBA). Many Xilinx FPGAs1 are typically equipped with special I/O transceivers (GTP/GTX) that allow communication over high-speed serial links. However, these transceivers only provide the physical layer of SATA. To the best of our knowledge, in this paper we present the first open-source and freely available [3] SATA core for Xilinx FPGAs.

II. SATA PROTOCOL OVERVIEW

The SATA interface [1] is defined as a protocol consisting of the five layers depicted in Figure 1.

The application layer of the HBA provides an interface to communicate with host software (this is typically a driver in a conventional PC). In our implementation, we omit this layer and assume that a user circuit on the FPGA can directly interface to the command layer.

The command layer is responsible for execution of SATA commands. Executing a command involves the host and the device exchanging a number of packets—so-called Frame Information Structures (FISs). Finite state machines in the command layer define the series of FISs that correspond to each command.

The transport layer creates FISs requested by the command layer. These FISs are buffered and most of them can be resubmitted in case of a transmission error (see Section III-D).

The link layer is responsible for individual FIS-based transactions. 8b/10b encoding also technically belongs to the link layer, but as this is already implemented by the GTP transceivers on the FPGA, we consider it part of the physical layer, together with parallel-serial conversion.

Figure 1. SATA protocol layer overview: our HBA implementation comprises the command layer, the transport layer, and the link layer.

1Such as the Xilinx Virtex-5 LX110T FPGA used here.
III. HOST BUS ADAPTER (HBA) DESIGN

In this section, we discuss the most important components of our design. Later, in Section IV, we will extend this design to support native command queuing (NCQ).

A. Design Goals

The main goal of the codebase presented here is to provide a fast, lightweight, and easy to understand framework for FPGA developers to build upon. Therefore, rather than trying to support all SATA commands (e.g. legacy programmed IO (PIO)), we focus on those needed for quickly reading and writing disk sectors. Nevertheless, the code is extensible to support more commands, as we will show in Section IV.

B. Design Overview

Figure 2 depicts a modular view of the HBA. The top-level HBA Module provides the interface to the user application, consisting of control signals (e.g. issuing commands) and data buses for reading/writing data (e.g. disk sectors).

At the other end, the HBA Module is connect to the I/O pins leading to the drive. A high-speed serial transceiver (GTP tile) is instantiated and configured for SATA operation. We support SATA I (1.5 GHz) and SATA II (3 GHz) devices. At the time of writing, we did not have access to an FPGA (6 GHz) is not yet supported.

C. Link Module

The Link Module is responsible for FIS-based transactions (i.e. transmitting and receiving individual FISs). After the link has been initialized (please refer to [2] for details on the initialization procedure), the link layer protocol state machine is in the idle state. However, this does not mean that the link is electrically idle. On the contrary, at the link layer, communication constantly flows in both directions between host and device. The communication stream consists of so-called primitives\(^2\) (special 4-byte tokens) and FISs that are embedded in this stream of primitives. An idle condition results in both host and device repeatedly and simultaneously sending SYNC (0xB5B5957C) primitives. Figure 3 illustrates a FIS transmission from host to device.

\(^2\)We denote primitives with capital, italic letters. Example: SYNC.

\[\text{Figure 2. HBA module - architectural overview}\]

Once the host is ready to transmit a FIS, it starts sending X_RDY (1) primitives (indicating that it has a FIS to send) until the device replies with R_RDY (2) primitives. Then the host uses an SOF and an EOF primitive to frame the FIS (3). During reception of the FIS the device returns R_IP (4) primitives, which means that FIS reception is in progress. Finally, the host asks the device to terminate the transaction (sending WTRM (5) primitives), upon which the device completes the transaction with R_OK (6) (or R_ERR in the case of a low-level error, e.g. a CRC error).

Besides implementing the link layer protocol for FIS-based transactions, the Link Module takes care of a number of other tasks such as CRC-32 generation/verification, FIS scrambling (to reduce electromagnetic interference (EMI) on the link), and flow control (transmission of a FIS can be paused by the device or the host using a HOLD primitive).

D. Command/Transport Module

Our Command/Transport Module implements the five commands given in Table I. The Identify Device command will return a 512-byte data structure indicating supported and enabled features. Read DMA and Write DMA are commands to read/write a given number of 512-byte sectors. The extended version of these commands allow larger reads/writes (i.e. chunks of up to 32 MiB as opposed to 128 KiB). Finally, First Party DMA Read/Write commands are used for native command queuing (NCQ) (discussed in Section IV).

Executing a SATA command involves the exchange of multiple FISs between host and device, which are individually transmitted via the Link Module. For example, execution of the Read DMA Extended command is illustrated in Figure 4. First, the read command is sent from the host to the device by means of a Register FIS, which also includes

\[\text{Table I}\]

\begin{tabular}{|l|l|}
\hline
Command & Code \\
\hline
Identify Device & 0xEC \\
Read DMA Extended & 0x25 \\
Write DMA Extended & 0x35 \\
First Party DMA Read (NCQ) & 0x60 \\
First Party DMA Write (NCQ) & 0x61 \\
\hline
\end{tabular}
the logical block address (the starting disk sector address) and the number of consecutive disk sectors to read. When the device has the requested data ready, it sends the data back to the host using one or more Data FISs. In the end, the device confirms the completion of the command by sending a Register FIS with status information to the host.

In addition to managing the FISs for the various commands, the Command/Transport Module checks for errors reported either by the device (via Register FIS) or by the Link Module (e.g. CRC errors). For certain types of errors the Command/Transport Module independently retransmits FISs. Two small buffers are used for incoming and outgoing traffic respectively. Each buffer is large enough to hold any type of FIS with the exception of Data FISs, which are only partially buffered and can thus not be retransmitted (as intended by the SATA specification [1]). An error regarding the transmission of a Data FIS cannot be solved by the HBA and is reported to the application.

E. HBA Interface

The interface exposed to the user is illustrated in Figure 5. A user application interacts with the HBA module by first issuing a command by setting cmd appropriately and asserting cmd_en. Depending on the command, lba (the starting disk sector address) and sectorcnt (the number of disk sectors affected) are also specified.

If the application issues a write command, it can start filling the write buffer using the write data port wdata and the corresponding handshaking signals (wdata_en and wdata_full). Assertion of cmd_failed indicates that a fatal error occurred and assertion of cmd_success (after all of the data has been sent) indicates write completion. A read command proceeds in a similar manner, with the user application monitoring rdata subject to the handshaking signals rdata_empty and rdata_next.

IV. Native Command Queuing

Native command queuing (NCQ) was introduced with SATA II and was originally designed to reduce seek time. A hard disk that supports NCQ can queue up to 32 commands for out-of-order execution. Although SSDs no longer have physical moving heads, they also benefit from NCQ. This is because SSDs typically consist of multiple flash chips that can be read-written concurrently. Furthermore, despite the high theoretical throughput of modern SSDs, each separate IOP can still exhibit considerable latency. With NCQ, the SSD can distribute small reads/writes in a more optimal way over these channels and process the commands asynchronously to hide latencies and maximize throughput.

A. Queuing Mechanism

Queuing is implemented by assigning a distinct 5-bit tag \( \{0,...,31\} \) to every command in flight. For this purpose, two new commands were introduced in SATA II: First Party DMA Read/Write. Figure 6 illustrates the execution of a First Party DMA Read command.

When a NCQ read is issued, the drive quickly acknowledges the command with a Register FIS. At this point the HBA can issue more commands while the drive is still processing the queued read. Later, when the drive has at least part of the read data ready, it communicates back to the HBA using a DMA Setup FIS (referencing the corresponding tag previously provided), followed by a DATA FIS (containing the actual data). Multiple DMA Setup FIS/DATA FIS pairs will be sent if more than 8 KiB of data were requested. Finally, the command is completed with the device sending a SetDevBits FIS to the host.

B. Changes to the Host Bus Adapter

To support NCQ we had to change our HBA implementation in two ways: (1) extend the HBA interface (2) extend the Command/Transport Module. Notice that no changes to the Link Module were required.

The user interface needed to be extended to expose queuing-specific information to the application (e.g. the application needs to be able to supply a tag together with a request to execute a command). In addition, the HBA now needs to inform the application explicitly of completed commands since partial completion is also possible.
We measured performance with two SSDs: the Intel SSD (W3520) quad-core processor and 6 GB of main memory. a 64-bit Windows 7 machine with a 2.67 GHz Intel Xeon might be used, an FPGA can provide better performance. This is particularly true for applications where small random accesses are unavoidable and latency is of paramount concern.

Beyond performance, one of our main design goals was to keep the SATA HBA lightweight. Thus, as seen in Table III, our implementation uses no BRAM and very little logic (3.8% of the available slices on our FPGA).

VI. CONCLUSION

We have presented the first open-source SATA host bus adapter (HBA) for FPGAs. We have implemented the necessary SATA commands to offer FPGA applications easy and fast access to persistent storage. Beyond the classical read/write commands, we extended our HBA to support native command queuing (NCQ). This significantly improves performance for small random accesses. Our code has excellent performance yet very modest resource requirements, using only 3.8% of the available slices on a Virtex-5 LX110T. This makes it ideal for performance-sensitive embedded applications. The code can be downloaded at [3].

REFERENCES


The Command/Transport Module was extended by modifying the code for assembling the Register FIS (e.g. embedding tag information) and adding finite state machine support for the First Party DMA Read/Write command protocols.

V. EXPERIMENTAL RESULTS

To demonstrate the potential performance advantages of direct FPGA access to permanent storage—beyond the issues of power consumption and system integration—we compared the throughput of our FPGA implementation to that achieved on a conventional PC. The throughput on the CPU was computed using CrystalDiskMark 3.012 on the Intel SSD 320 Series (300 GB) and the OCZ Vertex 3 (240 GB).

Table II shows that conventional software can introduce measurable performance overhead. Even when reading consecutive 1024 KB chunks (Read Seq DMA) the FPGA is roughly 1.07x faster for both the Intel and OCZ drives. However, as the accesses become smaller and random, the overhead of the software drivers becomes more apparent.

For example, in the case of non-NCQ accesses of random 4 KB chunks (Read Rnd DMA), the FPGA is 1.47x and 4.26x faster on the Intel and OCZ drives, respectively. The write performance follows a similar, but slightly less dramatic trend.

NCQ does help address some of the software overhead issues. For example, overlapping 32 NCQ IOPs (Read Rnd NCQ) closes the gap between the CPU and FPGA implementations for the Intel drive, but the overhead is still noticeable on the faster OCZ SSD (the FPGA is 1.16x faster than the CPU).

That said, looking at Figure 7, NCQ also significantly increases average latency. For example, with 32 read commands in-flight, latency increased 5.26x for the Intel drive and 5.06x for the OCZ drive.

Putting these factors together, considering embedded applications where a processor less powerful than a Nehalem Xeon might be used, an FPGA can provide better performance. This is particularly true for applications where small random accesses are unavoidable and latency is of paramount concern.

Table II

<table>
<thead>
<tr>
<th>Access Type/Size</th>
<th>Intel CPU</th>
<th>Intel FPGA</th>
<th>OCZ CPU</th>
<th>OCZ FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Seq DMA/1024 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Rnd DMA/4 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Rnd NCQ/4 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Seq DMA/1024 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Rnd DMA/4 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Rnd NCQ/4 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table III

<table>
<thead>
<tr>
<th></th>
<th>Slices</th>
<th>Flip-Flops</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>available</td>
<td>17,280</td>
<td>100.0%</td>
<td>69,120</td>
</tr>
<tr>
<td>occupied</td>
<td>652</td>
<td>3.8%</td>
<td>763</td>
</tr>
</tbody>
</table>

Figure 7. NCQ (4 KiB random reads): throughput and average latency

Table IV

<table>
<thead>
<tr>
<th>Access Type/Size</th>
<th>Intel CPU</th>
<th>Intel FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Seq DMA/1024 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Rnd DMA/4 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Rnd NCQ/4 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Seq DMA/1024 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Rnd DMA/4 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Rnd NCQ/4 KB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>