

Whitepaper: Optimizing Software Architecture in PCIe Accelerator Card System Configurations

Matthew Dharm
Lead System Architect, JumpGen Systems LLC

7 December 2010

Overview

Even as the number of cores in general-purpose CPUs continues to climb, many difficulties remain in building systems that are capable of handling large volumes of medium- and high-touch packet operations. This is due to several factors, including the ever-increasing rate of network traffic and some fundamental limitations in general-purpose computing architectures. System designers are constantly seeking ways to expand their system capacity beyond such limitations.

One such architecture that is commonly used to expand system capabilities beyond those of a general-purpose CPU is the addition of one or more “accelerator” cards. Such cards usually contain multiple network interfaces, some number of general-purposes CPU cores, and some number of hard-accelerator cores for specific tasks such as pattern matching, cryptography, compression, or even storage-related mathematical operations. Often, much of this capability is contained within a system-on-chip (SoC) Network Processor (NPU) or other similar part. Connectivity between the general-purpose CPU and the accelerator card is generally done via a multi-lane PCIe connection between the two subsystems.

While system design such as this offers significant increases in system performance, the task of the software designer now includes the added responsibility of making certain that all sub-system components are being utilized to their maximum capability. Also, software must maximize throughput and minimize latency in the data path between the general-purpose CPU and accelerator card.

In this paper, we will consider a hypothetical 1U rack-mount server with multiple PCIe accelerator cards. Each accelerator card will have multiple 10GigE and 1GigE interfaces and will be connected to the server CPU via a PCIe interface.

Note that the total amount of bandwidth required between the general-purpose CPU and the accelerator card varies greatly by the specific application. However, the techniques we will discuss here are applicable to all of these applications. Regardless of total bandwidth required, making maximum use of available system resources allows for higher performance at lower cost.

Efficient Data Transport

Transporting data between an accelerator and host processor is, perhaps, the most obvious area of focus for developers interested in optimizing system performance. System designers with experience in this area might find this section redundant, but the prevalence of systems with poor design in this area suggests that spending some time reviewing well-known best-practices for moving data across a PCIe bus would be time well-spent.

Classically, there are three particular areas which are of critical importance to moving data across a PCIe bus: the use of Direct Memory Access (DMA) transfers rather than Programmed I/O (PIO), the use of writes instead of reads, and the efficient use of interrupt requests (IRQs).

DMA vs. PIO

Many designers view the use of DMA as an opportunity to reduce the number of processing cycles devoted to moving data from one location to another. While that is one benefit, when dealing with multiple processing complexes (such as in a system with a host processor and an accelerator card), the real benefit is in maximizing the usage of the interconnecting PCIe bus.

For those not familiar with the technology, DMA involves the use of a dedicated “engine” specifically designed to copy data from one location to another. This engine is programmed by a processor, and they often have a variety of transfer modes suitable for packets, multi-dimensional arrays, storage blocks, or other common data types. Some even support descriptor-based operation, allowing multiple jobs to be queued-up and processed automatically.

What is, perhaps, less obvious about DMA engines, however, is that their base-unit of operation is usually quite large. For example, a processor may be able to operate on 64-bit values (8 bytes) in a single load/store instruction, while a DMA engine may easily be able to operate on 256-bit (32-bytes) at a time. DMA engines commonly operate on even larger data units; the base size often matches up with the maximum transfer size of a cache line, main memory burst, or other on-chip interface.

The difference between DMA and PIO, then, is transaction overhead on the PCIe bus. When using PIO to transfer data, each 8-byte chunk of data gets hit with the overhead of initiating and ending a transaction. When using DMA, that overhead penalty is at least 4 times less, possibly even 8 or 16 times less, than when using PIO.

PCIe Writes, not Reads

When transferring data between two entities, many designers will arbitrarily choose to ‘read’ the data from one side rather than ‘write’ the data from the other. To many designers, the choice is arbitrary and one of convenience rather than one of performance. However, this choice has serious performance implications.

The issue at hand is fundamental to the operation of PCI-like busses, including PCIe. When writing data on such a bus, all intermediate bridge devices between the initiator and target are allowed to buffer the data being written. Since the bridges are buffering data, multiple writes can be happening through any given bridge simultaneously. Several rules apply to this

buffering, including mandatory flushing of buffers under certain conditions to give the appearance of coherency.

However, reads performed on such busses are entirely different. In the case of a read, all intermediate bridge devices between an initiator and target must interlock simultaneously, providing a clear and continuous path for the two entities to exchange data until one decides to stop (or the expiration of the latency timer). The overhead to setup the read interlock is significant, and no other reads through the intermediate bridges can happen during this time.

It is worth mentioning that many system bugs are caused by developers not properly understanding the rules for buffering PCI writes, or not properly applying these rules properly to their system design. This often leads developers to switch to the lower-performance “read” configurations. Familiarity with these rules, known as PCI Write Posting Rules, is recommended for anyone implementing a PCIe-based system.

IRQ Management

While most people are familiar with interrupts signaled from an accelerator card to a host CPU, it is also common for a host CPU to signal an interrupt to an accelerator card. These interrupts can be used to indicate any one of a number of semantic meanings, but most commonly they signal that data is either ready for processing, or processing on a block of data has completed. Interrupts can be implemented any number of ways, from physical interrupt request lines, to PCIe in-band interrupt messages, to message signaled interrupts, to mailbox interrupts.

Receiving an interrupt, however, is generally the same regardless of how or where it is generated. Also, receiving an interrupt is a costly process in terms of computational resources. Receiving and processing an interrupt generally requires a processor to switch contexts, handle the interrupt (at least minimally, often deferring most of the work until a later time), and then restore context to what it was before the interrupt happened. The overhead of these operations is high, and as such high rates of interrupts can cripple a system. This phenomenon is the reason many systems have difficulty processing a low-bandwidth data stream that is composed of extremely short packets; each packet causes an interrupt, and the overhead of the interrupts overwhelms the processing capacity of the system, even at a relatively low bandwidth.

There are many ways to mitigate this issue. One such method is “interrupt coalescing”, where multiple interrupts that happen close together in time are issued as a single interrupt (this is usually done with some sort of hold-off or delay logic). Another technique to reduce interrupt frequency is to reduce how interrupts are used, changing from (for example) an interrupt for every packet to an interrupt for every 1000 packets or for every flow setup/teardown. Regardless of the method used to manage interrupts, managing the amount of processing power consumed by interrupts can be an important part of system performance.

Dividing Up the Workload

While it is important for system designers to be mindful of how data is moved between a host processor and an accelerator card, it is also important to consider where and how the data is manipulated. Specifically, it may be a performance gain to pre- or post-process data on an accelerator card, even if that accelerator card has a different primary function. Also, when

transferring data between a host processor and accelerator card, choosing a sensible format for the processor architectures involved can significantly improve performance.

Pre- and Post- Processing Data

Many systems that use an accelerator-based architecture initially target the accelerator for very specific parts of the processing workload. For example, many accelerators have dedicated hardware to support compression or certain types of cryptography. These dedicated acceleration blocks are faster than either the host CPU or the processor on the accelerator card. While this type of design does lead to improved system performance, often the processor on the accelerator is extremely under-utilized, leaving a large window of opportunity to improve system performance.

When allocating what workload will be done on the host or on the accelerator, the total capabilities of the accelerator, including any general-purpose processing capabilities, should be considered. Moving a task as simple as a packet checksum calculation to an accelerator that is already handling that packet can free up valuable host processor resources and provide a significant performance boost.

Formatting Data between Subsystems

While moving data between subsystems efficiently is important, moving that data in the best possible format is equally important. Different CPUs have different strong and weak aspects of their architecture, and using the accelerator to change the format of the data to be more suitable for host CPU processing can yield performance improvements.

For example, many CPUs are more efficient when data is aligned on a 64-bit boundary, but Ethernet packets have many fields which are not aligned on any reasonable boundaries. Misaligned memory accesses can be a significant source of performance problems, and are almost invisible to most debugging tools, such as profiling. However, if an accelerator card segmented the packet into multiple buffers such that fields of interest were aligned for the convenience of the host CPU, then the host CPU could process the data more efficiently at the relatively minor cost of a few bytes of memory. Alternatively, an accelerator card could extract the desired information from the packet, and then transfer an additional data block to the host CPU which provides a summary of that information in an easy to access format.

Conclusion

Accelerator cards can offer significant performance gains in a variety of tasks on many platforms. The availability of such accelerator cards in multiple standards-based form-factors makes them an attractive option to boost system performance, especially in systems where the increase in performance is viewed as a valued-added addition or option to the system. There are many accelerators to choose from on the market today. However, accelerators are not a “silver bullet” that can be implemented without regard for the overall system design or the interface to the host CPU. Careful design and implementation are necessary to insure optimal system performance.