

# Software and Performance Ramifications of PCI and PCIe Design Paradigms

Brad Parker  
*Heeltoe Consulting*  
*brad@heeltoe.com*  
October 2010

(rough draft; unreviewed)

## Abstract

This is a brief review of the effect on software of a range PCI and PCIe design choices. An attempt is made to survey the available PCI design choices and to enumerate the various software mechanisms which can be used to create an interface to these hardware designs.

No attempt is made to delve into the details of PCI/PCIe configuration nor to enumerate all the effects of PCI/PCIe topology. Where possible PCI and PCIe are treated as similar busses. When needed the differences are broken out and described.

The goal is to produce an overview which outlines which design choices are possible and to describe the affect these choices have on both software and system performance.

## 1. PCI Interface Choices

For the purposes of this paper it is assumed that there is a single CPU which is generally acting as a PCI master. The CPU handles all PCI bus setup and enumeration tasks. There is a single PCI device on the PCI bus which contains some functionality. Several different forms of device function will be discussed. Initially we assume that a small request is somehow transmitted to the PCI

device and after some time a small response is produced by the PCI device. Later more complex interactions will be explored.

The PCI device in its generic form presents two interaction spaces to the CPU. One space has been deemed suitable for I/O (input/output) access and the other for memory accesses. This differentiation is largely for historical reasons based on the X86 CPU architecture and instruction set.

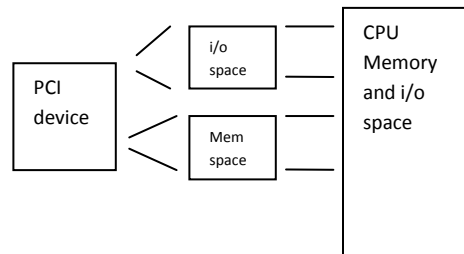


Figure 1, PCI memory and I/O spaces

The basic idea is that the PCI device, through its configuration registers, exposes an I/O space and a memory space. These spaces are subsequently mapped into the I/O space and memory space of the CPU. There are three types of PCI bus transactions – I/O, memory and configuration. We will focus on I/O and memory transactions.

## 2. Basic I/O Port Access to PCI Device

The simplest implementation of a PCI device provides one or more I/O ports in the CPU I/O port space. Access is somewhat X86 instruction set specific - an “in” instruction generates a PCI bus transaction to the PCI device to retrieve 32 bits of data. Conversely an “out” instruction generates a PCI bus transaction to the PCI device to send 32 bits of data. These transactions are simple and effective but they are not fast nor are they efficient in terms of CPU or PCI bus utilization.

PCI in/out bus timing, 33MHz PCI bus:

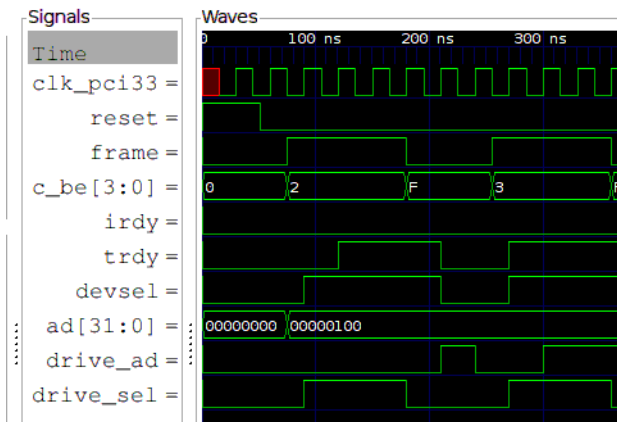


Figure 2, PCI Waveforms, simple I/O

NOTE: I/O writes may be “posted” or buffered in a PCI bridge, where I/O reads can not.

Measured PCI I/O performance

300ns between I/O writes (in)  
800ns between I/O reads (out)

Unrolled loops of “in” and “out” instructions can be used to minimize loop overhead but are limited by the basic speed of PCI bus I/O transactions. The X86 “string I/O” instructions can be used to create multiple back to back I/O transactions but do not

generate burst transaction and are therefore not any faster than unrolled loops.

It is most often the case that any PCI bridge will “delay” all I/O transactions and wait until the transaction is complete before acknowledging the transfer.

## 3. Memory Mapped Access to PCI Device

The PCI device can also present a memory space to the CPU. In this case the CPU can map the PCI device memory into its own CPU address space and perform any type of memory operation (i.e. read or write) on the mapped memory space. In this mode the CPU can create special data structures within the PCI device’s memory space.

Register I/O (i.e. I/O space access) is simple and does not have complex transfer issues. Memory I/O is not as simple. For example, the CPU can cache the memory space and can use burst accesses to read and write the memory. PCI bridges can buffer both reads and writes.

Potential issues with CPU-to-PCI device memory access:

- Memory coherency; invalid CPU cache entry versus valid PCI device memory contents
- Bus snooping/cache invalidation; second PCI master writing to PCI device memory
- Read/Write buffering by one or more intermediate PCI bridge(s)
- Read/Write burst access to PCI device memory
- PCI bus bandwidth

### 3.1 Memory Mapped PCI Access Modes

There are several different techniques available for getting data into and out of the PCI device when using a memory mapped mode. The PCI device can present “raw RAM” memory to the CPU which is filled by software and later read and interpreted by the PCI device. Alternately, the PCI device can present non-memory as memory, as in the in case of a FIFO which spans a large address space. The software can then use loop unrolling techniques to write to sequential PCI device memory locations all of which map to the same FIFO input.

In both cases the goal of the software is to maximize the data through-put between the CPU and the PCI device.

If a PCI bridge is inserted between the CPU and the PCI device, memory write transactions can be “posted” or buffered. Conversely memory read transactions are typically “delayed” by an intermediate bridge. Greater throughput can be achieved if all transactions are memory writes. Bridges can also buffer and coalesce bus writes which fall on cache line boundaries into a single cache line transaction.

*[aside: I once helped develop a system which used only PCI writes. Data was “pushed” from one PCI master to a PCI device with posted writes. The PCI device always responded by in-turn becoming a PCI master and “pushing” back responses as writes to the originating PCI master. The system achieved very close to the theoretical maximum 33MHz PCI bus throughput because it never used PCI read transactions.]*

It’s also important to note that PCI bus arbitration time is non-zero and therefore significant. CPU writes can be done as a “burst”, amortizing one arbitration cycle again multiple writes. If each read requires

a new arbitration, reads incur a greater time penalty.

#### 4. PCI Device as Bus Master

The PCI device itself can also become a bus master to read and/or write CPU memory. In this mode the CPU can set up data structures which the PCI device will interrogate and potentially modify.

Potential issues with PCI device-to-CPU memory accesses:

- Bus snooping/cache invalidation; PCI device write writing to CPU memory with stale CPU cache entry
- Read-delay/Write-coalescing by intermediate PCI bridge(s)
- Read/Write burst access to CPU device memory
- PCI bus bandwidth
- CPU bus utilization

Typically the PCI memory will be marked and mapped in such a way that no caching occurs the CPU side. This eliminates cache and cache snooping issues. It also eliminates the possibility of write bursts caused by cache flush operations.

##### 4.1 Memory Polling Model

When the PCI device acts as a bus master it is possible to take command and control information from the CPU memory. This is often done using data structures with regular shape and size. One downside is that this technique can force the PCI device to “poll” the CPU data structures waiting for input or additional data, eating up precious bus and memory bandwidth.

*[aside: take a look at OHCI and EHCI USB controllers. Both are intensive bus hogs largely due to this polling behavior.]*

## 4.2 Push Only Model

As an alternative, the PCI can operate in “push only” mode. This model has both the CPU and the PCI device only using write transactions. The CPU writes requests to the PCI device and the PCI device writes responses back to the CPU. There are no inefficient read transactions and since only writes are done there is the greatest possibility that writes will be coalesced into burst transactions and take advantage of posted writes. In addition, PCIe semantics (namely “relaxed ordering”) can be used to enhance performance through bridges and switches.

## 5. Sequence of Events

We examine the sequence of events in each model and the effect on CPU and PCI device utilization. The goal for maximum performance is 100% simultaneous utilization of the CPU and PCI device.

### I/O Access Timeline

- OUT instruction(s) to send request
- IN instruction(s) to poll for completion
- IN instruction(s) to receive data

### Simple RAM Access Timeline

- Write request to PCI device RAM
- Read (polling) PCI device ram waiting for completion
- Read PCI device RAM to get response

### Bus Master Polling Timeline

- CPU writes request into CPU RAM
- CPU marks CPU RAM request “ready”
- CPU polls CPU RAM for done status
- PCI device polls request queue
- PCI device notices pending transaction
- PCI device reads request transaction
- (processing occurs)
- PCI device writes response data
- PCI device writes response status; interrupt
- CPU notices request has completed

### Bus Master Push-only Timeline

- CPU writes request to CPU RAM
- CPU polls CPU RAM for result
- (processing occurs)
- PCI device writes response to CPU RAM; interrupt
- CPU notices request has completed

*<time line pictures for each model>*

## 6. Design Choices and Performance Limitations

PCI Device implementations with their associated software roughly fall into three categories. These categories are defined by the amount and direction of data being transferred:

- Many large buffers in both directions (Ethernet and disk controller)
- Most data moving in one direction with the possibility of caching/read-ahead (display controller)
- Many small buffers in both directions (specialized data processing)

We will examine each case and talk about performance issues.

### Lots of Data Moving in One Direction

This is the easiest problem to solve. The system is never required to wait for data. Since data only moves in one direction (i.e. writes), bursting is possible. Any amount of write coalescing will speed up the transfers. In a perfect world the CPU will burst write out to the PCI device when flushing a cache line. PCI devices when operating as bus masters will burst and take advantage of PCIe relaxed ordering rules.

### Large Buffers in Both Directions

This is one of the most common PCI scenarios. This solution ties up a lot of bus bandwidth but can be very efficient when larger buffers are used. Unfortunately high packet completion rates can cause CPU event congestion due to excessive interrupts (i.e. too high an interrupt rate for the CPU to service efficiently)

### Small Buffers in Both Directions

This is the most difficult scenario because small buffers are inefficient and high levels of interrupt rates can create CPU problems. The system may be constantly waiting for data and tied up in a “request/response loop”. If possible the goal should be to get as many requests in flight as possible and to decouple the response to one request from the initiation of the next request. In more simple terms, the system should generate a “train” of requests which will be followed by a “train” of responses, similar to classic hardware pipelining.

Some type of interrupt coalescing is generally required. There are several different techniques.

#### MSI

#### Timer Based Interrupts

A good reference from PCISIG (not specific to PCIx):

[http://www.pcisig.com/developers/main/training\\_materials/get\\_document?doc\\_id=00941b570381863f8cc97850d46c0597e919a34b](http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=00941b570381863f8cc97850d46c0597e919a34b)

Relaxed transaction ordering and “no snoop” transactions.

Interesting paper on PCI and CPU cache interactions:

[https://netfiles.uiuc.edu/rpelliz2/www/index\\_files/techreps/coscheduling.pdf](https://netfiles.uiuc.edu/rpelliz2/www/index_files/techreps/coscheduling.pdf)

## 7. The Fastest Way

With both PCI and PCIe the best way to improve performance is to eliminate reads. The next best thing to do is to increase the transfer size. PCIe efficiency drops dramatically with small transfer sizes (less

than 50 bytes). An ideal PCIe based system will have the PCI device act as a memory device and a bus master. The CPU will write transactions to the PCI device and the PCI device will become a bus master and write responses back to the CPU memory. A PCI device can generate burst transactions, a PCIe device can generate transactions with multiple writes.

The PCIe device will mark its transactions with “relaxed ordering” when possible and flush outstanding transactions with a final non-relaxed write. This will maximize the possibility for bursting to occur and minimize transaction blocking which can occur in bridges and PCIe switches.

Little or no polling will be done by the PCIe device and interrupts will be grouped and delivered in such a way as to minimize the interrupt frequency as seen by the CPU.

Using these techniques should provide the highest possible PCIe performance and the most efficient software. Special care must be taken when writing data with PCIe relaxed ordering rules. PCIe transactions marked as relaxed can occur out of order. If the PCI device performs its final status write using a non-relaxed transaction, the PCIe bus transaction ordering rules will ensure the non-relaxed transaction will occur after all of the data transactions have completed.

## PCI Bandwidth

4 byte transfers (32 bit) at 33MHz yields 132Mbytes/sec.

For single transfers, the maximum write transfer rate is 66MBytes/sec and the maximum read transfer rate is 44MBytes/sec.

For burst transfers, the maximum 4 byte transfers (32 bit) at 33MHz yields 132Mbytes/sec.

## PCIe – Differences from PCI

PCIe transfers are packet based. Small packets are sent to and from the device via the PCIe serial bus. These transactions are Transaction Layer Packets (TLP's).

Read – 3 DW, no data + ClpD  
Write – 4 DW, with data

2.5GHz clock

PCIe 1.0 x1 400MBytes/sec throughput, 2.5 GigaTransactions/second

4x 1.6GByte/sec throughput

PCIe switch can break down large transfers into 128 or 256 byte transfers.

PCIe low latency switches; down to 110ns of latency.

PCIe measured efficiency versus payload size

25 bytes	~ 50%
50 bytes	~ 70%
75 bytes	~ 80%
200 bytes	~ 90%

Small transfers are not very efficient.

PCIe measured maximum throughput versus payload size (approximate):

Size (bytes)	x1	x4	x8
25	100	600	1200
50	200	700	1500
75	200	800	1600
200	200	900	1800

A single lane (x1) tops out quickly.

Hardware buffers sizes are also important on x1 links due to flow control (FC) update latency.

References:

See Agilent paper, "PCI Express Performance Measurements"  
<http://cp.literature.agilent.com/litweb/pdf/59-4076EN.pdf>

Good performance comparisons in Altera FPGA environment:

<http://www.altera.com/literature/an/an456.pdf>

## PCI Bridge and Transaction Issues

Target-disconnect

Target-retry

Target-abort

Terminate-with-retry

Write = posted transaction

Write-invalidate-data = posted transaction (use to help snooping)

Cache line transactions – write combining

memory-read-multiple transaction

Posted memory write; writes can be buffered by bridge into cache line size transactions.

Prefetchable-read; bridge will prefetch read sequential addresses if they are in a cacheable region and master asserts frame#. Memory-read-line and memory-read-multiple transactions

Non-prefetchable-read; one and only one read is performed as a single transaction.

Memory read = delay transaction  
I/O read/write = delay transaction

Typically I/O space memory is not marked "prefetchable" (except for display cards).

---

**<references>**