Systems Performance Measurement on PCI Pamette

Laurent Moll
Pôle Universitaire Léonard de Vinci
La Défense, France.
Laurent.Moll@devinci.fr

Mark Shand
Digital Equipment Corporation, Systems Research Center
Palo Alto, California, USA.
shand@acm.org

Abstract

We describe the use of a reconfigurable board to obtain information on the performance that can be expected on particular systems. Our goal is to use the reconfigurability of the board's interface to test a system and discover not only the maximum bandwidth and best latency attainable, but also the way to reliably achieve these figures.

The board we present uses the now widespread PCI bus. PCI is sufficiently complex, and its implementations sufficiently varied, that it is impossible to guess the performance that can be obtained by a specific board on a specific computer with the only technical characteristics of the two in hand. We observe astonishing performance differences between almost identical systems and comparable figures between small PCs and big servers.

Our performance tests can be an end in themselves, however they also serve to demonstrate the value of a reconfigurable bus interface. With the same board, we can test and choose a system, make informed architectural decisions on the hardware/software interface, and then finely tune the bus interface to get maximum and predictable figures in the running application.

1 Introduction

PCI Pamette [1] is a reconfigurable computing device which uses the PCI bus to connect to its host processor. PCI Pamette follows strongly in the tradition of the PAM Project [2]: the reconfigurable logic is seen as a *coprocessor* designed to work in harmony with the host processor. Applications seek to leverage the maximum from the presence of a modern microprocessor replete with powerful graphics, fast disk subsystems and high-speed networking as is found on a modern workstation or high-end PC. As such, the performance of many applications depends critically on the performance of the host link.

We approached PCI with some trepidation. Our previous experience had been with systems based on Digital's TURBOchannel, a simple high performance system expansion bus that, as early as 1991, had enabled members of the PAM Project at Digital's Paris Research Laboratory to build coprocessors with DMA performance of 80-90 MB/s [3]. Although PCI promises theoretical peak performance of 133 MB/s, it is clear from the most cursory reading of the PCI Specification [4] that PCI is a far more complex bus than TURBOchannel and that it allows many more opportunities for implementations to introduce wait-states and other performance degrading artifacts. The first PCI chipsets confirmed our fears with many developers reporting never seeing performance above 30 MB/s. Nevertheless PCI has rapidly achieved total market dominance as the first tier expansion bus in all PC and Macintosh compatibles, many workstations and even some high-end servers, so PCI is the bus of choice for performance oriented reconfigurable coprocessors that seek to use a standard bus.

Apart from high-end graphics, few current devices can actually use 133 MB/s of throughput; reconfigurable coprocessors however are an exception. Therefore a significant part of the design effort in PCI Pamette has gone into understanding PCI performance: what is achievable, and how to achieve it. From the outset, system exercising and measurement were seen as important applications of PCI Pamette. We had already used reconfigurable technology to measure TURBOchannel performance, and understood its value [3]. The complexity of PCI and disappointing early performance figures obtained by others made repeating these sorts of measurements on PCI even more compelling.

PCI has an aggressive electrical specification making it challenging to implement in anything but a custom chip. However, we quickly ruled out the use of a commercial PCI interface chip; all were flawed in one way or another and seemed unlikely to offer reasonable performance. In any case PCI Pamette is a 64-bit PCI board and no commercial interface is 64-bit. Thus the logic of PCI Pamette is implemented entirely from FPGAs, *including* the PCI interface, and all can be reconfigured in circuit.

This paper describes a collection of simple designs implemented on PCI Pamette, which are used to gather performance figures. For each of these designs, sample results are provided, giving an idea of the sorts of results obtainable, and how they may vary. The platforms used are Intel-based PC compatible systems running Windows NT, and Alpha-based workstations and servers running Digital Unix and Windows NT. The paper's contributions are three-fold:

- Measurement and bus exercising is an application domain in its own right; we show the utility of reconfigurable computers in this domain. This application domain has the added utility that applications in other domains can easily reuse the PCI interfaces developed in the bus exercising applications.
- The results themselves are of use to architects of other PCI-based reconfigurable systems.
- Our results can also help the design and test of conventional fixed circuit PCI interfaces and thus our work represents an instance of emulation by FPGAs. Our use of an easily reconfigured FPGA based system makes it easy to embed the instrumentation circuitry for measurement in the same reconfigurable board that is performing the emulation.

In section 2, we present a simple application that spies on PCI bus traffic. As it is fully passive, it is not system intrusive and can be used to monitor full PCI transactions like an in-system logic analyzer.

In section 3, we use a fully programmable PCI master/slave engine, which, associated with section 2's passive spy, allows the user to test the maximum bandwidth of his system in each of its operating modes, and to tune the parameters of his application to get the best from his system. This setup can also be used to test prototype boards and PCI-to-PCI bridges.

In section 4, we describe an enhanced and open version of the passive spy that can be adapted to specific needs. We offer two examples.

In section 5, we show a simple setup that tests the interrupt latency figures of a specific Machine/Operating System pair. These figures are particularly interesting for real-time applications.

2 Simple passive bus spy

2.1 PCI basics

The PCI bus is designed to be versatile, adapted to high-end servers as well small PCs. As such, it does not make any assumptions on a device's ability to sustain a continuous transaction for any number of cycles. We can observe on the same bus a very long transaction with a 32-bit word transmitted every cycle, followed by a single byte transaction lasting 30 cycles.

Here are some details of the PCI bus, which will help to better understand the rest of the article:

- All data transfers are broken into *transactions*. Each transaction is between an *initiator* and a *target*. The host bridge (processor to PCI interface) is just one of the devices on the bus. Each device responds to a certain (or to multiple) *address space* assigned individually at boot time.
- The initiator *requests* a transaction, and when it gets a *grant* from the *arbiter*, it sends an address and *command* (transaction type) and waits for some target to respond. The data transfer can then happen in either direction (from the initiator to the target or from the target to the initiator), depending on the command that was issued.
- At any cycle, either the target or the initiator can declare they are not ready to accept or deliver data, thus introducing a wait cycle.
- After any amount of data already transferred, both the initiator and the target can decide to stop the transaction. In particular, the target can decide to *retry* the transaction before any data has been transferred, because it is temporarily unavailable, or to *disconnect* the transaction after some data has been transferred when it will not be ready again within PCI's latency requirements.
- PCI clock frequency can vary dynamically from DC to 33 MHz. However most system keep the clock frequency around 33 MHz. All our calculations and measurements assume a 33 MHz clock.

From a bandwidth perspective, the best case occurs when the transfer between the target and the initiator starts quickly after the address cycle, data words are sent every cycle for a long time, and the master stops the transaction when it has transferred all the data it wished to transfer. The result is a near optimal use of the bus bandwidth, allowing up to 130 MB/s on a 32-bit PCI and 260 MB/s on

a 64-bit PCI. Sometimes the maximum length of a transfer needs to be limited to reduce overall latency seen on the bus, but in reality PCI performance is most likely to be limited because devices exhibit one or more of the following features:

- They are often not ready and *retry* frequently (e.g. PCI-to-ISA bridges).
- They have a very long latency between the address cycle and the initial data cycle (this is especially true for host bridges when required to read host memory).
- They can not send or receive data on every cycle.
- They always stop a transaction after a (small) fixed amount of data has been transferred.

When two devices have to exchange data, these features combine and the result can be most disappointing. Moreover, many devices have bugs, and are configured to operate in degraded performance modes, where, although they actually function correctly, performance is very poor. On most systems, PCI board performance ranges from 5 to 30 MB/s. On the same systems, it is usually not hard to get 80 to 120 MB/s with a well designed and specifically tuned PCI interface.

2.2 Implementation of the spy

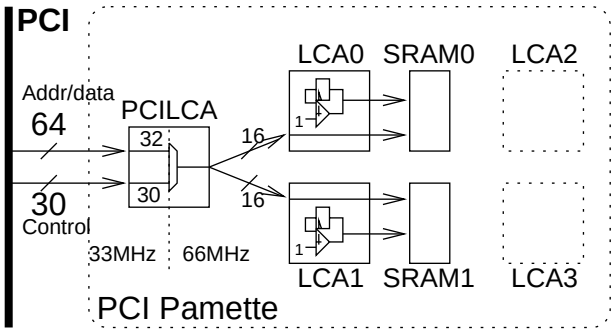

Figure 1: Simple spy

The passive spy implemented on PCI Pamette uses the straightforward design shown in figure 1. PCI Pamette consists of a 2×2 matrix of FPGAs called the *user area* and a fifth FPGA called the *Pcilca*. Since PCI Pamette is based on Xilinx FPGAs, we use the terms FPGA and LCA interchangeably is describing the board. The PCI interface chip has a special promiscuous mode where all the PCI control signals (30 seen by the board, including all the interrupts and 64-bit PCI control signals) and 32 bits of the address/data bus (out of 32 or 64 depending on the motherboard) are sent to the user area (multiplexed in time: 32 bits @ 66 MHz). In the user LCAs, a simple counter counter increments the external SRAMs address lines, while data is sent to the data lines (there are two banks of 64k $\times$ 16, 12 ns SRAMs, which provide the necessary 32 bits @ 66 MHz bandwidth). To read the data, we just download a new configuration containing a standard SRAM-reading design, and analyze the results.

2.3 Use of the spy

To use the spy to look at transactions on the PCI, we simply prefix the code that starts these transactions by a write to the board which triggers data collection. After a short delay, we can download the SRAM-reading design to see what has happened. This simple mode can be used for instance to watch the types and lengths of transactions issued by a graphics or a SCSI card.

3 Traffic generation and performance analysis

3.1 PIO slave and DMA engine

To actually obtain the maximum performance and tune PCI transactions, we have to add a traffic generation scheme to the simple passive spy. We have focused on two kinds of transactions:

- Programmed Input/Outputs (or PIOs) are initiated by software. They consist of reading or writing values to/from a board. In this case, the host bridge is the initiator and the board is the target.
- Direct Memory Accesses (or DMAs) are initiated by the board and transfer data to/from host memory. The board is the initiator and the host bridge is the target.

.

PIOs are generated by a simple piece of code that accesses the board, and of course the *slave* part of the board that responds to the requests. For these tests the interface has been programmed so that the response of the slave to 64-bit requests depends on the target address used. Thus, we can choose 32-bit or 64-bit transactions directly from software (on 64-bit PCI systems only of course).

DMA traffic is generated by a full *PCI master* which has been programmed into the Pcilca. Contrary to most DMA engines, this one must be extensively programmable, so that we can test many PCI transaction parameters. The test DMA engine in PCI Pamette is capable of generating transactions with the following attributes:

- Full set of PCI command codes: memory or legacy I/Os, read line, read multiple, write and invalidate...).
- Total amount of data to be transferred.
- Size of the bursts (after which the engine stops a transaction, if the target has not already done so).
- Number of idle cycles between transactions.
- Use of 64-bit requests.

Using the passive spy of the preceding section, we can generate traffic and read the full transaction log. We can then very precisely analyze the results, finding patterns, trying to tune the parameters to achieve better performance.

3.2 Single-board performance tests

As the passive spy does not use any resources in the PCI interface LCA, we can fit both the PCI master and target engine and the spy on the same board. We can then use an automated program which can test all the characteristics of a system by generating all kinds of traffic and modifying parameters. Figure 2 shows examples of performance measured on some Alpha-based workstations and servers and Intel-based PC compatibles.

Four kinds of performance figures are interesting:

3.2.1 DMA Write (board writes to memory)

This is usually the most impressive figure. As address and data are sent in same the direction, latency is usually small. For memory systems which are wide enough, the asymptotic bandwidths of 130 MB/s or 260 MB/s on 64-bit PCI can be reached. Nevertheless, there can be behaviors which degrade performance considerably. Older host bridges often disconnect any transaction at cacheline or double cacheline boundaries; in such cases, the board's interface should be programmed to do bursts of this size exactly. Sometimes a small number of idle cycles should be inserted before next request for the bus, to ensure that the host bridge is ready in time for the next burst and will not retry the transaction. Some host bridges allow only the PCI *memory write and invalidate* command to perform long bursts and will disconnect ordinary *memory write* commands. Other more common problems include disconnects or wait states inserted every 4th or 8th data word because the memory system is saturated, or the internal buffers of the host bridge are full. On an AlphaStation 200 or 400, the host bridge disconnects after 16 data words, and we can reach 107 MB/s. On a simple PC with a 80430FX (aka. Triton I), we can get 124 MB/s. On an AlphaServer 4100 (64-bit PCI), we can get 260 MB/s. On an AlphaStation 500, though, 64-bit DMA Writes are only marginally faster than 32-bit mode. The designers of this workstation made an explicit decision that $\sim$100 MB/s was more than adequate for disk and network I/O [5], and focused most effort on support for graphics whose primary bandwidth requirement is from the host to the PCI option (DMA Read and PIO Write).

3.2.2 DMA Read (board reads from memory)

This kind of transaction resembles DMA Write except that, as it involves a roundtrip to the memory system before first data can be returned, the latency to initial data can be very large. On high-end servers like the AlphaServer 8400, where the memory is in a different subsystem to the PCI host bridge, the latency is many tens of PCI cycles, even though the bandwidth of the memory backplane is 2 GB/s. On our Pentium Pro 200, the memory latency averaged 35 cycles, with values up to 60 (in spite of the fact that version 2.1 of PCI specification states that this value should never be greater than 32). On DMA Read, it is especially critical to do long bursts, to amortize the cost of the memory latency. The PCI protocol provides three different flavors of memory read command which provide hints to the target of how long a transaction is likely to be that it can optimize the prefetch of subsequent data. On most platforms that support long bursts it is essential to use these hints to sustain a long burst. Platforms systematically disconnecting transactions like the AlphaStation 200 or 400 can reach only 67 MB/s, while standard long bursting platforms reach 120 MB/s to more than 200 MB/s for 64-bit PCI machines.

3.2.3 PIO Write (processor writes to board)

There is little one can do to optimize the PIO Write performance on the target side, except respond quickly and accept data every cycle. PIO Write performance depends mainly on the ability of the processor and the host bridge to aggregate writes. A program can only issue 32 or 64 bit writes with a single machine instruction. If the writes are sequential or on the same region, the processor and the host bridge can aggregate the writes in order to issue a longer burst, thus amortizing the overhead of starting a PCI transaction. Alpha and Intel platforms are very different in this respect. On Alpha, the writes can be aggregated, reordered and even merged, and the processor may delay a write for as much as 100 CPU cycles in the hope that other writes will be performed to nearby locations and these may be aggregated with the pending write before it is sent off chip. The result of this policy is that the ordering of writes must be explicitly managed by the

Platform	CPU & Host bridge	DMA Write	DMA Read	PIO Write	PIO Read
AlphaServer 4100 5/300	21164/300 w/ custom bridge	130 (260)	120 (240)	83 (122)	47 (58)
AlphaStation 500/400	21164/400 w/ 21171	90 (95)	126 (200)	88 (132)	47 (58)
AlphaStation 500/333	21164/333 w/ 21171	82 (88)	130 (200)	96 (151)	54 (69)
AlphaStation 200 4/166	21064/166 w/ 21071	107	68	81	24
HP Vectra XU 6/200	Pentium Pro 200 w/ 80450KX	129	110	19	7
Digital Celebris GL 6200	Pentium Pro 200 w/ 80440FX	130	125	29	10
Digital Celebris GL 5133	Pentium 133 w/ 80430FX	124	130	65	15
Digital Proris XL Server 590	Pentium 90 w/ 80430NX	68	52	54	13

(All figures in MB/s. Where available, 64-bit PCI performance in parenthesis).

Figure 2: Maximum sustained bandwidth on selected systems.

machine code when order is significant (for instance in the face of order-dependent side-effects of writes to hardware devices), but performance can be fairly high. On Intel platforms, the writes to a PCI board are strongly ordered, never merged, and aggregated only if they are perfectly sequential and sent within a very short time period: on a Pentium 133, a standard C `for` loop writing to a board can only provoke 8 or 12 byte bursts whereas tight assembly code can lead to 16 byte bursts. PIO performance on Intel machines is usually fairly poor if no special optimization is used. On most 80450KX- (aka. Orion for workstation) based Pentium Pro machines, write aggregation is completely disabled because of bugs in some revisions of the chipset which can lead to data corruption. On these machines, the best PIO Write performance can be under 20 MB/s.

3.2.4 PIO Read (processor reads from board)

This kind of transaction is usually slow and should be avoided for large data transfers. In most cases, PIO Reads end up as single memory quantum reads (32 bits on an Intel, 64 bits on an Alpha). Exceptions are the recent alpha systems based on the 21164 CPU. The 21164 is able to support multiple outstanding reads, They can be aggregated and presented off chip as a single transaction. A tight loop of sequential reads on a 21164 based machine generally produces a stream of 16 byte bursts, giving, on the AlphaServer 4100 for instance, about 37 MB/s throughput. One might imagine this cannot be improved upon, but it turns out that a load sequence of the first word of a 32 byte block, then the last word, then the intervening words, usually generates a 32 byte burst on this platform, allowing us to reach as much as 69 MB/s in 64-bit mode.

3.2.5 No "One size fits all"

Other factors too numerous to be given a full treatment in this paper can affect PCI performance. For instance on versions of the AlphaServer 4100 with a 300 MHz CPU (but not with a 400 MHz CPU) if data is in the processor cache, 64-bit DMA Read performance can be degraded (but not 32-bit DMA Read performance). On the AlphaStation 200 or 400 on the other hand, the presence of data in the processor cache can boost performance by as much as 30%.

In section 2 we have presented the best figures we have been able to achieve. It is obviously difficult to forecast the actual bus bandwidth a certain board can obtain on a given platform, and matters get only worse if we begin to include application specific constraints. On PCs, PIOs are usually best avoided for large transfers, whereas PIO Writes can be largely sufficient on Alpha platforms. Such considerations often determine major architectural decisions, affecting the hardware and driving software, in many PCI Pamette applications.

3.3 Debugging setups

The combination of a simple passive spy and a traffic generator can be used as single-board performance tester, but it can also be used as a debugger for PCI interface chipsets. Many schemes can be envisaged, some of which are actually used within Digital:

- With a single board, we can spy on another specific board, provided it is on the same PCI bus as the PCI Pamette. We can generate peer-to-peer traffic, exposing the other board to any kind of transaction, emulating the behavior of almost any conceivable host bridge. We can also keep very long traces using DRAM and verify that no protocol violations have occurred.
- With two boards connected by a flat cable, each on a different system or at least on a different bus, we can

use one as a spy, and the other as a means to communicate the results in real-time. This setup results in a non-intrusive logic analyzer with a real-time, high-bandwidth link to another system.

- With a two-board setup, we can also debug special PCI devices like PCI-to-PCI bridges. We can put one board on each side of the bridge and generate traffic through the bridge, while getting a trace from both sides.

All these setups are minor variations of the basic PCI spying and exercising configuration. They exploit the reconfigurability of the boards because any special feature (like time-stamping) can be implemented in a matter of hours or days by a moderately experienced PCI Pamette programmer.

4 Application-specific bus spy

4.1 Basic design and possible extensions

The simple spy of section 2 is useful for cycle by cycle off-line analysis, but real-time, on-board trace computation is also useful, and needs a much more sophisticated design. For low-level PCI spying, a custom version is needed, but for most needs, a common front-end, hiding the PCI complexity, can form the basis of a variety of application specialized measurement applications. We implemented a core design that receives the full PCI signals, like the simple spy, and which computes for each PCI transaction a compact summary of its principal characteristics:

- The starting address.
- The transaction type.
- The number of words transferred.
- The address-to-first-data latency.
- The total number of wait-states.
- The total length of the transaction.
- How the transaction was terminated.
- A time-stamp of the transaction.

With the summary produced by the front-end, we can customize the spy to more specific needs by adding an application specific subcircuit. We can then use the rest of the resources of the board (remaining programmable logic gates, SRAMs and DRAMs, daughter board connector, PCI interface…) to implement the functions we need. Among the extensions one may want to add are:

- Programmable filters. Filtering on the address allows us to restrict attention to all traffic to/from a certain device. By getting the base address of the board from the system, we can get all transactions which have the chosen board as target. With some cooperation from the chosen board's driver, we can get the PCI-mapped address of the memory buffers used by the board and thus spy on DMAs initiated by the board.
- Real-time performance counters. By filtering the traffic related to a specific board and using it to maintain running totals of various transaction characteristics we can calculate in real-time: the amount of data transferred to/from that board, the PCI bus loading attributable to the board, the average latency, the average time between two transactions, and various other histogramming type measures.

4.2 Example 1: the PC's Real-Time Clock

A very simple application requiring only the implementation of a programmable comparator and mask is a spy that logs all transactions to a specific device (recognized by its base address). Using a legacy device's fixed address in I/O space or modern device's settable address in memory space, we can capture all the PIOs to the device.

Figure 3 shows the use of such a filter on address 70 and 71 (hex) in I/O space, which corresponds to a PC's Real-Time Clock (RTC). This device sits on the ISA bus, on the other side of a PCI-to-ISA bridge. With this log, we start to understand the true burden of ISA transactions carried through the PCI bus. This log comes from an AlphaStation 200 running Digital UNIX, but it uses a standard PC-like hardware configuration.

The PCI cycle deltas in the right-most column of the log confirm that it is indeed the RTC that we have logged; transactions appear in groups at intervals of approximately 32000 PCI cycles. At 33 MHz, this period corresponds to about 1 ms; under Digital UNIX the RTC is configured to interrupt at 1024 Hz. We see that the usual initial data latency for reads is 39 cycles (once again we find ourselves observing a device that is not PCI version 2.1 compliant). The total transaction lasts for more than 40 cycles, for a single byte transferred. For writes, the average latency is 33 cycles, but there is more going on. Transaction #00077 tries to write a single byte to address 70, but the PCI-to-ISA disconnects after 3 cycles with no data transferred. The write has been *retried* and must repeat until it is actually done. In this example, it takes 8 transactions (7 retried) for a total of 41 cycles to complete. This is an excellent example of how bad standard ISA devices can be on a PCI system and how the reliance on legacy buses can degrade performance at all levels of the system.

Trans.	Addr.	R/W	Lgth.	Dur.	Lat.	Δ Cycle#
00071	0070	W	1	35	33	32302
00072	0071	R	1	41	39	42
00073	0070	W	1	33	31	32332
00074	0071	R	1	41	39	40
00075	0070	W	1	33	31	32344
00076	0071	R	1	41	39	40
00077	0070	W	0	04	03	32329
00078	0070	W	0	03	02	6
00079	0070	W	0	03	02	10
00080	0070	W	0	03	02	5
00081	0070	W	0	03	02	5
00082	0070	W	0	03	02	5
00083	0070	W	0	03	02	5
00084	0070	W	1	35	33	5
00085	0071	R	1	41	39	42
00086	0070	W	1	36	34	32293
00087	0071	R	1	41	39	43
00088	0070	W	1	33	31	32348
00089	0071	R	1	41	39	40
00090	0070	W	1	35	33	32322
00091	0071	R	1	41	39	42
00092	0070	W	0	04	03	32337
00093	0070	W	0	03	02	6
00094	0070	W	0	03	02	15
00095	0070	W	0	03	02	5

Figure 3: Simple filter at I/O address 7x (Real-Time Clock)

4.3 Example 2: the Memory Channel spy

Memory Channel [6] is Digital's high-end cluster interconnect; it is used to tightly couple several servers through a high-speed link. The hardware component of this product consists of a full-length PCI board in each machine, and an active hub to exchange the data between the boards. Applications use a standard API to send or receive data through the Memory Channel system.

The data transfer model chosen in this system is a *push model*: data is sent to the local adapter through PIOs by software, the data then goes to the hub and is dispatched to targets. On the targets, the adapter sends the data by DMA to a specific place in the host memory. From the software point of view, a part of each other machine's address space is mapped into its memory space, and a part of its own physical memory is mapped into the address spaces of the other machines. Writing to a certain part of the memory automatically writes to the physical memory of other designated machines. Memory Channel also provides locking, error recovery and interrupt mechanisms.

The PCI Pamette can be used as a non-intrusive spy in Memory Channel systems. By dividing the address space into a number of fixed regions and keeping histograms on each of these, traffic to particular regions of Memory Channel address space can be identified. With more effort, the PCI Pamette can be programmed to know about the Page Control Table kept by the MC adapters (which selects the destination and mapped address of a local or remote page in the mapped space), and can use the same control information received by the MC adapter to maintain a copy of the PCT. With additional information from the software on the subsystems (disk, database, failure recovery…) associated with each PCT entry, it can produce real-time or off-line histograms on the use of each subsystem. This can be used to physically optimize a cluster and also to optimize the code of the client (for instance guaranteeing that PIO Writes undergo the maximum possible aggregation in Alpha write buffers so that scarce PCI bandwidth is better utilized).

The advantage of such an approach is that there is no need to instrument software (which would distort the true response of the system), there is no need either for any hardware modification of the MC adapters, and the spy can evolve with the hardware and with the needs of the customers without any physical change.

5 Interrupt dispatch latency and real-time capabilities

5.1 Motivation

Since all PCI Pamette applications execute in part on the host processor and in part on the PCI Pamette we need mechanisms to synchronize the two. PIOs allow the processor to poll the state of PCI Pamette and signal its own state to it with latencies measured in fractions of a microsecond. For some applications this is all the synchronization that is needed, but many PCI Pamette applications have strong real-time requirements. The PCI Pamette host runs a multitasking operating system, real-time applications cannot be at the mercy of the operating system scheduler, nor can they tie up the processor with a busy polling high priority process. PCI Pamette needs a mechanism to initiate synchronization with the processor. Unfortunately DMAs do not perform for PCI Pamette what PIOs do for the host processor. Instead, interrupts must be used.

Interrupt latencies are much longer and far, far more variable than PIO latencies. Characterizing these latencies is essential to designing applications that will meet their real-time goals. Two aspects are important, the average cost of handling an interrupt which will determine to overall load placed on the system by a given interrupt rate, and the maximum interrupt handling latency which will determine the amount of buffering needed while the PCI Pamette is waiting for interrupt service.

There are several levels where interrupts are handled (all the terms used here are specific to Windows NT [7] , but the overall interrupt scheme is common also to Unix systems):

- In the Interrupt Service Routine, where the kernel-mode device driver has to decide whether its board is responsible for the interrupt (interrupt lines are shareable on PCI), shut it off, and do some small amount of processing related to the interrupt. The ISR runs at a very high-level priority and should not execute for long as the system is stalled while it is running.
- In the Deferred Procedure Call, still in kernel-mode, but at a much lower priority level, where the driver finishes the tasks associated with the interrupt (often setting up a new DMA).
- In some user-mode code called by the driver using a blocking I/O call, where some further processing can be done.

Interrupt handling in standard boards occurs mainly at the first two levels. However, as PCI Pamette in a reconfigurable board, we have sought to make the device drivers supporting it completely generic as we do not want to oblige every PCI Pamette application developer to write kernel-mode non-portable code. All control of the board is relegated to user-mode code which has direct access to board. In particular, the interrupt signals are routed to the user-mode application that requested to receive them. The driver does a minimum of generic actions, which consist of checking the source of the interrupt and shutting it off. It then queues a request for a Deferred Procedure Call that unblocks the user-mode thread which is waiting for the interrupt.

There is unfortunately one big disadvantage to this approach, namely the time necessary to wake up a user-mode thread, which is always longer than processing the interrupt directly from kernel-mode. The interrupt dispatch time depends mostly on the operating system (scheduling policy, maximum locked times…) and on other device drivers (time spent at hardware priority levels…).

5.2 Setup and results

The design on the board that is used to test the interrupt latency is fairly simple. It contains a free-running settable counter which triggers an interrupt when its value crosses zero. This mechanism allows us to record the time when:

- The interrupt was triggered.
- The Interrupt Service Routine started and ended.
- The Deferred Procedure Called started.

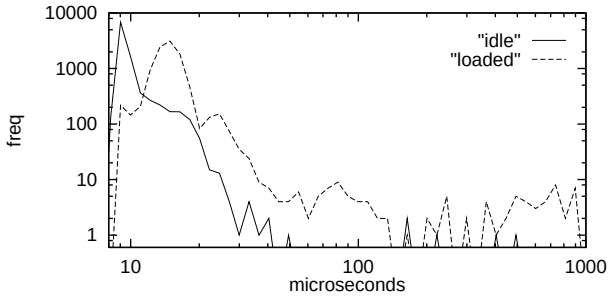

Figure 4: Histogram of time to ISR (Pentium Pro 200 w/ Windows NT 4.0)

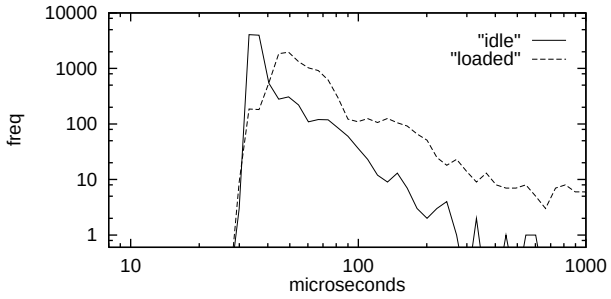

Figure 5: Histogram of time to user handler (Pentium Pro 200 w/ Windows NT 4.0)

- The user-mode thread was unblocked.

It is important to trigger the interrupt pseudo-randomly using the free-running counter since it allows our measures to more accurately reflect the ensemble of system activity. Figure 4 shows a histogram of latency between the hardware interrupt and the entry point of the ISR for 10000 interrupts on an idle and loaded[1] system. Figure 5 shows the corresponding data for the unblocking of the waiting user-mode thread. All axes are logarithmic scales, all times above 1ms have been truncated.

The figures presented have been obtained with a thread at maximum priority (31 or “real-time time-critical” [8]) on a HP Vectra XU 6/200 (Pentium Pro 200) running Microsoft Windows NT 4.0. The same tests can also be applied to Alpha systems running Windows NT or Digital Unix.

The histograms exhibit sharp peaks. Even on the loaded system, in 98% of cases the ISR is called within 30 microseconds and the user-mode thread starts to run within 200 microseconds. From the position of the peak of user-mode thread unblocking latency on an idle system we can estimate the kernel-mode cost of a typical interrupt. It appears to be 30-40 microseconds, so the system should sup-

[1] After some experimentation the most effective way we found to *load* a Windows NT 4.0 system was to rapidly drag a large window.

port an interrupt load of several thousand per second. However the tail is long. Times to ISR of over 1 millisecond and times to the user-mode thread in excess of 3 milliseconds have been observed, so PCI Pamette applications with strong real-time requirements need to budget for at least this much autonomy on this platform.

6 Conclusion

We have presented a number of applications of a reconfigurable computing platform called PCI Pamette that come under the general heading of system exercising and measurement. Our results have a particular focus on the PCI bus.

We have demonstrated practical techniques to measure parameters of live systems. Some of the measurements could perhaps have been done more conventionally with a logic analyzer, although in the course of writing this paper the authors have personally gathered results on systems in Palo Alto CA, Merrimack NH, Maynard MA, Paris France, Sydney Australia and Ayr Scotland. Neither author has ever been to Ayr, but a PCI Pamette has, and while there it was attached to a computer on the internet. The use of reconfigurable technology has given a software aspect to an application that traditionally involves direct physical manipulation. The active and application specific measurements in sections 3 and 4 vigorously exploit reconfigurability and would be much more difficult to perform with conventional techniques. The interfaces we have developed and modes of operating them to achieve best performance can be reused in other applications of PCI Pamette that involve the *use* rather than the *measurement* of the PCI bus.

The data we have presented goes a little deeper than the common presentation of PCI as a 33 MHz bus with peak throughput of 133 MB/s. It is however understandable that manufacturers are reluctant or unable to accurately characterize the performance of their devices. PCI performance is a complex issue and no device can be treated in isolation. We trust our results will be useful to designers and users of PCI based systems in the reconfigurable computing community for whom PCI performance is an important issue.

Lastly PCI Pamette has served as a traffic generator, emulating PCI devices with various performance characteristics during the test and debug of conventional fixed circuit PCI interfaces and systems.

7 Acknowledgements

This work owes an intellectual debt to Jean Vuillemin founder of the PAM Project. For PCI Pamette, Didier Roncin, Philippe Boucard and Patrice Bertin all helped in the development of an early prototype.

References

[1] Mark Shand *PCI Pamette V1* World-wide-web http://www.research.digital.com/SRC/pamette 1996 and 1997.

[2] Jean Vuillemin, Patrice Bertin, Philippe Boucard, Didier Roncin, Mark Shand, Hervé Touati *Programmable Active Memories: Reconfigurable Systems Come of Age* IEEE Transactions on VLSI Systems, April 1996.

[3] Mark Shand *Measuring Unix Kernel Performance with Reprogammable Hardware* PRL Research Report #19, Aug 1992. ftp://ftp.digital.com/pub/DEC/PRL/research-reports/PRL-RR-19.ps.Z

[4] *PCI Local Bus Specification 2.1,* PCI Special Interest Group, 1995.

[5] John H. Zurawski, John E. Murray, and Paul J. Lemmon *The Design and Verification of the AlphaStation 600 5-series Workstation* Digital Technical Journal, Volume 7, Number 1, Special Edition 1995.

[6] R. Gillett and R. Kaufman, *Experiences Using the 1st-Generation Memory Channel for PCI Network,* Hot Interconnects Symposium IV 1996, Stanford University, Palo Alto (CA).

[7] H. Custer, *Inside Windows NT,* Microsoft Press, 1993.

[8] *Real-Time Systems and Microsoft Windows NT,* MSDN Library, Microsoft Corporation, June 1995.