PEX 8619 DMA Performance Metrics

16-Lane; 16-Port Gen 2 PCIe Switch with Integrated DMA Engine

1 Introduction

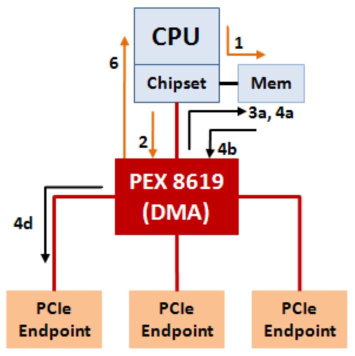
This white paper discusses various options to program the integrated DMA engine on the PEX 8619 switch for performance. After a brief overview of the DMA engine, this document provides guidelines for programming on-chip registers to get the best performance for an application. The document concludes with a description of tests run in the lab that apply the recommended programming. In summary, it was found that the integrated DMA engine in the PEX 8619 is capable of sustained throughput of over 1.4 GB/s in each direction on x4 links with SerDes speeds of 5.0 GT/s, and 2.9 GB/s in each direction on x8 links with SerDes speeds of 5.0 GT/s, both of which get over 99.5% link utilization.

2 DMA Overview

DMA is a method to move data without tying up the CPU. The CPU sets up DMA descriptors to move data, turns on the DMA and then is free to work on other tasks while the DMA engine moves the data. The DMA engine can optionally interrupt the CPU at various stages to give a progress report.

A descriptor instructs the DMA engine what to move: it provides a read address, a write address, and a transfer size. The DMA engine acts on the descriptor by reading at a specified start address, handling the resulting read completions, and then writing to the specified destination address. This continues until all the data has been transferred, and then another descriptor is started.

The following diagram illustrates the sequence of operations for a DMA. In this diagram, the DMA moves data from system RAM (Mem) to an endpoint. The DMA descriptor could be modified to move data from an endpoint to Mem, or from Mem to Mem, or from endpoint to endpoint. The DMA engine simply reads the data from and writes the data to anywhere the descriptor addresses instructs it.[1]



1. CPU programs descriptors in Mem
2. CPU enables DMA
3. DMA reads descriptors in Mem
   a. DMA prefetches 1 – 256 descriptors
4. DMA works on 4 descriptors at a time
   a. DMA reads source – e.g. Mem
   b. Completions arrive in switch
   c. Completions are converted to writes
   d. DMA writes to destination – e.g. PCIe Endpoint
   e. Repeat read/write to satisfy descriptor
   f. After last write, invalidate descriptor (option)
   g. Interrupt CPU per descriptor (option)
   h. Start next descriptor
5. When done with descriptors, stop (DMA done) or loop
   a. Interrupt CPU after last descriptor (option)
6. CPU handles interrupts (if enabled), may restart DMA

Figure 1. System Level DMA Flow

[1] *The DMA cannot hit internal register memory space.*

The PEX 8619 has four DMA channels; each can be enabled independently. However, all four channels share the same resources, so if only one or two channels are needed, the DMA should be configured to be in one or two channel mode to get more on chip resources.

There are three methods that the DMA engine can get descriptors:

- **block mode** : a single descriptor is programmed directly into CSRs
- **on-chip descriptor mode**: an on chip descriptor RAM is programmed with one or more descriptors
- **off-chip descriptor mode**: descriptors are programmed off chip (system Memory typically) and the DMA engine will prefetch possibly multiple descriptors to hide the descriptor fetch time from the DMA data movement.

2.1.1 Software Interface

The following diagram shows a software view of a switch with DMA. A switch is composed of multiple PCI to PCI (PCI-PCI) bridges. The DMA controller will be seen as a Type 0 Endpoint on Function 1 of the upstream PCI-PCI bridge on the upstream port.

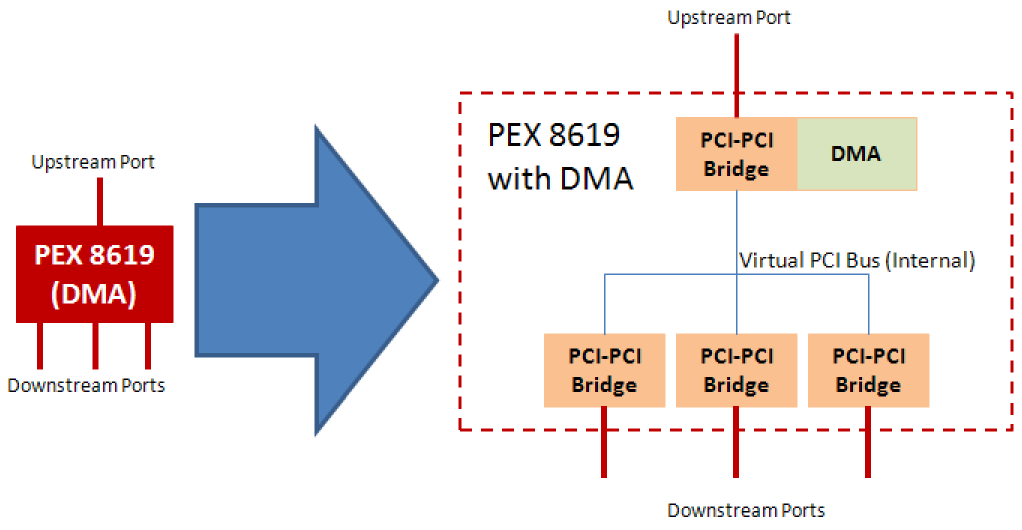


Figure 2. Software View of Switch with DMA

This hierarchy needs to be understood especially when using DMA with Non-Transparent (NT) port. When using DMA and NT, the DMA engine's requester ID needs to be programmed in the requester ID look up table to enable DMA to read through the NT port and get completions back. The requester ID is composed of {upstream_bus[7:0], device_number[5:0], function[2:0]}. For the PEX 8619 DMA engine, the device number is 0 and the function number is 1. You will need to determine the upstream bus number by reading register 0x2C8 bits [7:0] to get the primary bus number of the upstream port.

Each of the PCI-PCI bridges has a 4KB on chip register space. In addition, the Upstream Device DMA Function 1 claims 128KB of internal resources, with only 8KB used and the rest reserved. The first 4KB are used for DMA registers and the next 4KB are used for on-chip descriptor RAM. The memory-mapped DMA register space starts at offset 0x0 from the address contained in the BAR0/BAR1 registers of the DMA function.

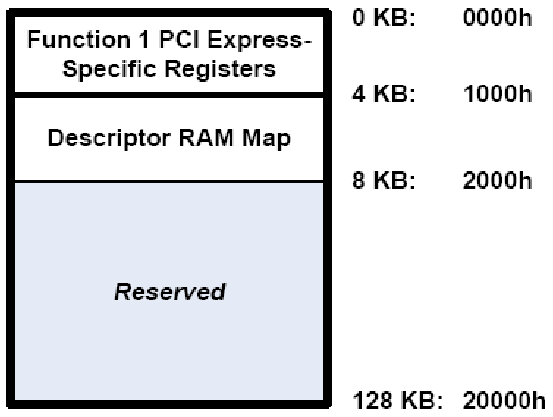


Figure 3 DMA BAR0/BAR1 Window

The following tables provide a quick reference to all registers involved in DMA. The global registers are programmed one time for all channels, and the channel registers must be programmed independently for each channel.

An offset map for the DMA channel registers is listed in Table 1.

Table 1. DMA Channel Register Map

DMA Channel	Registers Begin at Offset
0	0x200
1	0x300
2	0x400
3	0x500

Table 2 is a detailed register list for a DMA Channel (0).

Table 2. DMA Channel Register List (DMA Channel 0)

Register Offset	Description
0x200	source address lower 32 bits
0x204	source address upper 32 bits
0x208	destination address lower 32 bits
0x20C	destination address upper 32 bits
0x210	transfer size on 27 bits
0x214	descriptor ring base address lower 32 bits
0x218	descriptor ring base address upper 32 bits
0x21C	next descriptor address lower bits (same as base when start)
0x220	descriptor count
0x224	read only status: current descriptor pointer
0x228	read only status: current descriptor transfer size
0x22C	DMA bandwidth control via read to read delay
0x230	do not use
0x234	prefetch limit
0x238	channel control: start, pause, abort, etc
0x23C	interrupt enable and interrupt status
0x240	error header log DW0
0x244	error header log DW1
0x248	error header log DW2
0x24C	error header log DW3
0x250	error type
0x254	6bits control maximum number of data reads override

Table 3 below lists the Global DMA registers and Table 4 lists the DMA Channel registers which were programmed in the performance tests discussed in this white paper. **Note: the registers are listed in sequence as programmed.**

Table 3. Global DMA Registers (All channels)

Register Offset	Description
0x70	extended tag enable on bit 8
0x1FC	number of DMA channels, on/off chip descriptors

Table 4. DMA Registers Used

Register Offset	Description
0x214	descriptor ring base address lower 32 bits
0x218	descriptor ring base address upper 32 bits
0x21C	next descriptor address lower bits (same as base when start)
0x220	descriptor count
0x22C	DMA bandwidth control via read to read delay
0x234	prefetch limit
0x23C	interrupt enable and interrupt status
0x238	channel control/status: start, pause, abort, etc

3 Round Trip Time for a Read Request

Round Trip time for a Read Request refers to the time it takes from sending a read to getting a read completion. In order to maximize throughput, the DMA engine needs to have a constant stream of completion data that it can forward as a constant stream of write data. To achieve this, the read round trip time must be covered by the DMA sending enough adequately sized reads.

For example, if the DMA could only send a single 4B read, it could only cover the time needed to send 4B on the PCIe wire. At 5.0 GT/s and a PCIe link width of x8, 4B can be sent in 1 ns. If the read round trip time were 500ns, the single 4B read would only get 1/500th of the possible throughput on the PCIe wire.

To get better throughput, the DMA must either use a larger read size or have more reads outstanding (or both). In the example of 500ns read round trip time, one 512B read, or two 256B reads, or four 128B reads, all would suffice to cover the read round trip time and get near ideal PCIe throughput. The PEX 8619 DMA allows programming of both the read size and the number of outstanding reads, giving the user control over how to mask the read round trip time and achieve maximum throughput.

4 Programmable Read Size

The maximum read size is programmable per channel. For Channel 0 as an example, programmable options include 4B, 64B, 128B, 256B, 512B, 1024B, and 2048B in the ***DMA Channel 0 Control/Status Register (0x238[18:16])***. DMA Source Max Transfer Size 000=64B; 001=128B; 010=256B; 011=512B; 100=1KB; 101=2KB; 111=4B; others reserved

The PEX8619 DMA restricts the *maximum* read size to the PCIe maximum payload size (MPS). This is because the completions for a single read are combined to make a single memory write, and the memory write is constrained by the MPS. PCIe spec requires a minimum value for the MPS of 128B, so this value is always safe. Many systems support larger MPS for correspondingly higher throughput possibilities.

The read size of 4B is set to allow DMA to read the memory mapped register space of many PCIe devices that only accept 4B read requests. A single descriptor can be used to read multiple registers.
Another control of the read size is the descriptor itself. If the transfer size is smaller than the target read size, only the transfer size number of bytes will be read. Small transfer size descriptors will not get efficient read sizes.

One last control of read size is the alignment of the descriptor. The PEX8619 DMA will attempt to align reads on a 64B cache line boundary. If a transfer size is large enough to require multiple reads, and the start read address is not 64B aligned, then the first read may be smaller than the target read size. Similarly the last read, which reads the remainder of the descriptor transfer size, may be smaller than the target read size.

5 Programmable Number of Reads

The number of reads outstanding is limited by three things: PCIe spec, number of DMA channels enabled, and descriptor transfer size relative to read size.

The PCIe spec allows only 32 reads per function unless extended tags are enabled. The PCIe 2.1 spec requires that the extended tag field default to 0, so a first step to get more reads is to enable the extended tag capability. Note that the 0.7 draft of the PCIe 3.0 spec will allow a default value of 1. In all of the systems the PLX has tested, there appears to be no side effect to always enabling extended tags, so doing so is strongly recommended. To enable extended tags on the PEX 8619 DMA function, set the following bit in the ***DMA Device Control Register (0x70[8] = 1)***.

The PEX8619 DMA engine can support up to 64 outstanding reads total. An indirect way to control the number of reads is to control how many channels are enabled. The following table illustrates the number of reads available for different configurations. Note that 64 reads are only possible when extended tags are enabled, otherwise the maximum number of outstanding read requests is 32.

Table 5. Number of Tags Available per DMA Channel

0x1FC[1:0]	Num DMA channels enabled	Num of Tags Per Channel	
		0x70[8]=1 (Extended)	0x70[8]=0 (Normal)
1h	1	64	32
2h	2	32	16
0h	4	16	8

The number of DMA channels enabled is controlled by the ***DMA Global Control Register (0x1FC[1:0])***. The table above shows the encoded values available to program the number of DMA channels.

The value of the tag on the PCIe TLP can tell which DMA channel is doing the read, as shown in the following tables. The tag number may be seen via a protocol analyzer or a captured header. For example, when four channels are enabled and extended tags are also enabled, channel 0 will use tags 0-14 for data read and tag 15 for a descriptor read.

Table 6. Tag Numbers available to DMA Channels with Extended Tags Enabled

Number of DMA channels	CH0 Available Tags	CH1 Available Tags	CH2 Available Tags	CH3 Available Tags
1	0-62, 63	--`	--	--
2	0-30, 31	--	32-62, 63	--
4	0-14, 15	16-30, 31	32-46, 47	48-62, 63

Table 7. Tag Numbers Available to DMA Channels with Extended Tags Disabled

number of DMA channels	CH0 Available Tags	CH1 Available Tags	CH2 Available Tags	CH3 Available Tags
1	0-30, 31	--`	--	--
2	0-14, 15	--	16-30, 31	--
4	0-6, 7	8-14, 15	16-22, 23	24-30, 31

One other factor can limit the number of reads available to DMA. There can only be 4 descriptors active per channel at a time. As a result, if the transfer size for the descriptor is small, the number of outstanding reads for that one descriptor can also be small. For example, consider the case where the target read size is 128B, extended tags are enabled, and two DMA channels are enabled. In this case, one might expect to get up to 32 reads per channel from the above table. But if the transfer size is only 256B, then only 8 reads (2*128 = 256, multiplied by 4 descriptors) would be available and the resulting performance could be impacted.

6 DMA Overhead to CPU

The whole point of DMA is to offload the CPU. Rather than having the CPU move the data directly using PIO reads and potentially stalling, instead a DMA engine is used. The PEX8619 DMA can move large amounts of data in the background and inform the CPU when it is done.

Depending on the application, there is a threshold above which it becomes more efficient to use DMA instead of PIO transfers by the CPU. For the PEX8619 DMA, a single descriptor requires 16B of CPU write to set up, a 4B write to start, and potentially, it may require an interrupt to inform the CPU when the transfer is done.

Multiple descriptors can be set up for each DMA start, reducing the CPU write load. The programming to get started consists of only posted writes, which typically do not stall the CPU.

6.1 DMA Descriptors

A DMA descriptor is a set of instructions which tells the DMA engine what to do. Each 16B DMA descriptor needs to provide the source address, destination address, transfer size, and other control information such as interrupt enable and constant addressing as shown in Figure 3. Note: Figure 3 shows the Standard descriptor format. The use of Extended Descriptors will not be covered in this White Paper. Please refer to the PEX8619 Data Book for information on Extended Descriptors.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DWord 3	Lower SrcAddr [31:0]																															
DWord 2	Lower DstAddr [31:0]																															
DWord 1	SrcAddr[47:32]																DstAddr[47:32]															
DWord 0	DV	D	RC	WC	DF	Transfer Size [26:0]																										

Figure 4. DMA Descriptor Format

The control bits are in the upper bits of the lowest DWord and are defined as follows:

- Bit 31 – Descriptor Valid: must be set for the descriptor to be worked on
- Bit 30 – Descriptor Done Interrupt Enable: Interrupt when done with descriptor
- Bit 29 – Hold Read Address Constant
- Bit 28 – Hold Write Address Constant
- Bit 27 – Descriptor Format (0 = Standard; 1 = Extended).

Once the descriptors are loaded and the DMA operation is activated, the data movement phase begins. Data is read from the Descriptor source address by sending one or more memory reads at a programmable rate. The read completions will be received by the DMA engine and transformed into writes to the destination address. This continues until the transfer size is met, indicating that a particular descriptor is done.

From a transaction ordering point of view, all of the data reads are sent in strict descriptor order. Similarly, all the data writes are sent in strict descriptor order. However, it is possible that reads for a second descriptor could occur after all the reads for the first descriptor, but before the last write for the first descriptor. This is only an issue if trying to read the data written by the previous descriptor.

6.2 DMA Descriptor Modes

The PEX8619 DMA allows descriptors to be accessed in three modes.

1. Block Mode – One (single) descriptor that is written directly to dedicated DMA function registers
2. On-chip Mode – Multiple descriptors are written to on-chip Descriptor RAM
3. Off-chip Mode – Multiple descriptors are written in system Memory (Mem)

In all three modes, the descriptors should be set up before the DMA is initiated. Each descriptor will be checked for a valid bit. If the descriptor is invalid (valid bit is not set), the DMA engine will pause at that descriptor and will require the CPU to write the valid bit to a one. In this case, the CPU must also clear the ***DMA Channel N DMA Descriptor Invalid Status*** bit in the DMA ***Channel N Control/Status Register (0x238[8] for channel 0)*** to resume the DMA transfer.

DMA Block Mode is enabled by clearing the ***DMA Channel X DMA Descriptor Mode Select bit (0x238[4]=0 for channel 0)***. To enable DMA Ring Descriptor Mode, set the ***DMA Channel X DMA Descriptor Mode Select bit (0x238[4]=1 for channel 0)***. In this mode, the descriptors will be fetched from either the on-chip RAM or off-chip from system memory. The ***DMA On-Chip Descriptor Mode bit*** in the ***DMA Global Control Register (0x1FC[2])*** determines which descriptor mode is used.

- 0x1FC[2] = 0: the descriptors are read from the external descriptor pointer location (Off-Chip)
- 0x1FC[2] = 1: the descriptors are read from internal RAM (On-Chip)

6.2.1 Block Mode (Single Descriptor Mode)

The DMA block transfer moves data from a source to a destination based on the information programmed in the DMA Block Mode registers for source address, destination address, transfer count, and also DMA channel control registers. This information can be thought of as a single descriptor. The DMA block mode descriptor is programmed by a CPU before the CPU starts DMA.

When a DMA block transfer starts, the DMA engine reads data from the source and writes to the destination. Once all data has been transferred, the DMA block transaction is done. It is recommended that an interrupt be enabled in order to inform the CPU that the transfer has completed. Block mode is ideal for very large transfers.

The following outlines a simple example of the programming steps needed to setup and initiate a DMA Block Mode transfer for channel 0.

1. Program the source address where the DMA transfer reads are to start into the ***DMA Channel 0 Source Address Lower*** (200h) and ***DMA Channel 0 Source Address Upper*** (204h) registers.
2. Program the destination address where the DMA transfer Writes are to start into the ***DMA Channel 0 Destination Address Lower*** (208h) and ***DMA Channel 0 Destination Address Upper*** (20Ch) registers.
3. Program the total transfer size in bytes into bits [26:0] of the ***DMA Channel 0 Transfer Size*** register at offset 210h.
4. Program the ***DMA Channel 0 DMA Source Maximum Transfer Size*** bits (18:16]). This encoded value must not exceed the PCIe maximum payload size (MPS) because source read requests are converted into destination write requests. Also set the ***DMA Channel 0 DMA Start*** bit ([3]) in the ***DMA Channel 0 Control/Status*** register at offset 238h. Setting the Start bit will initiate the DMA engine for the channel and begin moving data.

Block mode progress could be monitored by polling the ***DMA Channel 0 In-Progress*** status bit ([30]) in the ***DMA Channel 0 Control/Statu***s register at offset 238h. More efficiently, one could set the ***DMA Channel 0 DMA Done Interrupt Enable*** bit ([30]) in the **DMA Channel 0 Transfer Size** register at offset 210h. When the Block Mode DMA completes, an Assert_INTB Message Request will be sent to the CPU. The service routine would confirm the DMA completion by reading the ***DMA Channel 0 DMA Descriptor Done Interrupt Status*** bit ([18]) in the ***DMA Channel 0 Interrupt Control/Status*** register at offset 23Ch. The action of writing a one to clear that bit will result in a Deassert_INTB Message Request to be transmitted to the CPU.

6.2.2 On-Chip Descriptor Mode

Up to four descriptors can be active at a time, per channel. The CSRs controlling the descriptor (used in block mode) do not need to be programmed, as these values will be taken from the internal descriptor RAM.

The on-chip descriptor RAM holds up to 256 descriptors, each 16B. Table 8 below lists the number of descriptors available for the number of DMA channels enabled.

Table 8. Number of On-Chip Descriptors Available per DMA Channel

Number of DMA channels	Maximum Available Descriptors Per Channel	Channel 0 RAM Offset range for descriptors	Channel 1 RAM Offset range for descriptors	Channel 2 RAM Offset range for descriptors	Channel 3 RAM Offset range for descriptors
1	256	000h – FFFh	n/a	n/a	n/a
2	128	000h – 7FFh	n/a	800h – FFFh	n/a
4	64	000h – 3FFh	400h – 7FFh	800h – BFFh	C00h – FFFh

The descriptor RAM is in the CSR memory space at offset 0x1000 from the BAR0 of the DMA function.

The advantage to using internal descriptors is to avoid the descriptor fetch on the PCIe link, which may cost some performance overhead. The disadvantage is that it is a limited size.

On-chip descriptors can operate in a ring mode, where the last descriptor will point back to the first descriptor and the DMA will continue in an infinite loop.

The following outlines a simple example of the programming steps needed to setup and initiate one DMA operation for channel 0 using an on chip descriptor list.

1. Enable the on chip descriptor mode by setting the DMA ***On-Chip Descriptor Mode*** bit ([2]) in the ***DMA Global Control*** register at offset 1FCh from the Upstream port DMA Function base address. In the same register, bit [0] also needs to be set so that the ***DMA Channel Mode*** field (bits [1:0]) is encoded to enable only a single DMA Channel (0).
2. Write from one to 256 descriptor entries into the on-chip DMA descriptor RAM. Each descriptor uses 4 dwords of RAM space. On-chip RAM locations available for a single DMA Channel 0 mode range from offsets 1000h to 1FFFh. These offsets are relative to the memory-mapped address programmed into the BAR0 register (32-bit address is default) of the Upstream DMA Function. Make sure that the ***Descriptor Valid*** bit ([31]) is set in the first dword of each descriptor. Also set the ***Interrupt When Done With Descriptor*** bit ([30]) in the first dword of the last descriptor of the list.

3. Program the 32-bit starting address of the first descriptor in the on-chip DMA descriptor RAM into the ***DMA Channel 0 Descriptor Ring Address Lower*** register at offset 214h. For this example, the value you would place into this register would be the base address programmed by the system into the ***Base Address 0 DMA*** (BAR0) register of the Upstream DMA Function plus 1000h. You do not need to program the ***DMA Channel 0 Descriptor Ring Address Upper*** register at offset 218 because 32-bit descriptor addressing is the power-up default.
4. Program the same write data in step 3 above into the ***DMA Channel 0 Next Descriptor Ring Address Lower*** register at offset 21Ch. It is required to make this register and the ***DMA Channel 0 Descriptor Ring Address Lower*** register hold the same value before the initial startup of a DMA operation when using on-chip, or off-chip descriptor lists.
5. Program a value equal to the total number of descriptors into the DMA ***Channel 0 Descriptor Ring Size*** register at offset 220h.
6. Program the ***DMA Channel 0 Control/Status*** register at offset 238h. You will need to set the ***DMA Channel 0 DMA Descriptor Mode Select*** bit ([4]) to enable DMA Descriptor Ring Mode. For the simplest case, set the ***DMA Channel 0 DMA Stop Mode*** bit ([5]) to stop the DMA operation when the complete descriptor list is exhausted. Program the ***DMA Channel 0 DMA Source Maximum Transfer Size*** bits ([18:16]) to your desired value. This encoded value must not exceed the PCIe maximum payload size (MPS) because source read requests are converted into destination write requests. You must also set the ***DMA Channel 0 DMA Start*** bit ([3]) to initiate the DMA engine for the channel to begin moving data.

In this example, an Assert_INTB Message Request will be transmitted when the last descriptor has been processed. The interrupt service routine would read the ***DMA Channel 0 Interrupt Control/Status*** register and check the state of the ***DMA Channel 0 DMA Descriptor Done Interrupt Status*** bit ([18]) to validate the interrupt. The action of writing a one to clear that bit will result in a Deassert_INTB Message Request to be transmitted to the CPU.

On-chip descriptor mode may be very suitable for looping on the same descriptor ring continuously without modification. In that case you would not set bit [5] in the ***DMA Channel 0 Control/Status*** register at offset 238h. Neither would the ***Interrupt When Done With Descriptor*** bit [30] be set in the first dword of the last descriptor of the list.

It may be desirable to re-program descriptors after a pass through the list. Once the interrupt has been serviced, and the descriptors are reprogrammed, setting the ***DMA Channel 0 DMA Start*** bit ([3]) in the ***Channel 0 Control/Status*** register at offset 238h will get the DMA operation going again. Note that the ***DMA Channel 0 DMA Start*** bit is self clearing, and does not need to be toggled back to a zero state first. Simply the act of writing the bit to a one is sufficient to begin the DMA data transfer.

6.2.3 Off-Chip Descriptor Mode

In Off-Chip mode, there is virtually unlimited space to store descriptors. A pointer to the descriptors as well as the size of the descriptor space must be programmed. The starting descriptor is programmed in the Next

Descriptor register for the channel. The DMA can branch to other descriptors by changing the Next Descriptor.

Multiple descriptors could be sequential in a ring. The CPU needs to program the ***DMA Channel X Descriptor Ring Address registers (0x214h – 0x218h for channel 0)*** along with the ***DMA Channel X Control/Status register (0x238h for channel 0)*** for some control information. The CPU then sets the start bit and DMA engine starts working on the descriptors.

For best performance, the PEX 8619 DMA engine will prefetch four sets of request descriptors at a time. The descriptor read will be 64B at a time, corresponding to four 16B descriptors. The descriptor starting address is always 64B aligned. The descriptor read has its own single dedicated tag for each channel, no tag reservation is needed. The descriptor read continues forever until an invalid descriptor is detected or the DMA reaches the end of the ring and ring looping is not enabled or the DMA is aborted. Up to four descriptors can be active at a time, per channel.

The following outlines a simple example of the programming steps needed to setup and initiate one DMA operation for channel 0 using an off chip descriptor list.

1. Program bits [1:0] in the ***DMA Global Control*** register at offset 1FCh from the Upstream port DMA Function base address. This insures that the ***DMA Channel Mode*** field (bits [1:0]) is encoded to enable only a single DMA Channel (0). Make sure that the ***DMA On-Chip Descriptor Mode*** bit ([2]) in the same register is cleared to enable the off chip descriptor mode.
2. Program the off-chip descriptor list into host memory. Make sure that the ***Descriptor Valid*** bit ([31]) is set in the first dword of each descriptor, and for this example set the ***Interrupt When Done With Descriptor*** bit ([30]) in the first dword of the last descriptor of the list.
3. Program the 32-bit starting address of the first descriptor in the off chip descriptor list into the ***DMA Channel 0 Descriptor Ring Address Lower*** register at offset 214h. This example assumes that the descriptor list is referenced in 32-bit address space, so there is no need to program the ***Channel 0 Descriptor Ring Address Upper*** register at offset 218h. The DMA engine will issue 64-bit requests if the Descriptor Ring Address Upper register is non-zero.
4. Program the same write data in step 3 above into the DMA ***Channel 0 Next Descriptor Ring Address Lower*** register at offset 21Ch. It is required to make this register and the ***DMA Channel 0 Descriptor Ring Address Lower*** register hold the same value before the initial startup of a DMA operation when using off-chip descriptor lists.
5. Program a value equal to the total number of descriptors into the ***DMA Channel 0 Descriptor Ring Size*** register at offset 220h.
6. Program the desired maximum number of descriptors to be pre-fetched into the ***DMA Channel Maximum Prefetch Limit*** register at offset 234h, bits [7:0]. The legal values that can be programmed are either a multiple of 4, or the value of 1 (only one descriptor is prefetched). If the actual number of valid descriptors in the last prefetch group is not a whole multiple of 4, then the DMA engine will simply discard the first and subsequent invalid descriptors of that group.
7. Program the ***DMA Channel 0 Control/Status*** register at offset 238h. You will need to set the ***DMA Channel 0 DMA Descriptor Mode Select*** bit ([4]) to enable ***DMA Descriptor Ring Mode***. For the simplest case, set the ***DMA Channel 0 DMA Stop Mode*** bit ([5]) to stop the DMA operation when the

complete descriptor list is exhausted. Program the ***DMA Channel 0 DMA Source Maximum Transfer Size*** bits ([18:16]) to your desired value. This encoded value must not exceed the PCIe maximum payload size (MPS) because source read requests are converted into destination write requests. You must also set the ***DMA Channel 0 DMA Start*** bit ([3]) to initiate the DMA engine for the channel to begin moving data.

In this example, an Assert_INTB Message Request will be transmitted to the CPU when the last off chip descriptor has been processed. The CPU would read the ***DMA Channel 0 Interrupt Control/Status*** register and check the state of the ***DMA Channel 0 DMA Descriptor Done Interrupt Status*** bit ([18]) to validate the interrupt. The action of writing a one to clear that bit will result in a Deassert_INTB Message Request to be transmitted to the CPU.

It is possible to monitor descriptor progress for the current DMA operation by examining the ***DMA Channel 0 Last Descriptor Address Lower*** register at offset 224h and the DMA ***Channel 0 Last Descriptor Transfer Size*** register at offset 228h.

It may be desirable to re-program descriptors after a pass through the list. Once the interrupt has been serviced, and the descriptors are reprogrammed, setting the ***DMA Channel 0 DMA Start bit*** ([3]) in the ***Channel 0 Control/Status*** register at offset 238h will get the DMA operation going again. Note that the ***DMA Channel 0 DMA Start*** bit is self clearing, and does not need to be toggled back to a zero state first. Simply the act of writing the bit to a one is sufficient to begin the DMA data transfer.

Beyond this simple example, there are other user options for updating the descriptor list, either at the end of a pass through the ring, or concurrently with the active DMA operation. These options, such as the ***DMA Descriptor Writeback*** feature (clears the Valid bit of each descriptor when it is retired), are described later in this paper, as well as the device Data Book.

6.3 DMA Interrupts

The PEX 8619 DMA interrupt format can be MSI or legacy INTx. MSI interrupts obviously have performance gains, in that the CPU may not have to read the interrupt status on the device to determine what interrupt service routine to start.

MSI interrupts are enabled via the PCIe standard capability registers starting at offset ***0x48*** in the DMA function configuration space.   The PEX 8619 supports Eight MSI vectors (two per channel): one error and one normal interrupt. In the table below, the only error interrupt is bit[0]; all the other cases are considered normal interrupts.

It is recommended that DMA interrupts should be setup to use the default INTx type of message signaling. That is because there are rare instances when an MSI may not convey the true cause of the DMA interrupt. An error message could get folded into another MSI vector, thus masking the true primary cause of the interrupt. For example, the DMA engine may generate a memory read request that is rejected by the internal request routing logic because the source port has experienced a link down condition. The DMA controller would enter Abort, and no error interrupt would be generated. Only the DMA Abort Done would be associated with the vector of the MSI received by the CPU. With INTx message signaling, the CPU can read device status registers (such as the ***DMA Channel N Device Specific Error Status*** register at channel offset 250h). This would allow software to discover the error condition that caused the DMA engine to abort the operation. The DMA Function 1 Endpoint uses INTB Interrupt Signaling Message coding as a hardware default.

Table 9. MSI Interrupt Vectors

0x23Ch	DMA CH0 Interrupt Control/Status
[0]	DMA Error Interrupt Enable
[1]	DMA Invalid Descriptor Interrupt Enable
[2]	Reserved
[3]	DMA Abort Done Interrupt Enable
[4]	DMA Graceful Pause Done Interrupt Enable
[5]	DMA Immediate Pause Done Interrupt Enable
[15:6]	Reserved
[16]	DMA Error Interrupt Status
[17]	DMA Invalid Descriptor Interrupt Status
[18]	DMA Descriptor Done Interrupt Status
[19]	DMA Abort Done Interrupt Status
[20]	DMA Graceful Pause Done Interrupt Status
[21]	DMA Immediate Pause Done Interrupt Status
[31:22]	Reserved

Table 10. DMA Error Status Bits

0x250h	DMA CH0 Device Specific Error Status
[0]	Data Read Error Status
[1]	Data Write Error Status
[2]	Descriptor Read Error Status
[3]	Descriptor Write Error Status
[31:4]	*Reserved*

The enable bit for the ***DMA Descriptor Done Interrupt*** Status bit ([18]) is found in DWord 0 of each descriptor (bit [30]). Any or all descriptors could generate an interrupt. Since the interrupt enable bit is placed on the descriptor, this allows software to determine the appropriate interrupt rate. Rather than interrupt on every descriptor, it may be beneficial to interrupt once per 16 descriptors for example, in order to allow the CPU to work on other things. The impact to a system which can result from interrupts are system and application dependent and are outside the scope of this document.

Additionally, the status of the descriptors can be written to system memory, which the CPU can access faster than a register on the PEX 8619. By enabling '***Writeback*** mode' on the descriptor, when the last data of the descriptor has been exhausted, the PEX8619 DMA engine will overwrite DWord 0 of the off chip descriptor to update the ***Valid*** bit ([31]), and potentially the ***Transfer Size*** field ([26:0] of a Standard Descriptor format entry). The descriptor write back clears the valid bit on the descriptor once that descriptor has completed successfully or was aborted before full completion. This allows the CPU to monitor the descriptor ring directly and then update an entry once it has been completely processed by the DMA engine.

To enable descriptor write back, set the ***DMA Channel X DMA Completion Status Write Back Enable bit (0x238[2]=1 for channel 0)***. Note that an erratum currently exists for the ***Writeback*** feature. The ***Transfer Size*** field in DWord 0 may not reflect the true partial progress of a descriptor prior to an Abort condition. This erratum is only applicable for 32-bit descriptor addressing. The ***Transfer Size*** write back erratum does not apply to 64-bit descriptor addressing.

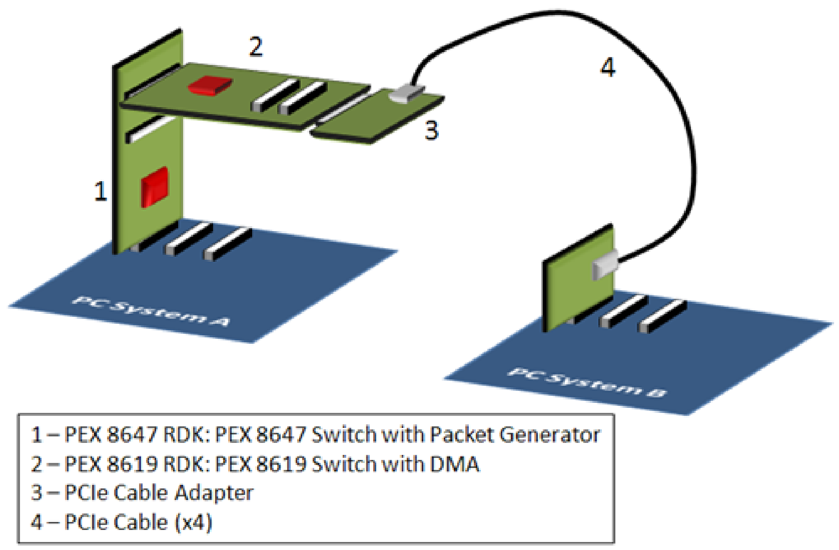
7 Testing

A Rapid Development Kit (RDK) is a piece of hardware which is available from PLX and serves as a test and/or evaluation platform for the PLX devices. Moreover, PLX also provides a software suite (SDK), available for download at www.plxtech.com/products/sdk. The SDK includes device drivers for the DMA function and the packet generator function which are used in the testing described in this document. Additionally, the SDK also provides a nice interface for displaying the traffic through the switch by reading the data collected by the PCIe switch counters and displaying it in a graphical format for easier dissection.

For the following tests, two PLX RDKs were used. One is the 8619 RDK, which has the PLX 8619 PCIe switch with DMA. Another is the 8647 RDK, which has the PEX 8647 with the packet generator and which proved to be useful in many ways.

There were two PC systems tested. Both systems consisted of an MSI motherboard, AMD Phenom 9500 Quad-Core socket AM2+ Processor, NVIDIA nForce 780a SLI SPP rev. A2 chip set, 4096 Mbytes DDR2 400 MHz RAM (in two slots of memory).

The figure below illustrates the setup. The 8647 RDK plugged in to the motherboard (to bus number 5) and trained with a x16 link with 5.0 GT/s SerDes. The 8619 RDK was plugged in to port4 (P4) of the PEX 8647 (bus number 7) and trained with a x8 link with 5.0 GT/s SerDes. Furthermore, the NT function on the PEX 8619 was enabled on port1 (P1). System B was connected to System A via the NT port on the PEX 8619 using the standard PCIe cable. The cable to System B trained as a x4 PCIe link with 5.0 GT/s SerDes.


1
2
3
4
PC System A
PC System B
1 – PEX 8647 RDK: PEX 8647 Switch with Packet Generator
2 – PEX 8619 RDK: PEX 8619 Switch with DMA
3 – PCIe Cable Adapter
4 – PCIe Cable (x4)

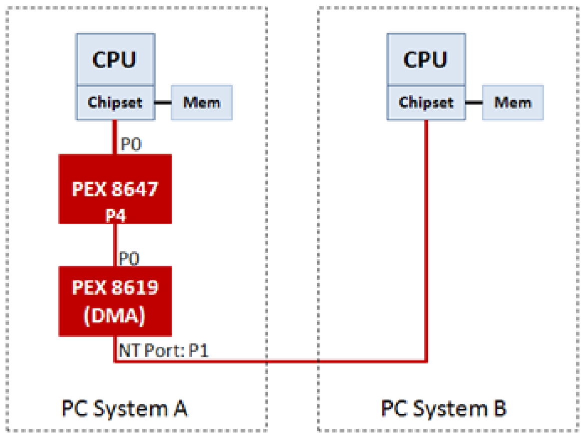


Figure 5. DMA Test System using PEX 8647 and PEX 8619

The first series of tests are for system calibration and used the setup described in Section 7.1. The last series of tests added System B. In the testing and results discussion below, references to gen2 links imply PCIe links which implement 5.0 GT/s SerDes.

7.1 System Calibration

Before running performance tests, it is often useful to calibrate the system. The PEX 8647 with its packet generator is perfect for this task. The PLX driver that loads when a PLX switch is found, and the SDK software is installed, will automatically create a memory buffer in system RAM that can be used as a target for the packet generator TLPs.

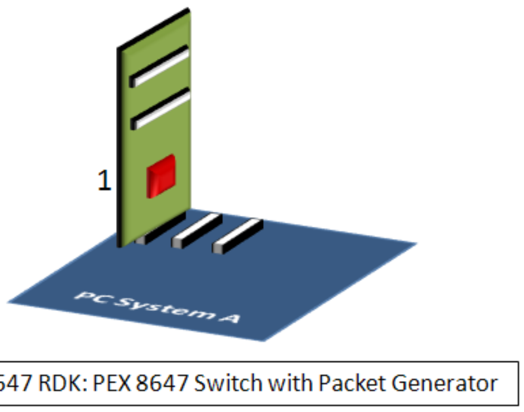


Figure 6. System Calibration Setup using the PEX 8647 RDK

For this system (System A), it was found that the first x16 slot chosen is only electrically x8. After moving to another slot, it was found to be electrically x16 but only got the performance of an x8.

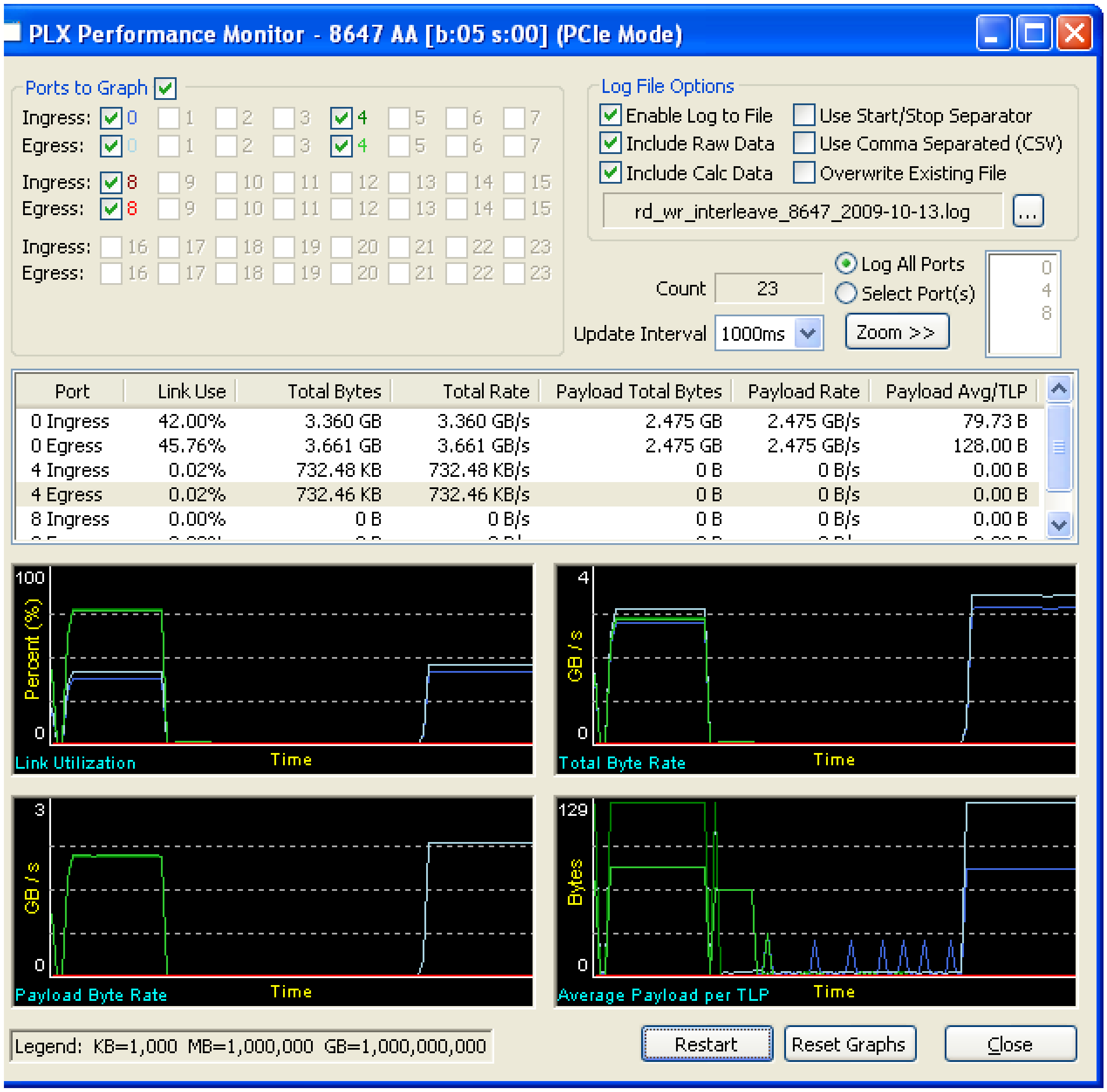
Three simple tests were run: 100% reads (burst 8 sequential reads in infinite loop), 100% writes (burst 8 sequential writes in infinite loop), and 50/50 reads and writes (burst read/write interleaving, 8 each, with reads sequential and writes sequential, in infinite loop). The results are summarized in the table below.

Table 11. Calibration Test Results

Packet Generator x16 gen2	average payload ingress (bytes)	payload throughput ingress (GB/s)	average payload egress (Bytes)	payload throughput egress (GB/s)
read only	127.96	3.696	0	0
write only	0	0	128.00	3.444
read/write	79.73	2.475	128.00	2.475

The payload throughput is shown, as that is the work that actually gets done. The average payload size is also shown, because with that the theoretical throughput can be calculated and compared to the measured throughput.

The data in Table 10 was extracted from the SDK GenMon. An example screenshot of one of the results is shown in the figure below.



PLX Performance Monitor - 8647 AA [b:05 s:00] (PCIe Mode)

Ports to Graph ✓

Ingress:	✓ 0	1	2	3	✓ 4	5	6	7
Egress:	✓ 0	1	2	3	✓ 4	5	6	7
Ingress:	✓ 8	9	10	11	12	13	14	15
Egress:	✓ 8	9	10	11	12	13	14	15
Ingress:	16	17	18	19	20	21	22	23
Egress:	16	17	18	19	20	21	22	23

Log File Options

- ✓ Enable Log to File
- ✓ Include Raw Data
- ✓ Include Calc Data
- Use Start/Stop Separator
- Use Comma Separated (CSV)
- Overwrite Existing File

rd_wr_interleave_8647_2009-10-13.log

[✓ Log All Ports] Select Port(s)

Count 23

Update Interval 1000ms

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	42.00%	3.360 GB	3.360 GB/s	2.475 GB	2.475 GB/s	79.73 B
0 Egress	45.76%	3.661 GB	3.661 GB/s	2.475 GB	2.475 GB/s	128.00 B
4 Ingress	0.02%	732.48 KB	732.48 KB/s	0 B	0 B/s	0.00 B
4 Egress	0.02%	732.46 KB	732.46 KB/s	0 B	0 B/s	0.00 B
8 Ingress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 7. Read/Write System Calibration Results

The GenMon tool was set to monitor the PEX 8647. From the PEX 8647 point of view, port0 is upstream, so port0 ingress is data coming from memory (i.e. read completions) and port0 egress is going to memory (i.e. reads and writes). This test shows heavy traffic in both directions, so it must be (and is) for the read/write 50/50 test. The payload rate and payload avg/TLP match the numbers recorded in Table 10 above. A read and write size of 128B was chosen, which is why the 128B is the egress port0 payload size. PCIe systems all must support a minimum of 128B for its max payload size (MPS), and while a few support larger than 128B, for this system only the default payload size was tested. ***Note: A system that supports larger MPS can achieve higher payload throughput using the otherwise same PCIe link. The device status and control register, 0x70, bits [7:5] reflect the maximum payload size set by the system; PLX devices support up to 2048B payload.***

Also recorded in Figure 6 is the link utilization which can provide some insight into the system behavior. The port0 ingress only got to 42% of the link bandwidth, while the egress got to 45.76%. This data shows two things about the system.

One, the system memory controller is not up to gen2 x16 throughput, getting less than half of possible, which is less than gen2 x8 rates (which would be 50% of gen2 x16 link). Clearly in this case, the system is the bottleneck and not the PEX 8647, which can saturate gen2 x16.

Two, the upstream path (port0 egress) is getting more load than downstream. This makes sense if you consider the average payload and the traffic. Writes (upstream) all have a 128B payload, but the upstream link is also used by reads, at a rate of 1 read per 1 write, so the 'real' average is 64B payload per TLP (reads have 0B payload). Read completions (downstream) average 79.73B, slightly more efficient than the 64B upstream. As a result, upstream traffic (port0 egress) will be the bottleneck in total throughput if the system bottleneck is removed.

The read completion average result is interesting: the completions are either 128B or 64B, as that is the only choice that the root complex is allowed per PCIe spec. So a value of 79.73B indicates that some read completions are 128B (25%) and some are 64B (75%). This is an interesting result, and so the read completion size is something that we should track for DMA tests.

It has been found that when a bus is close to 100% saturation, sometimes the system might hang if there are any CPU issued PIO reads to the saturated bus. The MRd downstream might be stuck behind the DMA read completions, for example. In this calibration test, the monitor feature of the SDK is doing periodic reads to the PEX 8647 – and if those reads time out, the system may hang. An I^2C connector with its sideband connection would get around this issue. For this system, it turned out there was no issue either since the memory controller could not saturate the PCIe link. Note however that when the x8 link in the PEX 8619 is connected behind the PEX 8647, the x8 link could become saturated. To avoid any system read timeouts and not having an I^2C connection available, the tests will read the monitor registers on the PEX 8647, not the PEX 8619.

7.2 Test 1a: DMA: Read RAM, Write RAM

The easiest test to set up is to read main memory and write main memory, as that only requires a PC and a PEX 8619 RDK. As mentioned previously, a PEX 8647 was used in this test as well. The PEX 8619 was plugged into port4 of the PEX 8647, and the PEX 8647 had port0 plugged into the mother board. The PEX 8647 was used to monitor the DMA bandwidth between the PEX 8619 and System A.

Before testing, it is helpful to predict what kind of performance is possible. From the system calibration with the PEX 8647, the best that can be achieved is 2.47 GB/s when reading and writing system RAM. If a test can be done without involving software, i.e. avoid interrupt handling and its random latency adder, then we will see what the hardware can do. The expectation is to achieve close to 2.47 GB/s. DMA has some ‘extra’ traffic in the descriptor fetch, but with an 8KB transfer size per descriptor, the 16B of descriptor overhead is minimal (less than 0.2 percent).

Similar to the packet generator, addresses to read and write and addresses to store the descriptors all need to be programmed. This test used an internally developed Python driver to make it easy to program registers (this driver may be available for win32 machines), and it creates a system buffer which will be labeled “pc_mem”. The 1 MB buffer given by Python needs to be split to store descriptors, read data, and write data.

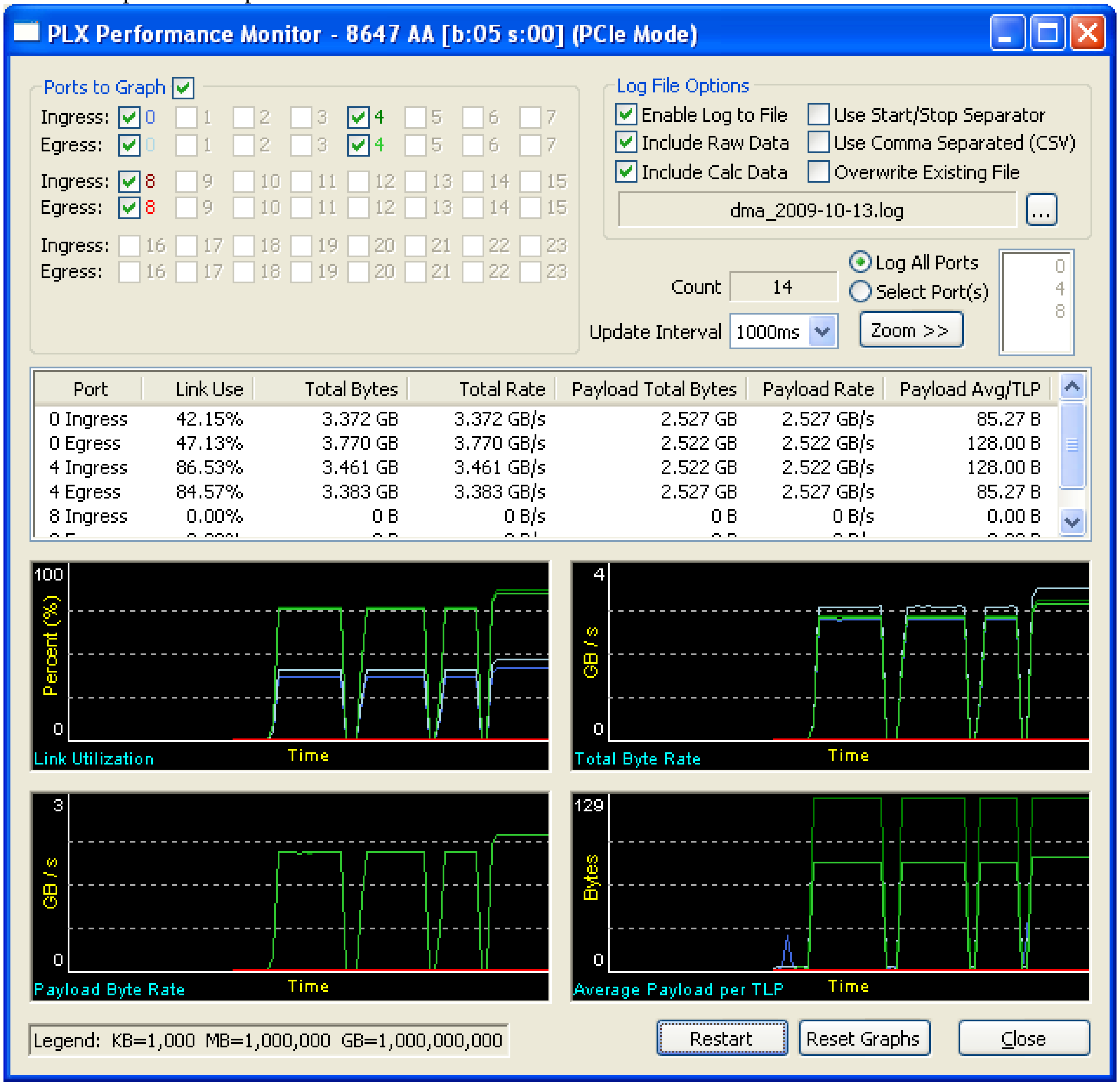
For the first test using one DMA channel, the following spaces are defined:

- pc_mem to (pc_mem + 0x1_ffff): channel 0 read data (128KB)
- pc_mem+0x2_0000 to (+128KB): channel 0 write data
- pc_mem+0xf_f000 to +0xf_ffff: channel 0 descriptors

Off chip descriptors is a common use model, so this configuration was tested. To minimize the impact of descriptors on data throughput, each descriptor has an 8KB transfer size, where 16 descriptors will be run in a continuous loop. To eventually test 2 channels, the 1 MB space was divided into 8 128k spaces. The descriptors are placed in the top 258K of the memory buffer.

The DMA has many programming options, but this test turns up all the performance knobs to show what can be achieved. The source code for the python based test is in the Initialization section at the end of this document. All of the necessary DMA registers are shown with the values and the order used for these tests.

Testing 8KB transfer per descriptor, read Mem, write Mem, extended tags, one channel, 16 descriptors, infinite loop on descriptors



PLX Performance Monitor - 8647 AA [b:05 s:00] (PCIe Mode)

Ports to Graph ✓

Ingress:	✓ 0	1	2	3	✓ 4	5	6	7
Egress:	✓ 0	1	2	3	✓ 4	5	6	7
Ingress:	✓ 8	9	10	11	12	13	14	15
Egress:	✓ 8	9	10	11	12	13	14	15
Ingress:	16	17	18	19	20	21	22	23
Egress:	16	17	18	19	20	21	22	23

Log File Options

✓ Enable Log to File — Use Start/Stop Separator
✓ Include Raw Data — Use Comma Separated (CSV)
✓ Include Calc Data — Overwrite Existing File

dma_2009-10-13.log ...

Count 14

[● Log All Ports] Select Port(s)

0
4
8

Update Interval 1000ms

Zoom >>

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	42.15%	3.372 GB	3.372 GB/s	2.527 GB	2.527 GB/s	85.27 B
0 Egress	47.13%	3.770 GB	3.770 GB/s	2.522 GB	2.522 GB/s	128.00 B
4 Ingress	86.53%	3.461 GB	3.461 GB/s	2.522 GB	2.522 GB/s	128.00 B
4 Egress	84.57%	3.383 GB	3.383 GB/s	2.527 GB	2.527 GB/s	85.27 B
8 Ingress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

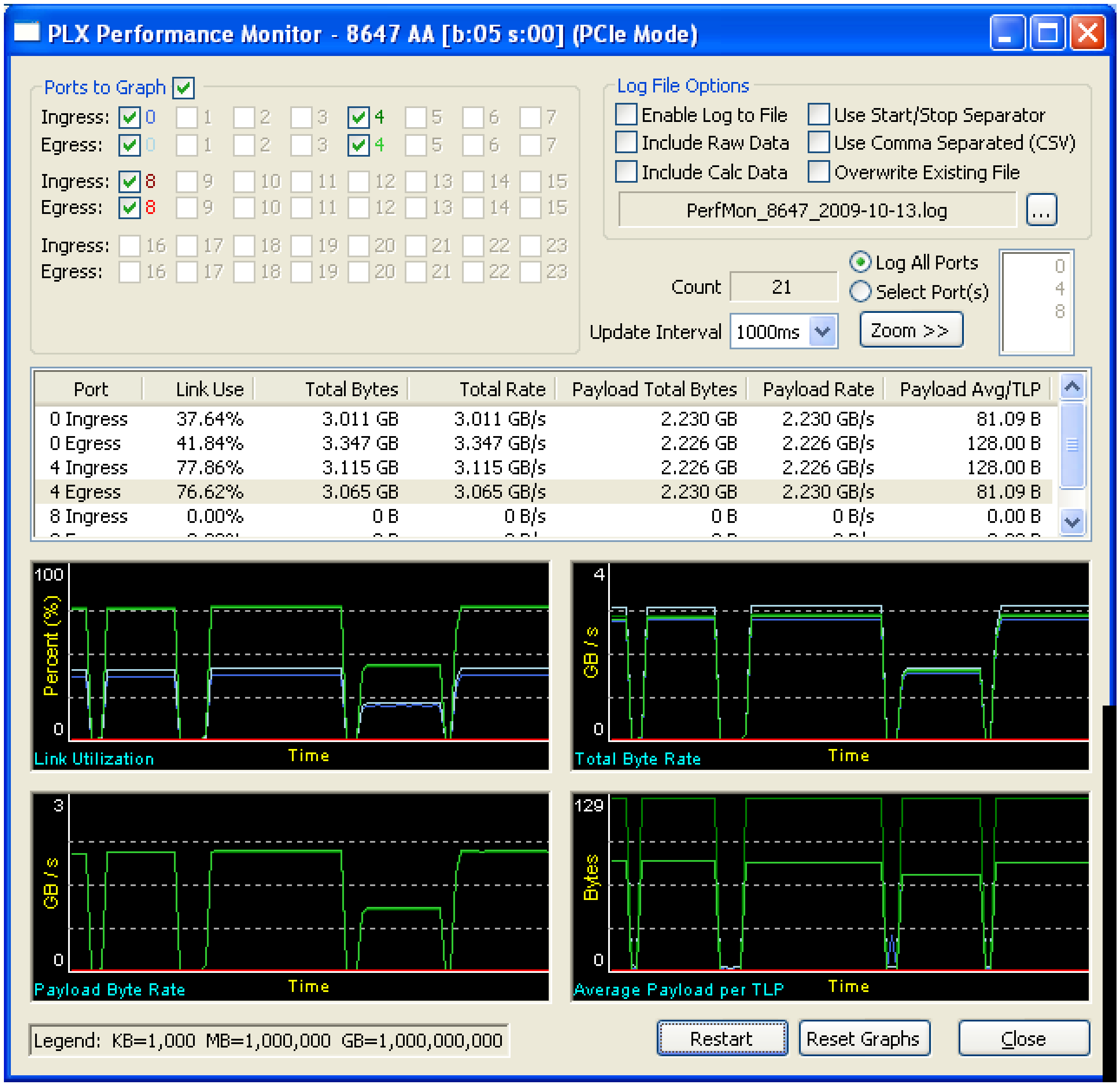
Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Restart — Reset Graphs — Close

Figure 8. 8KB DMA Transfer Using One Channel and Extended Tags Enabled

The DMA test did very well, reaching 2.52 GB/s. Note that the throughput was actually a bit “better” than the calibrated case (2.475 GB/s). This can be explained by the larger average observed payload size (85.3B vs 79.7B). It is not clear why the system gave the DMA load slightly better payload response compared to the Packet Generator test.

Testing without extended tags



PLX Performance Monitor - 8647 AA [b:05 s:00] (PCIe Mode)

Ports to Graph ✓

Ingress: ✓ 0, 1, 2, 3, ✓ 4, 5, 6, 7
Egress: ✓ 0, 1, 2, 3, ✓ 4, 5, 6, 7
Ingress: ✓ 8, 9, 10, 11, 12, 13, 14, 15
Egress: ✓ 8, 9, 10, 11, 12, 13, 14, 15
Ingress: 16, 17, 18, 19, 20, 21, 22, 23
Egress: 16, 17, 18, 19, 20, 21, 22, 23

Log File Options

Enable Log to File; Include Raw Data; Include Calc Data; Use Start/Stop Separator; Use Comma Separated (CSV); Overwrite Existing File

PerfMon_8647_2009-10-13.log

Count 21

Update Interval 1000ms

[✓ Log All Ports] Select Port(s)

Zoom >>

0 4 8

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	37.64%	3.011 GB	3.011 GB/s	2.230 GB	2.230 GB/s	81.09 B
0 Egress	41.84%	3.347 GB	3.347 GB/s	2.226 GB	2.226 GB/s	128.00 B
4 Ingress	77.86%	3.115 GB	3.115 GB/s	2.226 GB	2.226 GB/s	128.00 B
4 Egress	76.62%	3.065 GB	3.065 GB/s	2.230 GB	2.230 GB/s	81.09 B
8 Ingress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

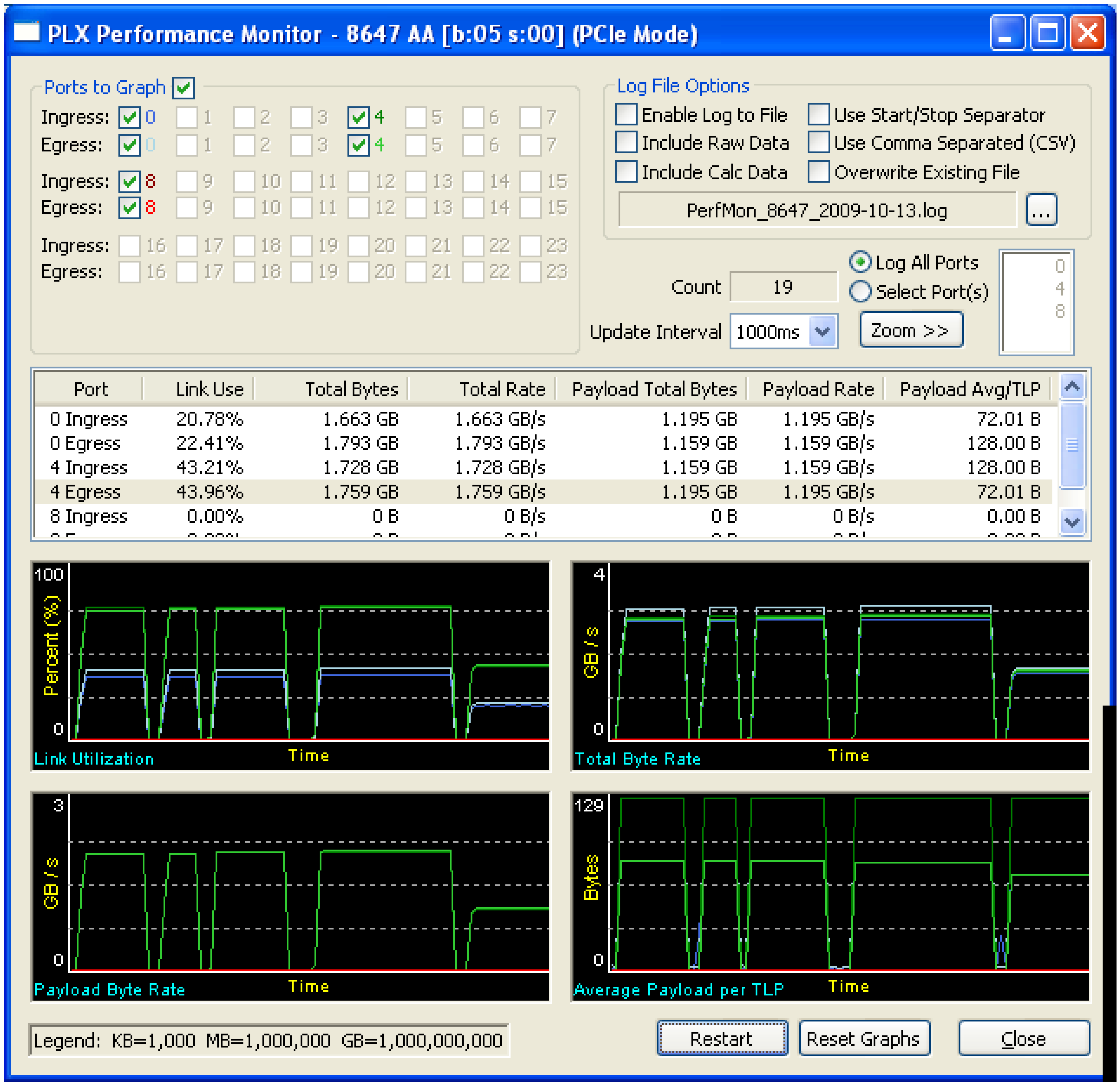
Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Restart | Reset Graphs | Close

Figure 9. 8KB DMA Transfer Using One Channel and Extended Tags Disabled

The performance dropped to 88% of the earlier case with extended tags, simply by reducing the number of 128B reads from 63 to 31. This indicates that the read round trip time may vary, but that it can be over the time it takes to send 31x 128B completions on the wire. That time, for gen2 x4, is about 31 * (20 + 128) / 2 = 2294ns.

Test reducing the transfer size to 512B (from 8192B). (extended tags enabled)



Ports to Graph [✓]

Ingress: [✓ 0] 1 2 3 [✓ 4] 5 6 7
Egress: [✓ 0] 1 2 3 [✓ 4] 5 6 7
Ingress: [✓ 8] 9 10 11 12 13 14 15
Egress: [✓ 8] 9 10 11 12 13 14 15
Ingress: 16 17 18 19 20 21 22 23
Egress: 16 17 18 19 20 21 22 23

Log File Options

Enable Log to File | Use Start/Stop Separator
Include Raw Data | Use Comma Separated (CSV)
Include Calc Data | Overwrite Existing File

PerfMon_8647_2009-10-13.log

Count 19

Update Interval 1000ms

[✓ Log All Ports] Select Port(s)

0
4
8

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	20.78%	1.663 GB	1.663 GB/s	1.195 GB	1.195 GB/s	72.01 B
0 Egress	22.41%	1.793 GB	1.793 GB/s	1.159 GB	1.159 GB/s	128.00 B
4 Ingress	43.21%	1.728 GB	1.728 GB/s	1.159 GB	1.159 GB/s	128.00 B
4 Egress	43.96%	1.759 GB	1.759 GB/s	1.195 GB	1.195 GB/s	72.01 B
8 Ingress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 10. 512B DMA Transfer Using One Channel and Extended Tags Enabled

The smaller transfer size runs into a limitation in the hardware. There are only 4 active descriptors total. With 512B transfer and 128B read, that means only 4 reads per descriptor. The total number of reads is limited to 16 in this case, even though there are 63 tags available.

7.3 Test 1b: "Improved" root complex: Read Mem, Write to PEX 8647 port

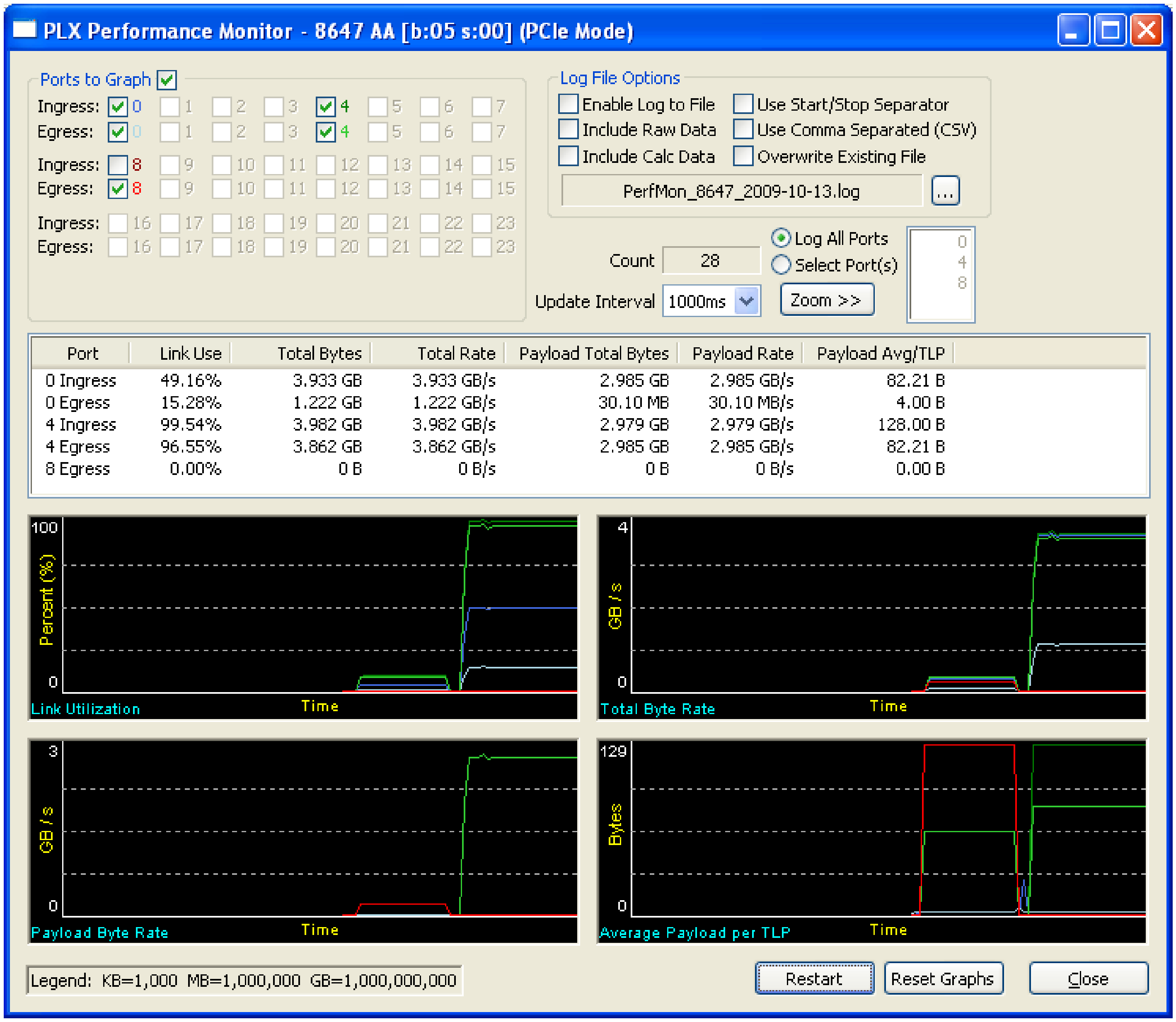
To get around the known memory bottleneck of the system, the PEX 8619 DMA was programmed to read main memory but dump the write data to a port that was disabled on the PEX 8647 (port 8). The port had the memory enable turned off but was programmed with a base and limit. The DMA was programmed to hit within the port's base and limit. The effect is that any writes sent to port8 will be discarded.

The best that gen2 x8 can do will be the ideal throughput for 1 MRd(128B) + 1 MWr(128B) going upstream. These 128B of payload have (20+20)B of overhead on the wire, if you include framing, data link layer, and TLP header. The 8b10b derating, with a payload derating, gives an ideal target of:

- ideal gen2 x8 = 5.00 * 0.8 * 128 / (128+40) = 3.05 GB/s

This ideal does not include DLLP overhead or skips, nor does it take into account the occasional descriptor fetch (1x per 4 descriptors), which might drop the number by about 3% to 2.96 GB/s.

The results closely approach the ideal. Note in the picture below that the port4 gen2 x8 link has over 99% utilization, and it is achieving 2.979 GB/s. Note also that, unlike previous screen shots, the egress throughput on port0 drops to almost 0 (30 MB/s). This indicates that the DMA writes are going somewhere else – in this case, the PEX 8647 is 'swallowing' them.



Ports to Graph ✓

Ingress:	✓ 0	1	2	3	✓ 4	5	6	7
Egress:	✓ 0	1	2	3	✓ 4	5	6	7
Ingress:	8	9	10	11	12	13	14	15
Egress:	✓ 8	9	10	11	12	13	14	15
Ingress:	16	17	18	19	20	21	22	23
Egress:	16	17	18	19	20	21	22	23

Log File Options
- ☐ Enable Log to File
- ☐ Use Start/Stop Separator
- ☐ Include Raw Data
- ☐ Use Comma Separated (CSV)
- ☐ Include Calc Data
- ☐ Overwrite Existing File

PerfMon_8647_2009-10-13.log

Count 28

[✓ Log All Ports] Select Port(s)

0
4
8

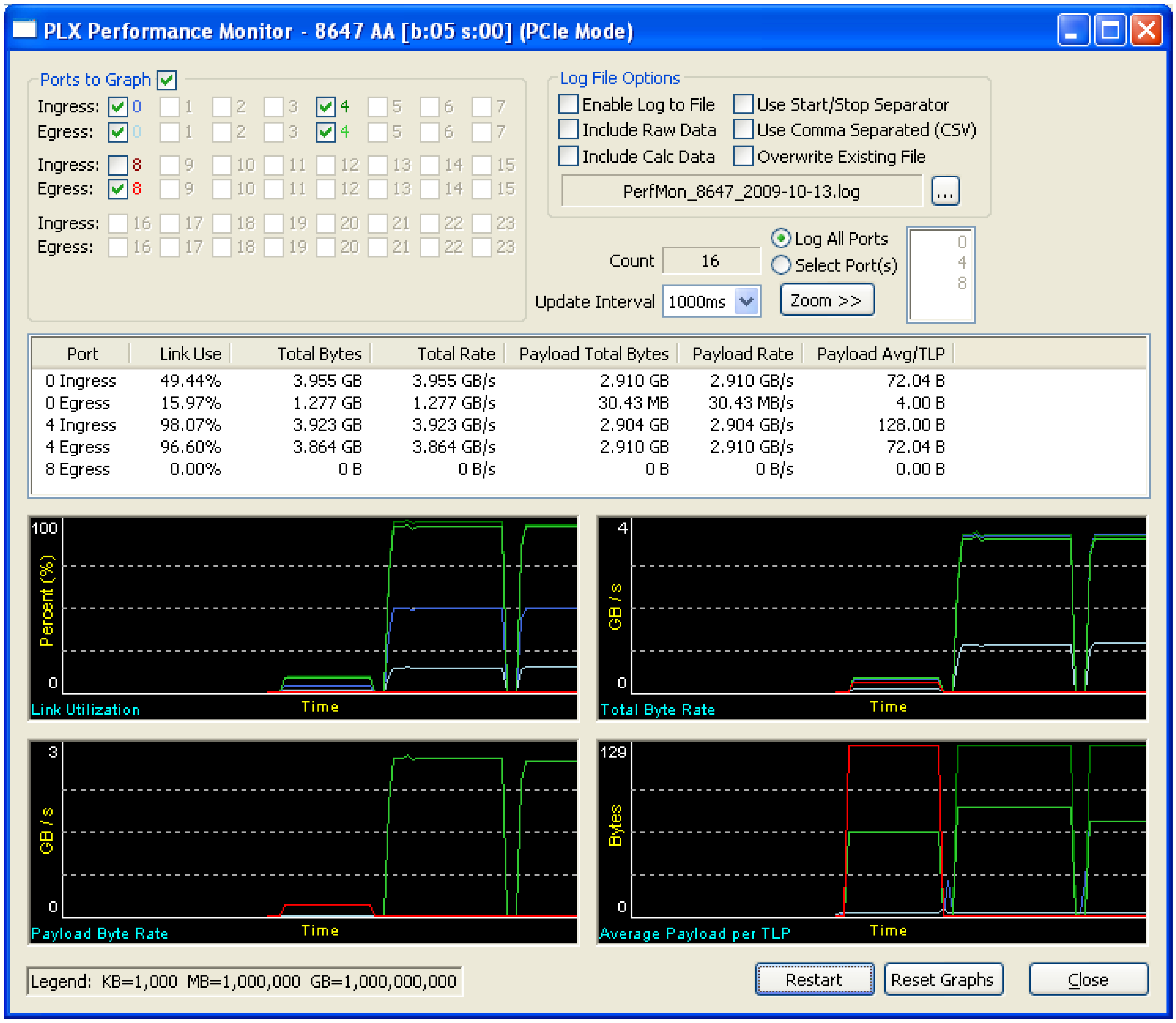
Update Interval 1000ms

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	49.16%	3.933 GB	3.933 GB/s	2.985 GB	2.985 GB/s	82.21 B
0 Egress	15.28%	1.222 GB	1.222 GB/s	30.10 MB	30.10 MB/s	4.00 B
4 Ingress	99.54%	3.982 GB	3.982 GB/s	2.979 GB	2.979 GB/s	128.00 B
4 Egress	96.55%	3.862 GB	3.862 GB/s	2.985 GB	2.985 GB/s	82.21 B
8 Egress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 11. DMA Transfer to PEX 8647 RDK with Extended Tags Enabled

For the next test, the extended tag was disabled, reducing the reads outstanding from 63x 128B to 31x 128B. The performance drops only slightly, but notice that the read completion size drops from 82.2B to 72.0B – which explains the throughput drop.



PLX Performance Monitor - 8647 AA [b:05 s:00] (PCIe Mode)

Ports to Graph ✓
Ingress: [✓ 0] 1 2 3 [✓ 4] 5 6 7
Egress: [✓ 0] 1 2 3 [✓ 4] 5 6 7
Ingress: 8 9 10 11 12 13 14 15
Egress: [✓ 8] 9 10 11 12 13 14 15
Ingress: 16 17 18 19 20 21 22 23
Egress: 16 17 18 19 20 21 22 23

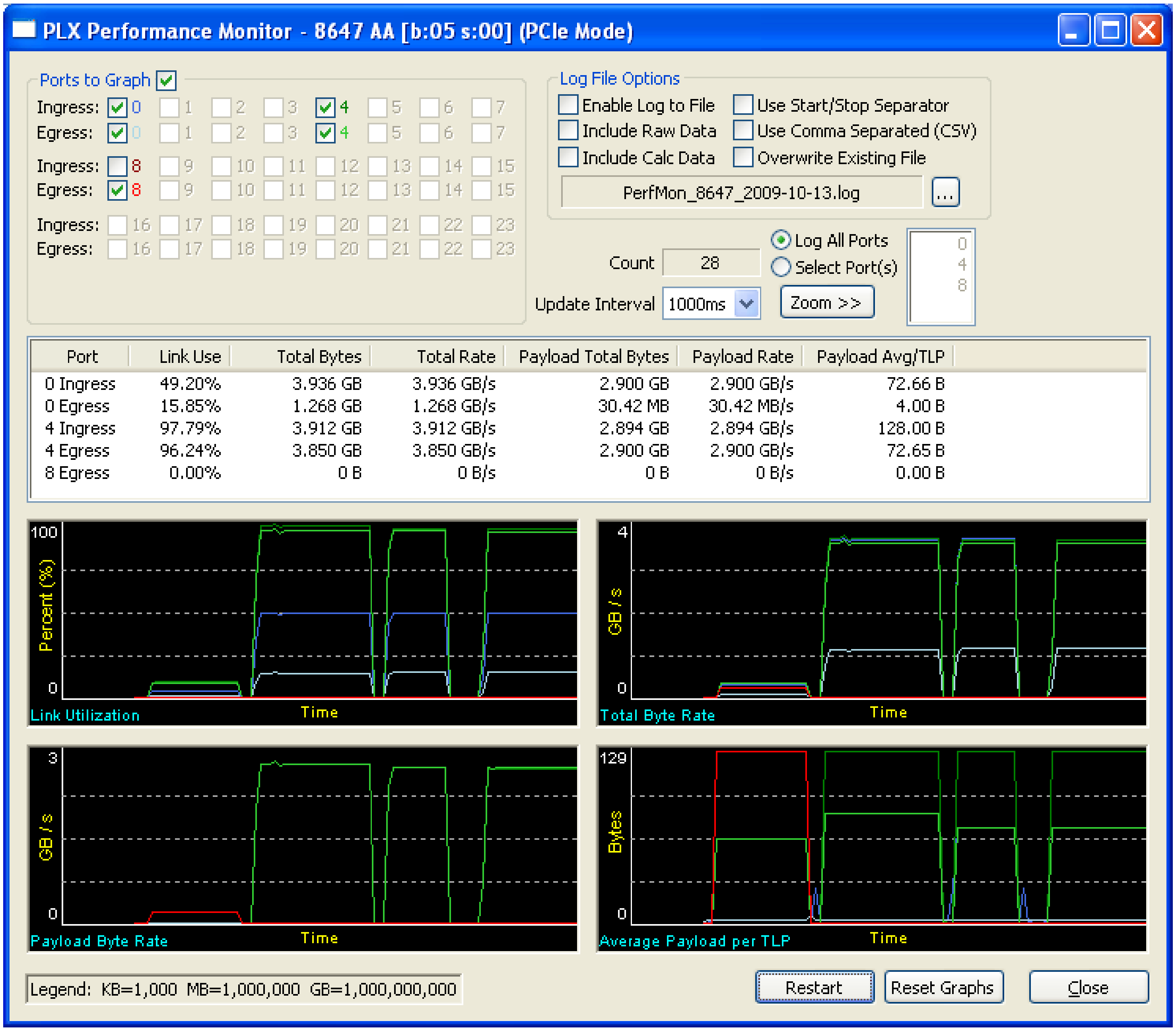
Log File Options
Enable Log to File; Use Start/Stop Separator; Include Raw Data; Use Comma Separated (CSV); Include Calc Data; Overwrite Existing File
PerfMon_8647_2009-10-13.log
Count 16
Update Interval 1000ms
[✓ Log All Ports] Select Port(s)
0 4 8

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	49.44%	3.955 GB	3.955 GB/s	2.910 GB	2.910 GB/s	72.04 B
0 Egress	15.97%	1.277 GB	1.277 GB/s	30.43 MB	30.43 MB/s	4.00 B
4 Ingress	98.07%	3.923 GB	3.923 GB/s	2.904 GB	2.904 GB/s	128.00 B
4 Egress	96.60%	3.864 GB	3.864 GB/s	2.910 GB	2.910 GB/s	72.04 B
8 Egress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 12. DMA Transfer to PEX 8647 RDK with Extended Tags Disabled

Exploring more around the number of reads outstanding, the next test has two DMA channels enabled (only 1 active), initially with extended tags enabled. With 2 channels enabled, each channel gets ½ of the tags: down from 63 to 31 each. This result is identical to the previous one, with 1 channel but no extended tag. Both tests have exactly 31 tags available to the active DMA channel.



Ports to Graph: Ingress 0, 4; Egress 0, 4, 8 checked

PerfMon_8647_2009-10-13.log

Count 28

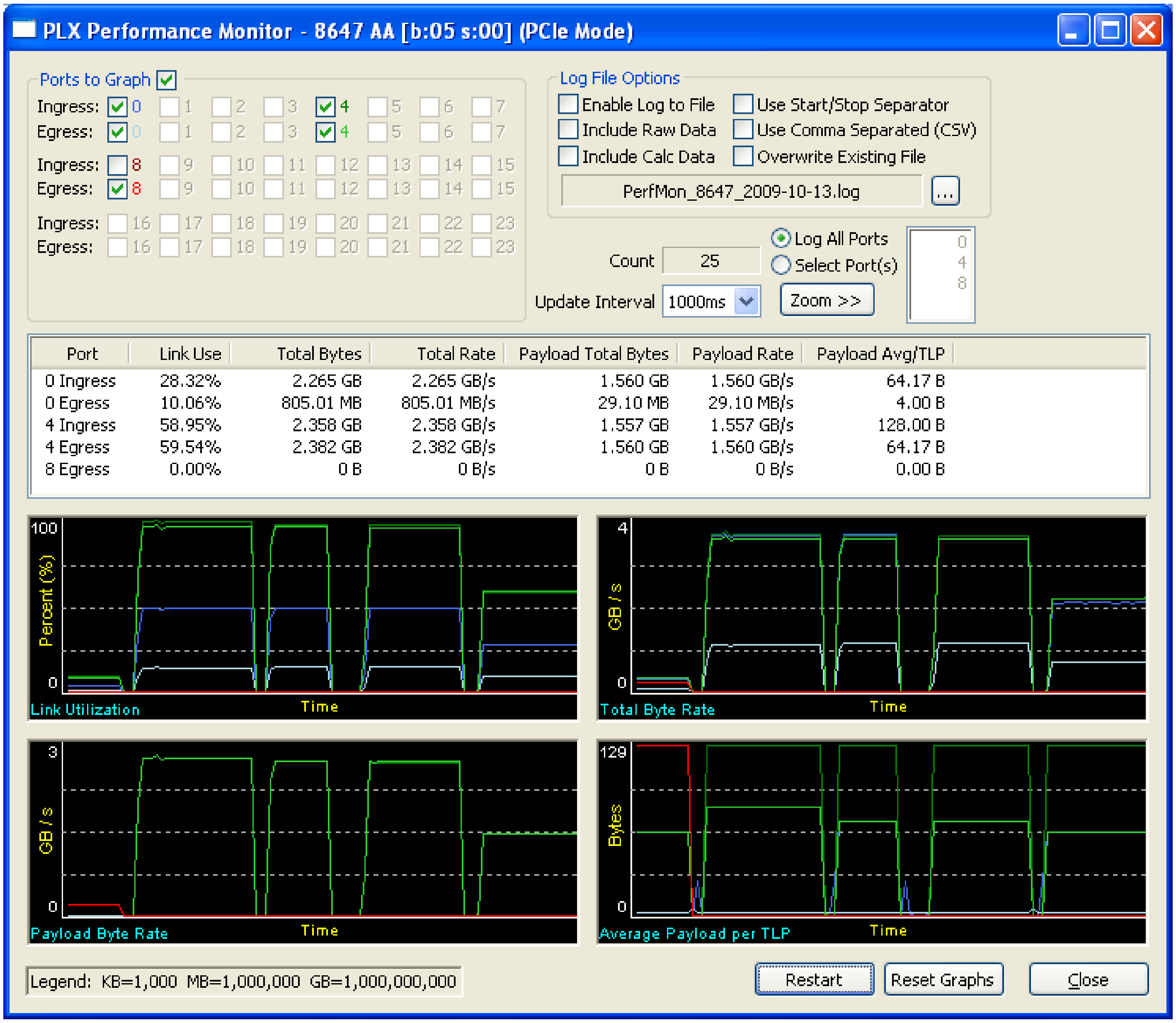
Update Interval 1000ms

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	49.20%	3.936 GB	3.936 GB/s	2.900 GB	2.900 GB/s	72.66 B
0 Egress	15.85%	1.268 GB	1.268 GB/s	30.42 MB	30.42 MB/s	4.00 B
4 Ingress	97.79%	3.912 GB	3.912 GB/s	2.894 GB	2.894 GB/s	128.00 B
4 Egress	96.24%	3.850 GB	3.850 GB/s	2.900 GB	2.900 GB/s	72.65 B
8 Egress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 13. DMA Transfer to PEX 8647 RDK – Two DMA Channels and Extended Tags Enabled

The next test has no extended tag, dropping the reads outstanding to 15 per channel. The performance drops remarkably, almost in half. Clearly, 15x 128B MRd are not enough to keep this particular system streaming completions to the DMA.


PLX Performance Monitor - 8647 AA [b:05 s:00] (PCIe Mode)
Ports to Graph ✓
Ingress: ✓ 0, 1, 2, 3, ✓ 4, 5, 6, 7
Egress: ✓ 0, 1, 2, 3, ✓ 4, 5, 6, 7
Ingress: 8, 9, 10, 11, 12, 13, 14, 15
Egress: ✓ 8, 9, 10, 11, 12, 13, 14, 15
Ingress: 16, 17, 18, 19, 20, 21, 22, 23
Egress: 16, 17, 18, 19, 20, 21, 22, 23
Log File Options
Enable Log to File
Include Raw Data
Include Calc Data
Use Start/Stop Separator
Use Comma Separated (CSV)
Overwrite Existing File
PerfMon_8647_2009-10-13.log
Count 25
Update Interval 1000ms
● Log All Ports
○ Select Port(s)
Zoom >>
0
4
8
Percent (%)
100
0
Link Utilization
Time
GB / s
4
0
Total Byte Rate
Time
GB / s
3
0
Payload Byte Rate
Time
Bytes
129
0
Average Payload per TLP
Time
Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000
Restart
Reset Graphs
Close

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	28.32%	2.265 GB	2.265 GB/s	1.560 GB	1.560 GB/s	64.17 B
0 Egress	10.06%	805.01 MB	805.01 MB/s	29.10 MB	29.10 MB/s	4.00 B
4 Ingress	58.95%	2.358 GB	2.358 GB/s	1.557 GB	1.557 GB/s	128.00 B
4 Egress	59.54%	2.382 GB	2.382 GB/s	1.560 GB	1.560 GB/s	64.17 B
8 Egress	0.00%	0 B	0 B/s	0 B	0 B/s	0.00 B

Figure 14. DMA Transfer to PEX 8647 RDK – Two DMA Channels and Extended Tags Disabled

7.4 Test2: DMA across NT Port (Read SystemA Mem, Write System B MEM)

All of the tests to this point had the 8619 DMA read upstream and write upstream. A test of reading upstream, writing downstream, and vice versa, was enabled using 2 PCs and the NT feature of the 8619. In this setup, there was still an 8647 plugged into PC System A in a gen2 x16 slot. An 8619, with the 8619 in x8_x8 mode with NT enabled, was plugged into port4 of the 8647. The 8619 connected via cables to a riser card, which plugged into the x16 slot on PC System B.

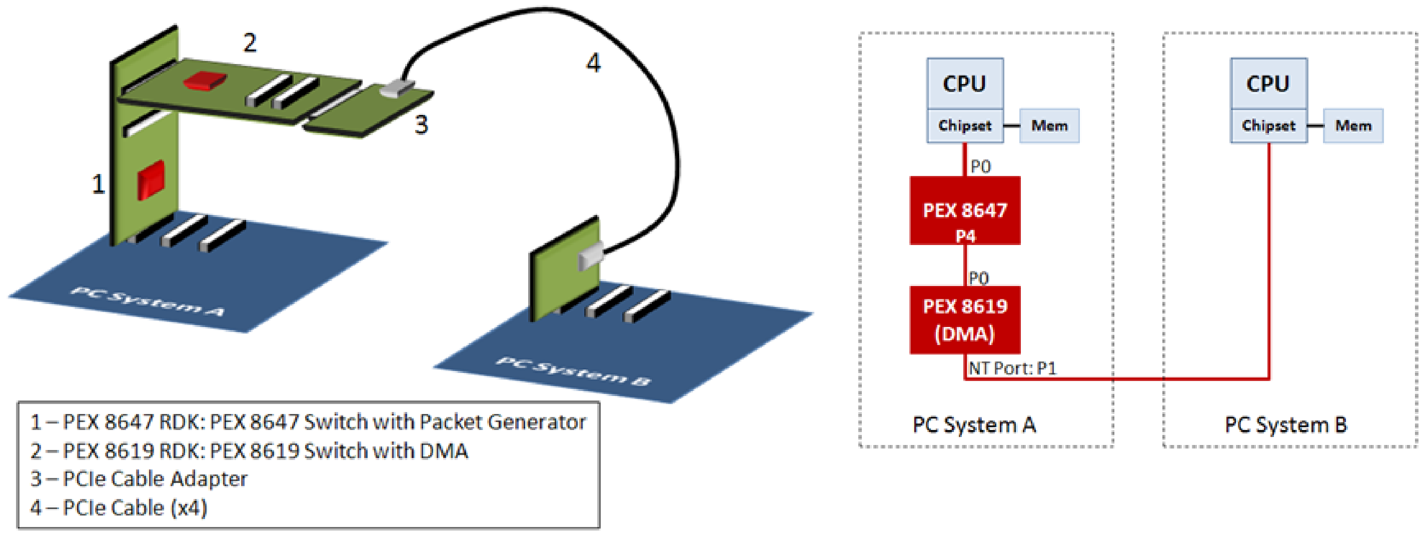


Figure 15. DMA Test System using the PEX 8647 RDK and PEX 8619 RDK with NT and DMA

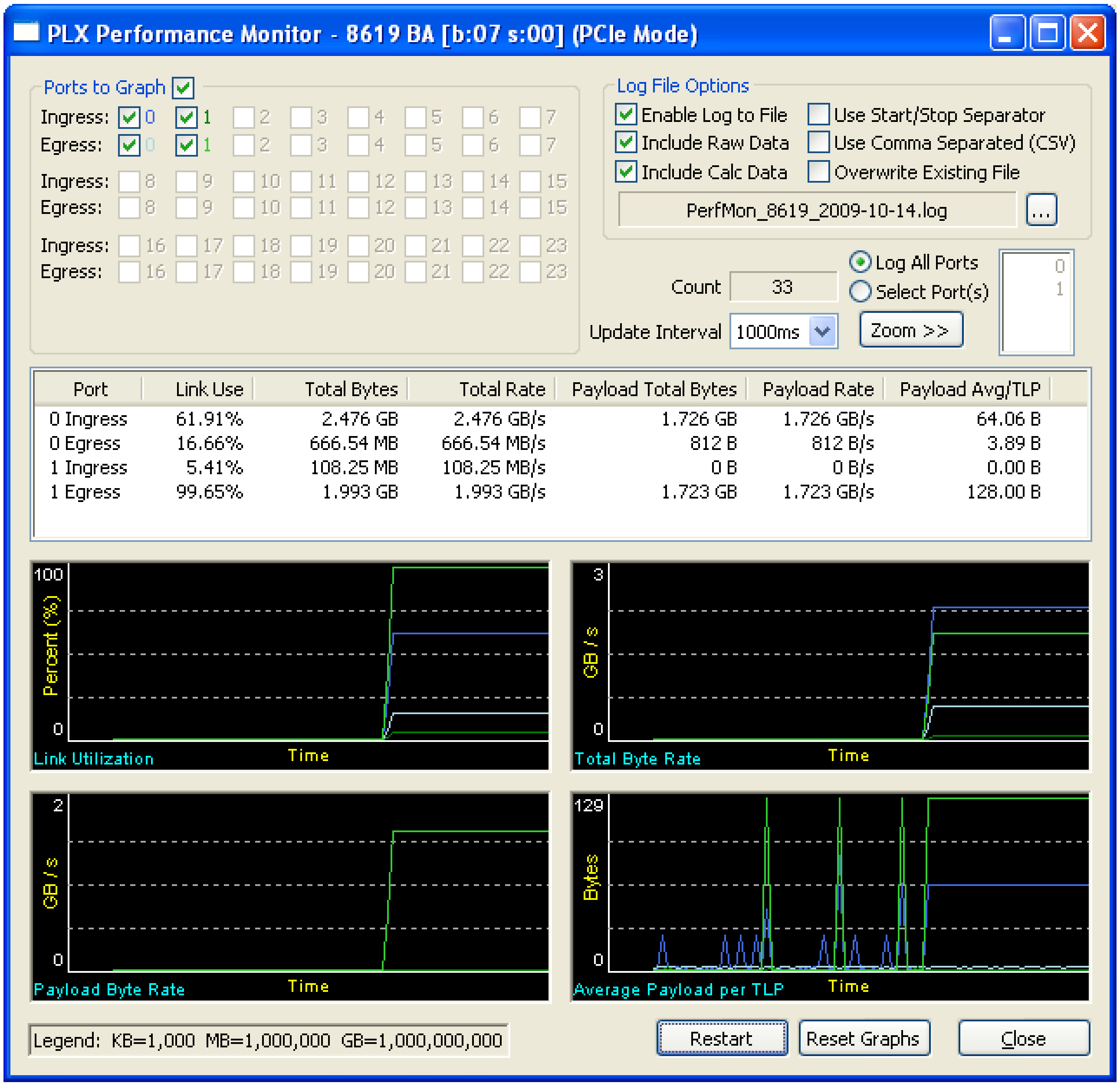
For this test, only an x4 cable between the systems was available and so the NT link down trained to x4.

Below is a brief description on configuring the NT port. Additional details on the NT port can be found in the PEX 8619 data sheet (www.plxtech.com/PEX8619).

A single window was opened using BAR2 of the virtual NT port to point to PC System B memory. The requester IDs for both the PEX 8619 DMA and the PC System A host (most upstream root port of PEX 8647/8619 hierarchy) were programmed into the NT Requester ID look up table for return completion routing. The System B memory space was a Python application memory space of 1 MB similar to System A, but with an independent address. Both System A host and 8619 DMA could read and write to System B memory via the NT BAR2.

To see the traffic going over the NT link, the monitor of the PEX 8619 was used. The PEX 8619 sees both the transparent upstream port (port0) and the NT downstream port (port1).

The first test reads transparent memory of system A via gen2 x8 on port0, and writes system B memory via the gen2 x4 NT link on port1. Port1 is clearly the bottleneck, getting to 99.65% utilization and 1.72 GB/s. A curious result is that system A completions are only 64B – something I cannot account for. Recall earlier tests where the completion size was 72B to 82B. It turns out that the read completions are not the bottleneck here, so it makes no difference.



Ports to Graph ✓

Ingress:	✓ 0	✓ 1	2	3	4	5	6	7
Egress:	✓ 0	✓ 1	2	3	4	5	6	7
Ingress:	8	9	10	11	12	13	14	15
Egress:	8	9	10	11	12	13	14	15
Ingress:	16	17	18	19	20	21	22	23
Egress:	16	17	18	19	20	21	22	23

Log File Options

✓ Enable Log to File — Use Start/Stop Separator
✓ Include Raw Data — Use Comma Separated (CSV)
✓ Include Calc Data — Overwrite Existing File

PerfMon_8619_2009-10-14.log

Count 33 — [✓ Log All Ports] Select Port(s) — 0 1

Update Interval 1000ms

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	61.91%	2.476 GB	2.476 GB/s	1.726 GB	1.726 GB/s	64.06 B
0 Egress	16.66%	666.54 MB	666.54 MB/s	812 B	812 B/s	3.89 B
1 Ingress	5.41%	108.25 MB	108.25 MB/s	0 B	0 B/s	0.00 B
1 Egress	99.65%	1.993 GB	1.993 GB/s	1.723 GB	1.723 GB/s	128.00 B

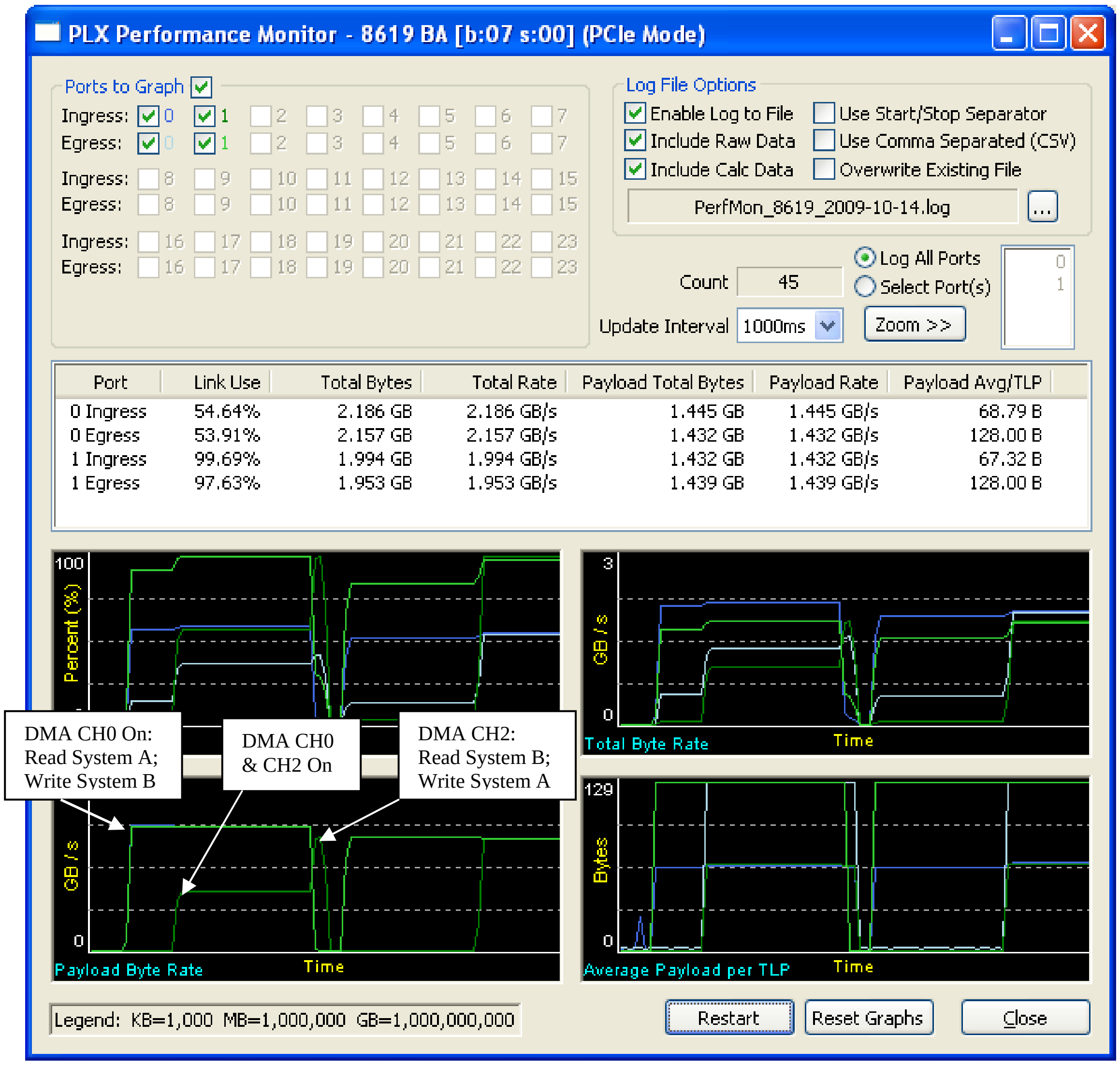
Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 16. DMA transfer from System A to System B

The next test has two channels enabled: one reading from System A, writing to System B; the other reading from System B, writing to System A.

In this scenario, there are two DMA channels going opposite directions. Each has a feedback loop on how fast it can inject MRd: it has to wait for a CplD before it can send a new MRd. But if one channel gets ahead of the other, then it will get further and further ahead as it sends more MWr in the other channel's MRd path, causing more delay on the MRd. As a result, performance in one direction could suffer.

This performance drop of one channel is exactly what happened in the first iteration of the test. The drop can be seen in the graphs with the callouts on the left side of the screen shot below.


PLX Performance Monitor - 8619 BA [b:07 s:00] (PCIe Mode)

Ports to Graph ✓
Ingress: ✓ 0 ✓ 1; Egress: ✓ 0 ✓ 1

Log File Options: ✓ Enable Log to File; ✓ Include Raw Data; ✓ Include Calc Data; Use Start/Stop Separator; Use Comma Separated (CSV); Overwrite Existing File
PerfMon_8619_2009-10-14.log

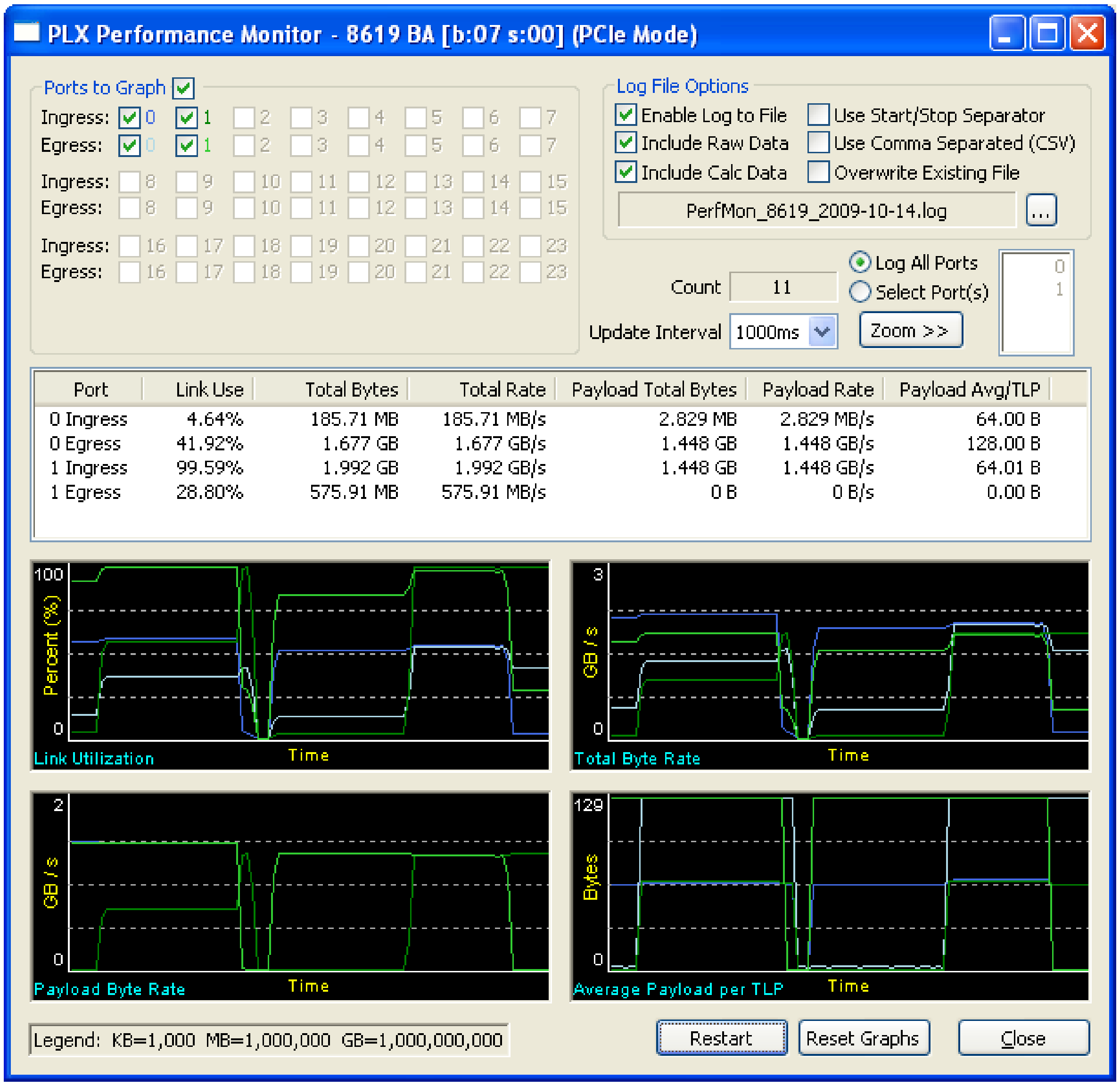
Count 45
Update Interval 1000ms
[● Log All Ports] Select Port(s)

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	54.64%	2.186 GB	2.186 GB/s	1.445 GB	1.445 GB/s	68.79 B
0 Egress	53.91%	2.157 GB	2.157 GB/s	1.432 GB	1.432 GB/s	128.00 B
1 Ingress	99.69%	1.994 GB	1.994 GB/s	1.432 GB	1.432 GB/s	67.32 B
1 Egress	97.63%	1.953 GB	1.953 GB/s	1.439 GB	1.439 GB/s	128.00 B

Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 17. Two DMA Channels Enabled with Traffic on Opposite Direction

The payload throughput left side of the graph shows non-delayed 2 channel traffic, where read system A/write System B starts first – and gets very good write throughput on port1. When read System B/write System A turns on, the port1 ingress throughput is about half of the other direction. The right side of the same graphs shows the effect of an 18 clock delay between reads (the next test). A delay of 18 clocks delay was chosen because 148 bytes / 8 bytes per clock is 18.5. Some tests were run to find 18 clocks got the best balanced throughput in this case. The throughput for both directions is closely balanced when a delay is applied to both channels for each read.Final test , reading from System B, writing to System A shows the following performance with 18 clock delay on the reads.



PLX Performance Monitor - 8619 BA [b:07 s:00] (PCIe Mode)

Ports to Graph ✓
Ingress: ✓ 0 ✓ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7
Egress: ✓ 0 ✓ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7
Ingress: ☐ 8 ☐ 9 ☐ 10 ☐ 11 ☐ 12 ☐ 13 ☐ 14 ☐ 15
Egress: ☐ 8 ☐ 9 ☐ 10 ☐ 11 ☐ 12 ☐ 13 ☐ 14 ☐ 15
Ingress: ☐ 16 ☐ 17 ☐ 18 ☐ 19 ☐ 20 ☐ 21 ☐ 22 ☐ 23
Egress: ☐ 16 ☐ 17 ☐ 18 ☐ 19 ☐ 20 ☐ 21 ☐ 22 ☐ 23

Log File Options
✓ Enable Log to File ☐ Use Start/Stop Separator
✓ Include Raw Data ☐ Use Comma Separated (CSV)
✓ Include Calc Data ☐ Overwrite Existing File
PerfMon_8619_2009-10-14.log

Count 11
Update Interval 1000ms
[✓ Log All Ports] Select Port(s)
0
1

Port	Link Use	Total Bytes	Total Rate	Payload Total Bytes	Payload Rate	Payload Avg/TLP
0 Ingress	4.64%	185.71 MB	185.71 MB/s	2.829 MB	2.829 MB/s	64.00 B
0 Egress	41.92%	1.677 GB	1.677 GB/s	1.448 GB	1.448 GB/s	128.00 B
1 Ingress	99.59%	1.992 GB	1.992 GB/s	1.448 GB	1.448 GB/s	64.01 B
1 Egress	28.80%	575.91 MB	575.91 MB/s	0 B	0 B/s	0.00 B

Legend: KB=1,000 MB=1,000,000 GB=1,000,000,000

Figure 18. Two DMA Channels Enabled with Traffic on Opposite Direction with 18 Clock Delay Between DMA Read Requests

For this test, the ingres port1 link is at 99.59% and is the bottleneck.

7.5 *Test summary*

With write size fixed at 128B, the upstream will be the bottleneck having both 1 MWr(128B) and 1MRd(128B) per 128B data, for 128/(128+20+20) = 0.7619 maximum link throughput, or 3.04 GB/s on 4.0 GB/s gen2 x8 link and 1.52 GB/s on gen2 x4 link.

Table 12. Test Result Summary for Single System

Test: To/From System A Mem	Outstanding Reads	Extended Tags Enabled	Ingress throughput GB/s	Ingress average payload size B	Egress throughput GB/s	Egress average payload size B
PEX 8647 read	infinite	NA	3.696	127.96	0	0
PEX 8647 write	0	NA	0	0	3.444	128
PEX 8647 bi-dir	infinite	NA	2.475	79.73	2.475	128
PEX 8619 8KB transfer	31	no	2.230	81.09	2.226	128
PEX 8619 8KB transfer	63	yes	2.527	85.27	2.522	128
PEX 8619 new RC	63	yes	2.985	82.21	2.979	128
PEX 8619 new RC	31	no	2.910	72.04	2.904	128
PEX 8619*	31	yes	2.90	72.65	2.84	128
PEX 8619*	15	no	1.56	64.12	1.56	128
PEX 8619 512B transfer	31	no	1.195	72.01	1.159	128

Calibration Results
DMA Results

* This setup enabled two DMA Channels. However, only one DMA channel was active during test. All other tests enable one DMA channel only

Table 13. Test Result Summary for System A and System B Behind the PEX 8619 NT Port (x4)

DMA Direction	DMA Read Request Delay (clocks)	System A DMA Read Completion Rate (GB/s)	System B DMA Write Rate (GB/s)	System B DMA Read Completion Rate (GB/s)	System A DMA Write Rate (GB/s)
A →B only	18	1.451	1.448	0	0
B →A only	18	0	0	1.448	1.448
bi-direction	18	1.445	1.439	1.432	1.432
A →B only	0	1.726	1.723	0	0
B →A only	0	0	0	1.461	1.461
bi-direction	0	1.5	1.5	0.7	0.7

8 Example Python Code Used for Initial Configuration

The following python script shows how the DMA was configured for the two channel testing. The configuration was quite similar for one channel testing, with just the addresses changed and 1 channel enabled instead of 2.

```
# ===============================================================================
# Global CNTL
#
#  [1:0] = 0 = 4 ch
#        = 1 = ch 0
#        = 2 = ch 0 & 2
# [2]    = 0 = external desc
#        = 1 = internal desc: put this before writing the descriptors

regnum = 0x1FC;
wdata  = 0x02; # ch 0 & 2
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# extended tag enable
regnum = 0x70;
wdata  = 0x100 # bit 8 is extended tag enable
# wdata = 0x000 # extended tag clear
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# ===============================================================================
# Write DMA off-chip descriptor memory

desc_list = [None]*4
desc_adr = desc_mem_base_ch0
dest_adr = dest_mem_base_ch0
src_adr = src_mem_base_ch0

## Format for first dword of descriptor
# [31]=desc valid
# [30]=intr when done
# [29]= read adr cnstant
# [28]= write adr cnstant
# [27]=0
# [26:0]= DMA length in bytes

for i in range (desc_cnt_ch0):
    desc_list[0] = 0x80000000+dma_length   # [31] is valid bit
    desc_list[1] = 0x00000000              # [31:16]=upper src, [15:0]=upper dest
    desc_list[2] = dest_adr                # [31:0]=lower dest
    desc_list[3] = src_adr                 # [31:0]=lower src

    for ii in range (4):
        rl.MemWritePrt (desc_adr+(ii*4), desc_list[ii])

    # Set for the next descriptor
    desc_adr += 0x10
    dest_adr += dma_length
    src_adr += dma_length

## channel 2
desc_adr = desc_mem_base_ch2
dest_adr = dest_mem_base_ch2 # getting dropped
src_adr = src_mem_base_ch2 # different read address for channel 2

for i in range (desc_cnt_ch0):
    desc_list[0] = 0x80000000+dma_lengt    # [31] valid bit
    desc_list[1] = 0x00000000              # [31:16]=upper src, [15:0]=upper dest
    desc_list[2] = dest_adr                # [31:0]=lower dest
    desc_list[3] = src_adr                 # [31:0]=lower src

    for ii in range (4):
        rl.MemWritePrt (desc_adr+(ii*4), desc_list[ii])

    # Set for the next descriptor
    desc_adr += 0x10
    dest_adr += dma_length
    src_adr += dma_length

# ========================================================================
# Program the DMA channel config registers

#(up_busnum, up_dev, dma_func, channel, desc_adr_l, desc_adr_u, desc_cnt, pref_limit, dma_len, bandwd):

bandwd = 0 # 0 clock delay after sending data read
rl.ExtWriteDmaCh(up_busnum, up_dev, dma_func, 0, desc_mem_base_ch0, 0x0, desc_cnt_ch0,
desc_pre_fetch, dma_length, bandwd)
# same desc count for ch0 and 2
rl.ExtWriteDmaCh(up_busnum, up_dev, dma_func, 2, desc_mem_base_ch2, 0x0, desc_cnt_ch0,
desc_pre_fetch, dma_length, bandwd)

# ====================================================================
# DMA start
# 3 start
# 4 ring mode 1 / block mode 0
# 5 stop continuous ring 0 (0 = continuous, 1 = stop)
# 15:13 -- wait for something?  000 = no wait
# 18:16 max transfer size 0=64, 1=128, 2=256, 3=512, 4=1K, 5=2K, 7=4 byte
# 21:19 TC
# 22 relaxed ordering descr read
# 23 RO data read
# 24 RO desc write
# 25 RO data write
# 29:26 no snoop
# 30 active channel status
# 31  header logging valid

data = 0x3c010018

key_in = raw_input ("\nPress CR to start DMA ch0 -> ")
rl.MemMapWrPrt(up_busnum, up_dev, dma_func, 0x238, data)

key_in = raw_input ("\nPress CR to start DMA ch2 -> ")
rl.MemMapWrPrt(up_busnum, up_dev, dma_func, 0x438, data)

key_in = raw_input ("\nPress CR to stop DMA ch0 -> ")
rl.MemMapWrPrt(up_busnum, up_dev, dma_func, 0x238, 0x00016011) # Set Pause bit

key_in = raw_input ("\nPress CR to stop DMA ch2 -> ")
rl.MemMapWrPrt(up_busnum, up_dev, dma_func, 0x438, 0x00016011) # Set Pause bit

# end of main program


# ==========================================================================
# function call to write external DMA descriptors

def ExtWriteDmaCh(up_busnum, up_dev, dma_func, channel, desc_adr_l, desc_adr_u, \
                desc_cnt, pref_limit, dma_len, bandwd):

    ch_offset = (0x100 * channel) + 0x200

    # ==========================================================================
    # Desc ring base memory address, lower

    regnum = ch_offset + 0x14
```

```
wdata  = desc_adr_l
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# ==========================================================================
# Desc ring base memory address, upper

regnum = ch_offset + 0x18
wdata  = desc_adr_u
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# ===========================================================================
# Next, desc ring base memory address, lower

regnum = ch_offset + 0x1C
wdata  = desc_adr_l
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# ===========================================================================
# Desc count

regnum = ch_offset + 0x20
wdata  = desc_cnt
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# ===========================================================================
# Channel Bandwith

regnum = ch_offset + 0x2C
wdata  = bandwd
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# =========================================================================
# Abort timer

regnum = ch_offset + 0x30
wdata  = 0x010
p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

# ===========================================================================
# Prefetch limit

regnum = ch_offset + 0x34
wdata  = pref_limit
```

```
    p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

    # ==========================================================================
    # DMA INTR - CNTL/Status

    regnum = ch_offset + 0x3C
    wdata  = 0x00
    p.MemoryMapWrite(up_busnum, up_dev, dma_func, regnum, wdata)

    # ===========================================================================
    # DMA INTR - CNTL/Status (reg 54)

    # Use defaults

    return ()

# end function call to write external DMA descriptors
# ==========================================================================
```

9 Appendix B: Intel Tylersburg/Nehalem Test

The exact same configuration was tested on a Nehalem/Tylerburg single socket system briefly. The packet generator calibration got to only ~739 MB/s for read, write, or read/write, which indicated some tweak needed to be made to the packet generator test sequence.

However, the DMA tests were outstanding. An MPS of 256B was supported, so that was tested. The time on the machine was limited, so only the read Mem, write Mem test was performed. However, the link got to 99% utilization, so the memory controller was not a bottleneck.

Table 14. Tylesburg/Nehalem System DMA Results

Descriptor	Transfer Size	Ext Tag	Throughput	MPS	Avg Pay
16	8192	no	3.387	256	
16	512	no	1.497	256	
16	512	yes	1.497	256	
16	1024	yes	2.690	256	149
16	2048	yes	3.381	256	221
16	2048	no	hung	256	hung