Mälardalen University

Performance of a Communication Mechanism over the PCI-bus

Leif Enblom

1 Introduction

This document describes the evaluation of a communication mechanism in the form of a message queue that has been implemented for communication over a CPCI passive backplane dual board system. The document will mainly treat the implementation and present performance measurements and lessons learned.

2 Overview

This document describes a simple message passing class that utilizes a shared memory area in an architecture with a system and a non-system board connected with a passive CPCI-backplane bus. Both the System and the Non-System boards form were Pentium III boards running at 850MHz. More specifically they were CT7 boards from SBS Technologies [SBS].

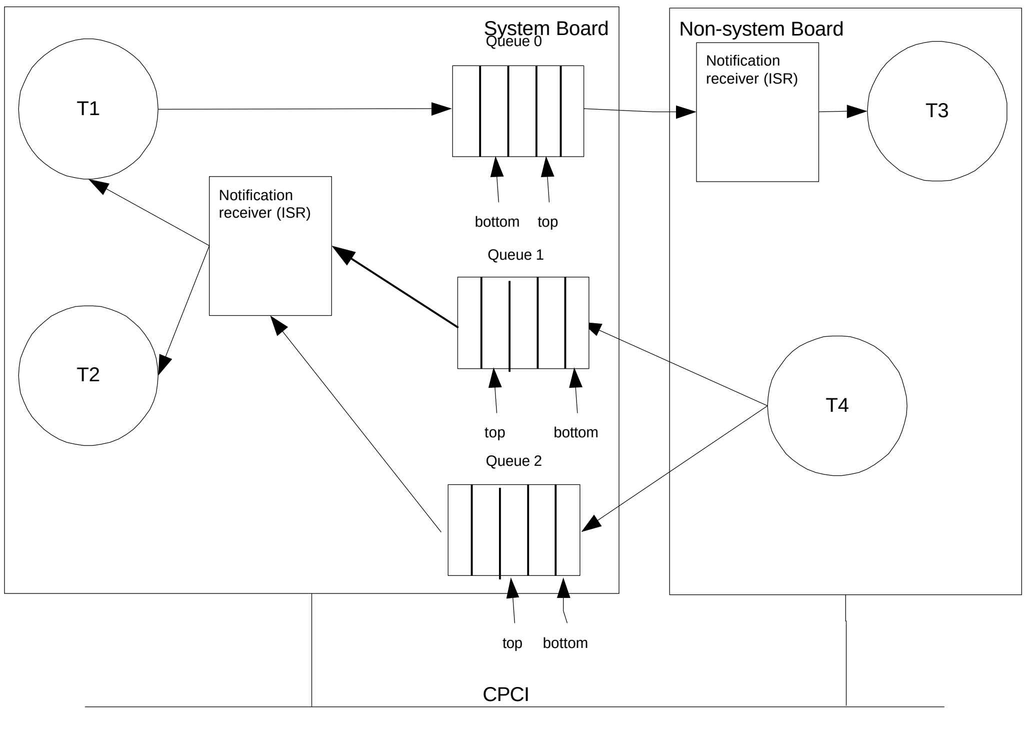


Figure 1,Overview of SimpleMessageQueue.

As can be seen in Figure 1 the receiver on each board is the notification receiver. Notification of the receiver is abstracted and can be modified to fit in cooperation with signals in UNIX or supporting notification over the PCI-bus. In the case of a PCI-interrupt over for example CPCI, the notification receiver is represented by an ISR. The first design will be for an environment over CPCI, so the ISR will be awakened by an interrupt over CPCI and the doorbell register in the Intel 21554 bridge residing on the non-system board. In the case with VxWorks, the receiving thread, as for example T2 in Figure 1, will take a semaphore and wait forever. When a message is ready the ISR will release the semaphore and T2 can act upon the data. The sender does not go through any layer of code, the SimpleMessageQueue class acts directly upon the respective registers and data structures. This makes SimpleMessageQueue feasible to use in operating system environments where threads are allowed to act upon hardware directly such as in the case of VxWorks. In these tests VxWorks has been used as operating system.

The first version supports only one reader and one writer. Is there need for multiple writers this can be solved in the future by protecting the writing actions with a semaphore on the "writing side". It is important to understand that if the senders are located on multiple processors, mutual exclusion has to be achieved between them. My recommendation is to use the LOCK# functional signal on the PCI-bus to achieve atomic read/modify/write transactions and thus the ability to build shared semaphores among nodes. Multiple readers may on the other hand not be necessary; it is better for the user to specify directly where the message is bound by waiting on messages on only one queue.

2.1 Summary of Features

SimpleMessageQueue is capable and limited to these features listed below at the moment:

- Only one reader per message queue.
- Receivers must be started before first message is sent. SimpleMessageQueue may be in need of a startup synchronization service to prevent these effects.
- The system board is responsible for clearing the message queues and at startup.
- Today the senders do not synchronize. Multiple senders must synchronize in the future. It is possible today that an interrupt is issued to the receiving board before a previous interrupt was acknowledged. These two limitations are easy to solve, but were not necessary to implement for the ping-pong test.

3 Memory Layout

This version of SimpleMessageQueue is based on a shared memory area on the system slot board. VxWorks by default wants to control all of the available primary memory, which can be a threat to the consistency of a shared memory area. This has been altered in the BSP, more specifically the definition of USER_RESERVED_MEM. This define changes the amount of memory from the top and downward that will not be controlled by VxWorks.

The only information that has to be provided to find the shared memory area on either side of the message queue (system or non-system) are the respective base addresses. On the system side this base address is provided in the SMQ_SHARED_MEMORY define and on the non-system side it is represented by SMQ_UPSTREAM_BASEADDRESS.

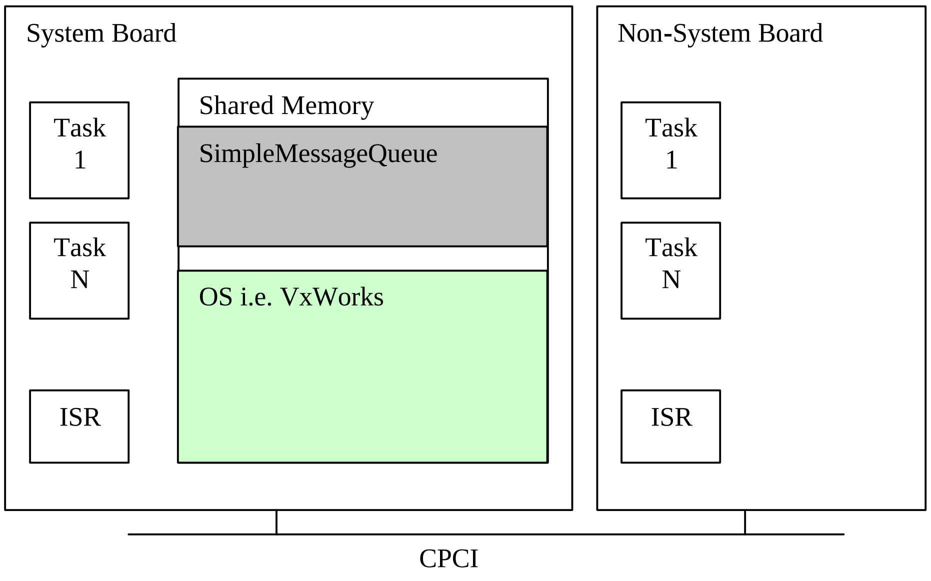


Figure 2, SimpleMessageQueue in the architecture.

Figure 3 below illustrates how the message queues are places in memory. From SMQ_SHARED_MEMORY and upwards the respective message queue with index 0, 1, 2 up to N are located.

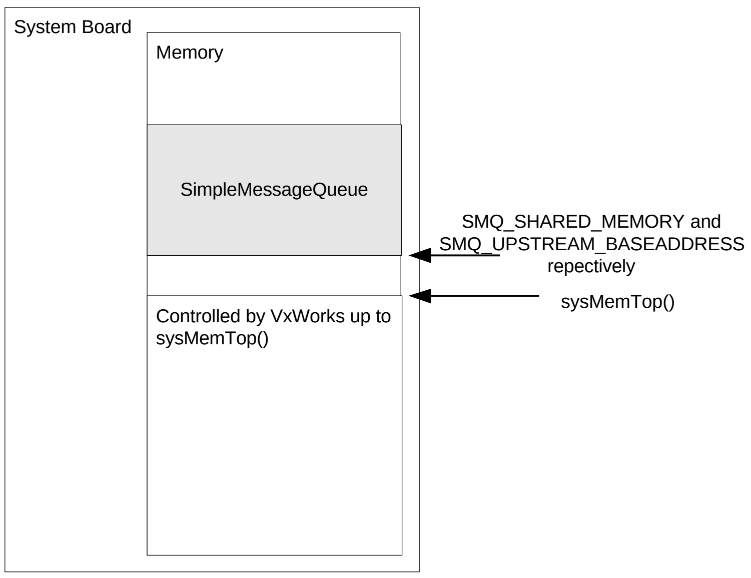

System Board
Memory
SimpleMessageQueue
SMQ_SHARED_MEMORY and SMQ_UPSTREAM_BASEADDRESS repectively
sysMemTop()
Controlled by VxWorks up to sysMemTop()

Figure 3, Memory Layout on System Board

The whole queue is described in a struct, SMQ, and sub-structs defined in simpleMessageQueueDefines.h. The memory layout of SimpleMessasgeQueue is illustrated in Figure 4 below. Each queue is described is in more detail in conjunction with Figure 5.

As can be seen in Figure 4 there exists

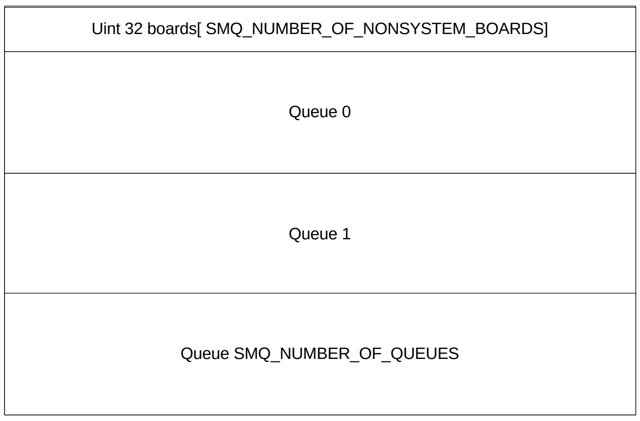

Uint 32 boards[SMQ_NUMBER_OF_NONSYSTEM_BOARDS]
Queue 0
Queue 1
Queue SMQ_NUMBER_OF_QUEUES

Figure 4, SimpleMessageQueue memory layout.

Each message queue as shown in the figure above consists of the data illustrated in Figure 5. There exists a top and a bottom pointer that will make the message queue act like a ring-buffer. Since only one writer and only one receiver is active at every instant, there will be no demand on locking mechanisms. A flow control field has been added for future use. It may be interesting to enable receivers to notify the sender of the activities at the receiving side. A receiver may for example want to inform senders of the messages currently read. In the implementation today were the each sending of a message is accompanied with an interrupt this information is inherently incorporated in the clearing of the interrupt. But in the case that the number of interrupts wants to be decreased, multiple messages may want to be sent per notification, and information between the receiver and the sender may want to be sent to notify about progress.

top	bottom	flowCont rol	index	not allocated
size	sender	message 0 [SMQ_ENTRY_SIZE]		
size	sender	message 1 [SMQ_ENTRY_SIZE]		
size	sender	message 2 [SMQ_ENTRY_SIZE]		
size	sender	message SMQ_NUMBER_OF_ENTRIES [SMQ_ENTRY_SIZE]		

Figure 5, Data Structure of one message queues, amount of bytes in brackets.

Each message entry is limited to SMQ_ENTRY_SIZE and is accompanied by information of the size of the message and the id of the sender. FlowControl information is accompanied to the queue for future use. Probable use is notification of read messages from the receiver to the sender and when a “batch” size (allowing multiple messages to be sent with only one notification) is defined. Each queue is also “aware” of its number, which is located in index.

4 Interface

Every sender and receiver has to instantiate SimpleMessageQueue, and has to provide parameters that will point out a specific index as well as setting the type of the queue and if it is a sender or a receiver.

SimpleMessageQueue(Uint32 in_index, Uint32 in_senderReceiver, Uint32 in_queueType)
Constructor that will enable the user to access the queue through the instantiated object. If the user instantiated the object as a receiver, a semaphore will be created that afterwards will be used by the receive method to wait for messages.

Uint32 SimpleMessageQueue::**send**(const void *data, Uint32 length, Uint32 timeout, Uint32 priority)
Sends a message of size length to the receiver (receiving queue was declared at instantiation.

Uint32 SimpleMessageQueue::**receive**(void *data, Uint32 *length, Uint32 timeToWait)
Through this method a thread can wait a period of time defined by timeToWait for a message to arrive to the queue. The user is responsible for having allocated at least the size of an entry at the memory area where the data pointer points to.

void SimpleMessageQueue::**notify**(void)
This method is used by the send method and notifies the receiver about a sent message. It is not intended to be used by the user and should thus be private declared.

Uint32 SimpleMessageQueue::**init**(Uint32 in_boardNumber, Uint32 in_numberOfBoards)
This method should be used at initialization of the application before any objects are created. It configures the board according to if it is a system or a non-system board. The 21554 bridge and the queues are configured according to the defines declared in simpleMessageQueueDefines.h. The boardNumber must be provided (use the slot number where the board is place in the chassis) as well as the number of boards in the system. This information is used for the boards to synchronize at startup.

void SimpleMessageQueue::**boardInitialized**(Uint32 boardNumber)
This mthod should be used by the non-system boards to notify the master, and thus the system, when they are up and running.

void SimpleMessageQueue::**synchronize**(Uint32 nice)

A method that enables the master to wait for all other boards to come up before it continues. The user can through the nice parameter calibrate the polling period of the status of the non-system boards.

void SimpleMessageQueue::**ISRNonSystemBoard**(int parameter)
void SimpleMessageQueue::ISRSystemBoard(int parameter)
These are the Interrupt Service Routines that are responsible for waking up the receiving thread.

5 Programming Examples

This section will illustrate the use of SimpleMessageQueue through a short example. One thread on each board is created. The system board sends a message to through SimpleMessageQueue number 0, and then waits for a message from the non-system board to arrive on SimpleMessageQueue number 1. The same is coded at the non-system board except that this thread first waits on a message from SimpleMessageQueue number 0 and then sends a message to SimpleMessageQueue number 1. This is a ping-pong communication test. The code for the system thread could look like this:

```
static void System( void )
{
  SimpleMessageQueue queuesend( 1, SMQ_TYPE_SENDER, SMQ_QUEUETYPE_FIFO );
  SimpleMessageQueue queuerec( 0, SMQ_TYPE_RECEIVER, SMQ_QUEUETYPE_FIFO );
  char buff[100];                    /* Storage area for text output */
  char data[1024] = "01234567890123456789";      /* Storage area to send message */
  Boolean run = TRUE;
  Uint32 size = 1;
  Uint32 length;

  size = 100;
  count = 0;

  while( run == TRUE ) {
        count++;
        if( queuesend.send( (void *)data, size, SMQ_WAIT_NOT, SMQ_NORMAL_PRIORITY ) !=
SMQ_OK ) {
                TRCPRN( traceIdTwinPowerSender, ("Error sending message to non-system board\n") );
        }
        if( queuerec.receive( (void *)data, &length, WAIT_FOREVER ) != SMQ_OK ) {
              TRCPRN( traceIdTwinPowerSender, ("Error while receiving message.\n") );
              vosSleepMs( 1 );
        }
        if( count %10000 == 0 ) {
              TRCPRN( traceIdTwinPowerSender, ("Sending 10000 messages to non-system board\n") );
        }
  }
```

The code at the non-system board could look like this:

```
static void twinPowerSender( void )
{
  SimpleMessageQueue queuesend( 0, SMQ_TYPE_SENDER, SMQ_QUEUETYPE_FIFO );
  SimpleMessageQueue queuerec( 1, SMQ_TYPE_RECEIVER, SMQ_QUEUETYPE_FIFO );
  char buff[100];                    /* Storage area for text output */
  char data[1024] = "01234567890123456789";      /* Storage area to send message */
  Uint32 size = 1;
  Uint32 count;
  Uint32 length;
  Boolean run = TRUE;

  size = 100;
  count = 0;
```

```
        while( run == TRUE ) {
                count++;
                if( queuerec.receive( (void *)data, &length, WAIT_FOREVER ) != SMQ_OK ) {
                        TRCPRN( traceIdTwinPowerReceiver, ("Error while receiving message.\n") );
                        vosSleepMs( 1 );
                }
                if( queuesend.send( (void *)data, size, SMQ_WAIT_NOT, SMQ_NORMAL_PRIORITY ) !=
SMQ_OK ) {
                                TRCPRN( traceIdTwinPowerSender, ("Error sending message to system board\n") );
                }
                if( count % 100000 == 0 ) {
                        TRCPRN( traceIdTwinPowerSender, ("Sending 100000 messages to system board\n") );
                }
        }
    }
```

6 Performance Measurements

A number of performance measurements have been performed to investigate behavior of mainly communication over the PCI-bus. Each test was made on two SBS CT7 [SBS] CPCI boards running at 850MHz.

Bus access latencies, the amount of time that expires from the moment a bus master requests the use of the PCI-bus until it completes the first data transfer of the transaction, consist of three components; arbitration latency, bus acquisition latency and target latency. Please observe that these latencies are regarding one bus. A bus-hierarchy as in the case of a local PCI-bus and a CPCI-bus, these latencies will accumulate.

The first simple test performed was a uni-directional communication from the system to non-system board. With the help of the spy tool in VxWorks, the average execution percentage of the background task in the base could be observed. Please observe that the spy tool rounds off figures. Therefore the sum of the percentages may not sum up exactly to 100%.

A clear tendency could be seen at once; the non-system board was having performance problems. The figures are presented in Figure 6 below. It is already here clear that performance is suffering from reads from the non-system board over the PCI-hierarchy.

Message Size	Frequency	Load on System Board	Load on Non-System Board
10 byte	1000Hz	1%	3%
100 byte	1000Hz	1%	7%
1000 byte	1000Hz	2%	59%

Figure 6, System load with uni-directional communication from system to non-system board.

The next test was more elaborate, creating two message queues and stressing the system with a ping-pong test. One thread on each board exchanges messages in an interleaved fashion, running as fast as they can. Complete copying from the memory area of each thread to the other is performed in the process. This test yielded results that are presented in Figure 7 below. The SendReceive threads are user threads performing ping-pong message passing. Bkgnd is a thread running at a low priority, acting like a idle thread. Kernel is the fraction of time spent in the Kernel as well as Int that depicts the fration of time spent handling interrupts (in the kernel).

Message Size	Messages per second and direction	Load on individual threads or modules on System Board		Load on individual threads or modules on Non-System Boards	
1000 byte	2224	SendReceive	1%	SendReceive	95%
		Bkgnd	96%	Bkgnd	2%
		Kernel	0%	Kernel	0%
		Int	1%	Int	1%

100 byte	15223	SendReceive 4% Bkgnd 87% Kernel 2% Int 5%	SendReceive 75% Bkgnd 13% Kernel 2% Int 7%
10 byte	34000 ± 100	SendReceive 9% Bkgnd 70% Kernel 6% Int 14%	SendReceive 51% Bkgnd 28% Kernel 5% Int 14%
0 byte	45400 ± 200	SendReceive 11% Bkgnd 60% Kernel 6% Int 20%	SendReceive 38% Bkgnd 35% Kernel 6% Int 20%

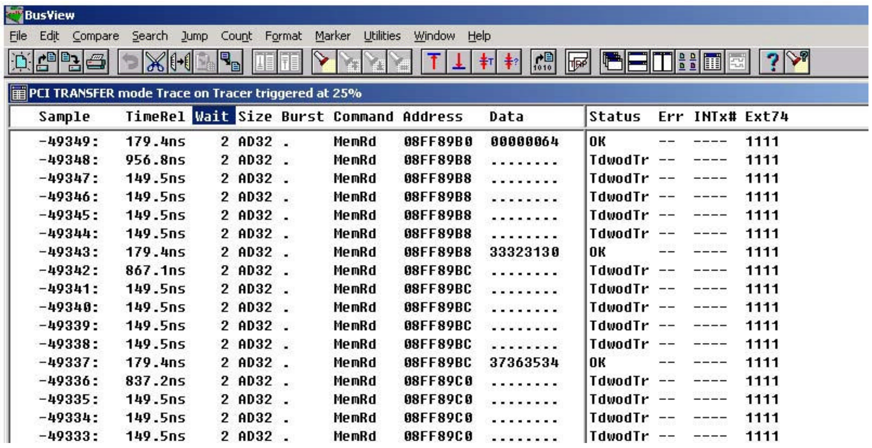
Figure 7, Performance of Ping-Pong test between system and non-system boards.

It is interesting to draw conclusions from these figures. First of all, as has been noted earlier, the reads from the non-system slot are disastrous for performance and is reflected in the load on the send thread. The SendReceive thread is responsible for reading the message, and in the case of large messages the execution of this thread is occupying the whole processor. As the messages become smaller and smaller, the significance of the reads become lesser, while the significance of context switches and interrupt handling becomes more significant. The Kernel module and interrupt module figures in Figure 7 are good indicators on context switch overhead and interrupt overhead.

Quite impressive is the pure notification case, where the message size is zero. The boards are able to handle 45400 interrupts, including one semaphore release per interrupt, as well as a context switch to the receiving thread.

Another interesting figure to note is the load on SendReceive on the system board in the case of large messages of 1000 bytes. The load is as low as 1% for the copying to the message area. This is due to copying to a memory with good locality, on the board itself.

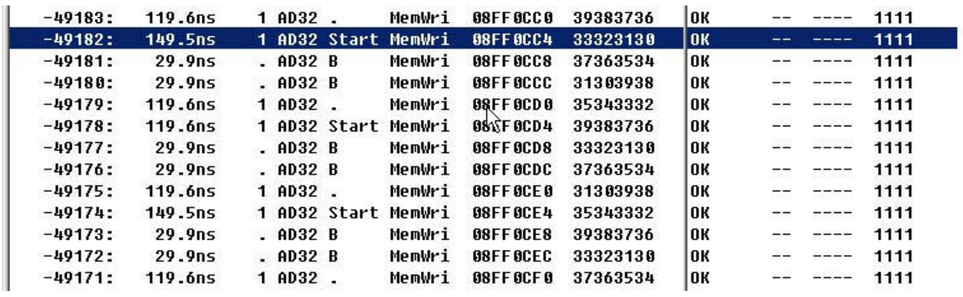
The significance and latency of transactions on the PCI-bus can be analyzed with the bus-analyzer from VMETRO as can be seen in Figure 8 below. The figures are shown in the example with a message size of 100 bytes and the analyzator was placed on the CPCI-bus.


BusView
File Edit Compare Search Jump Count Format Marker Utilities Window Help
PCI TRANSFER mode Trace on Tracer triggered at 25%

```
   Sample     TimeRel Wait Size Burst Command Address   Data     Status   Err INTx# Ext74
   -49349:    179.4ns     2 AD32 .    MemRd   08FF89B0  00000064 OK       --  ----  1111
   -49348:    956.8ns     2 AD32 .    MemRd   08FF89B8  ........ TdwodTr  --  ----  1111
   -49347:    149.5ns     2 AD32 .    MemRd   08FF89B8  ........ TdwodTr  --  ----  1111
   -49346:    149.5ns     2 AD32 .    MemRd   08FF89B8  ........ TdwodTr  --  ----  1111
   -49345:    149.5ns     2 AD32 .    MemRd   08FF89B8  ........ TdwodTr  --  ----  1111
   -49344:    149.5ns     2 AD32 .    MemRd   08FF89B8  ........ TdwodTr  --  ----  1111
   -49343:    179.4ns     2 AD32 .    MemRd   08FF89B8  33323130 OK       --  ----  1111
   -49342:    867.1ns     2 AD32 .    MemRd   08FF89BC  ........ TdwodTr  --  ----  1111
   -49341:    149.5ns     2 AD32 .    MemRd   08FF89BC  ........ TdwodTr  --  ----  1111
   -49340:    149.5ns     2 AD32 .    MemRd   08FF89BC  ........ TdwodTr  --  ----  1111
   -49339:    149.5ns     2 AD32 .    MemRd   08FF89BC  ........ TdwodTr  --  ----  1111
   -49338:    149.5ns     2 AD32 .    MemRd   08FF89BC  ........ TdwodTr  --  ----  1111
   -49337:    179.4ns     2 AD32 .    MemRd   08FF89BC  37363534 OK       --  ----  1111
   -49336:    837.2ns     2 AD32 .    MemRd   08FF89C0  ........ TdwodTr  --  ----  1111
   -49335:    149.5ns     2 AD32 .    MemRd   08FF89C0  ........ TdwodTr  --  ----  1111
   -49334:    149.5ns     2 AD32 .    MemRd   08FF89C0  ........ TdwodTr  --  ----  1111
   -49333:    149.5ns     2 AD32 .    MemRd   08FF89C0  ........ TdwodTr  --  ----  1111
```

Figure 8, Timing on reads from the non-system board to the system board.

Each address that will result in a 4 byte transfer from system memory on the system board to the non-system boards, will also take a long time to complete. The first transfer shown in Figure 8 takes as much as 956,8ns + 149,5ns + 149,5ns + 149,5ns + 149,5ns 179,4 = **1734,2**ns. This represents 1743,2ns / 29,9ns = **58** PCI clock cycles. During this time the processor is idle which will result in a waste of thousands of possible instructions executing on the receiver.



```
-49183:   119.6ns   1 AD32 .     MemWri  08FF0CC0  39383736 |OK     --  ----  1111
-49182:   149.5ns   1 AD32 Start MemWri  08FF0CC4  33323130 |OK     --  ----  1111
-49181:    29.9ns   . AD32 B     MemWri  08FF0CC8  37363534 |OK     --  ----  1111
-49180:    29.9ns   . AD32 B     MemWri  08FF0CCC  31303938 |OK     --  ----  1111
-49179:   119.6ns   1 AD32 .     MemWri  08FF0CD0  35343332 |OK     --  ----  1111
-49178:   119.6ns   1 AD32 Start MemWri  08FF0CD4  39383736 |OK     --  ----  1111
-49177:    29.9ns   . AD32 B     MemWri  08FF0CD8  33323130 |OK     --  ----  1111
-49176:    29.9ns   . AD32 B     MemWri  08FF0CDC  37363534 |OK     --  ----  1111
-49175:   119.6ns   1 AD32 .     MemWri  08FF0CE0  31303938 |OK     --  ----  1111
-49174:   149.5ns   1 AD32 Start MemWri  08FF0CE4  35343332 |OK     --  ----  1111
-49173:    29.9ns   . AD32 B     MemWri  08FF0CE8  39383736 |OK     --  ----  1111
-49172:    29.9ns   . AD32 B     MemWri  08FF0CEC  33323130 |OK     --  ----  1111
-49171:   119.6ns   1 AD32 .     MemWri  08FF0CF0  37363534 |OK     --  ----  1111
```

Figure 9, Impact of Writes over the CPCI-bus and the bus hierarchy.

Figure 9 above shows the case of the non-system board making writes over the CPCI-bus into the shared memory area on the system board. Timing is much better, but not optimal. The first write marked in the figure takes 149,5ns (five PCI-clock cycles) and the next, due to burst effects, allocates only 29,9ns or one PCI-bus cycle. The question is why the burst effect is interrupted. I have no answer today. The conclusion of these figures is of course:

> *Always perform writes from the sender instead of reads from the receiver in a PCI-bus hierarchy.*

This is a well-known fact, but the tests also show the necessity of identifying and optimizing PCI-bus transactions. One badly executed instruction in a message area transparently transferred over the PCI-bus hierarchy can waste many precious clock cycles in the processor. The correct utilization of bursts must be exploited as well as correct use of the 256 bytes of posted write and 256 bytes read buffers in each direction in the Intel 21554 PCI-PCI bridge. But buffering will not be better in a bridge hierarchy than its weakest link. Having only one bridge in the hierarchy with few buffers will degrade performance in the whole chain.

6.1 Message Passing with only Writes over the PCI-bus

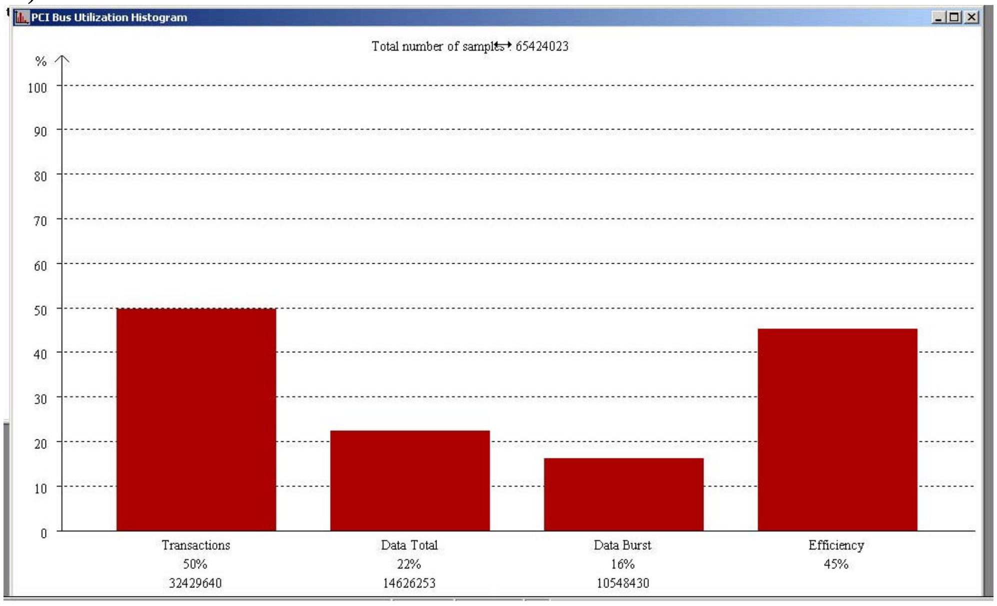
SimpleMessageQueue was modified to support message passing with only writes into the receiving memory area and the timing and performance to message-passing was highly improved as can be seen in the figures in Figure 10.

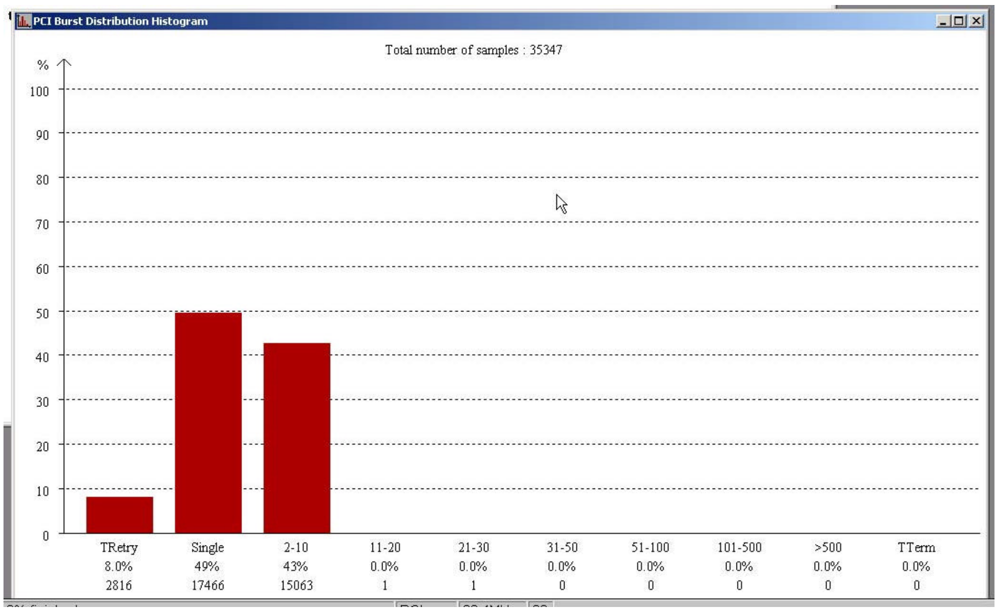
Message Size	Messages per second and direction	Load on individual threads or modules on System Board	Load on individual threads or modules on Non-System Boards
1000 byte	14450 ± 50	SendReceive 43% Bkgnd 48% Kernel 2% Int 6%	SendReceive 45% Bkgnd 46% Kernel 2% Int 6%
100 byte	36750 ± 50	SendReceive 31% Bkgnd 46% Kernel 5% Int 16%	SendReceive 26% Bkgnd 50% Kernel 5% Int 16%
10 byte	43150 ± 50	SendReceive 27% Bkgnd 47% Kernel 6% Int 18%	SendReceive 23% Bkgnd 53% Kernel 6% Int 17%
0 byte	45400 ± 100	SendReceive 25% Bkgnd 48% Kernel 7% Int 19%	SendReceive 23% Bkgnd 50% Kernel 7% Int 19%

Figure 10, SimpleMessageQueue with only writes over the PCI-bus.

The throughput reaches its maximum when 1000 byte big messages are sent. In that case 14450 messages * 1000 byte ≈ 14,5 MB/s in each direction is sent. The theoretical maximum throughput of the PCI-bus is 32bit * 33MHz ≈ 132MB/s which is much better that the figure presented by SimpleMessageQueue (14,5MB/s * 2 = 29 MB/s). SimpleMessageQueue is thus approximately 132/29 ≈ 4,5 times slower.

Performance results from the VMETRO PCI-bus analyzer are as shown in the following figures (all showing a snapshot of communication with a message size of 1000 bytes).





7 References

[Shanley99] Tom Shanley, Don Andersson, "*PCI System Architecture*", Third Edition, Mindshare, Addison Wesley Longman Inc., ISBN 0-201-30974-2, 1999.

[SBS] SBS Technologies Inc., http://www.sbs.com/