SCRAMNet GT

GT Memory Access: PIO vs. DMA

Abstract

When transferring data to and from a data store, selection of the most efficient and timely method for doing so is often critical. Selection of an under-performing method can have an adverse impact on what would otherwise be a high performance system. Therefore, when presented with a choice of methods, it is important to understand the characteristics that each provides so that the correct selection(s) can be made. This paper discusses the strengths, limitations, and performance characteristics of the methods for accessing GT memory to help ensure that the correct transfer method is selected for the target system.

Introduction

Management of data in GT memory can be accomplished using either of two transfer methods. These methods are Programmed Input/Output (PIO) and Direct Memory Access (DMA).

Identical operations on GT memory can be achieved with PIO and DMA. Still, it is important to select the method that meets application performance demands. Each transfer method has its own strengths. It is not necessary, however, for a program or system to use only one transfer method, as the GT hardware supports simultaneous PIO and DMA data transfers.

PIO Access to GT Memory

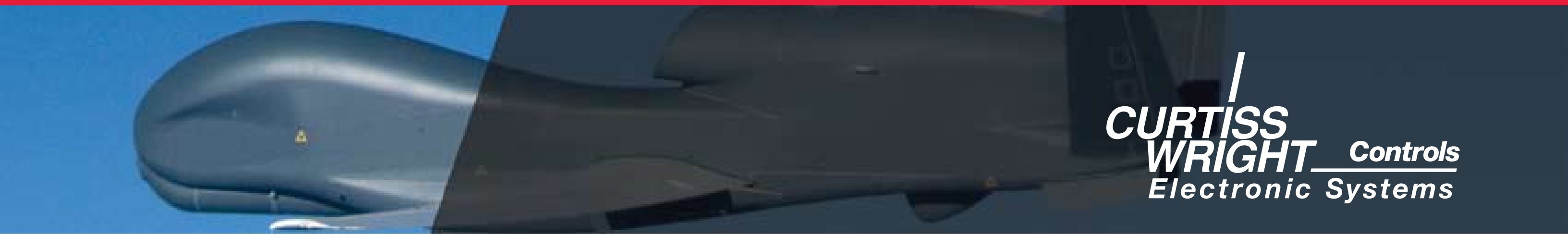
The PIO transfer method allows GT memory to be written and read similarly to system memory. During PIO transfers, the CPU initiates the movement of data over the PCI bus, between host memory and GT memory. This means that a program can read or write GT memory simply by executing a CPU instruction that dereferences an address within the GT memory address space. This also means there is no API, driver, or interrupt overhead associated with PIO transfers, as there is with DMA. This lack of overhead typically makes PIO more efficient than DMA on small data transfers.

GT memory can also be read or written in 8-bit, 16-bit, and 32-bit words as an atomic operation using PIO. Atomic operation is not possible on sub-32-bit words using DMA, for which sub-32-bit granularity is not available. To perform an 8-bit modification with DMA, one must read a 32-bit word, modify the correct byte in the word, and then write the word back with DMA.

Since PIO allows the CPU to access GT memory directly as if it were system memory, this gives the capability of transferring data to/from the GT device without also having the data in system memory. This system-memory-like access is achieved with PIO by using device memory locations as the operands in computations. For example, let pGT be a C-code pointer to some location in GT memory. The following C-code instruction would perform two read accesses and one write access to GT memory:

```
pGT[2] = pGT[0] + pGT[1];
```

CURTISS WRIGHT Controls
Electronic Systems



However, this type of access is only recommended for somewhat trivial situations, such as updating control information. High-speed processing of data in system memory, after retrieval from GT memory, is more appropriate for data processing situations. GT memory is not cached by the CPU, as system memory typically is. Also, if processing involves modification and re-use of information in the data set, write latency to GT memory caused by network propagation and insertion times can cause read-back of old data. Additionally, GT network bandwidth is shared by all nodes, and writes to GT memory generate network traffic in most configurations. Therefore, processing data in system memory prevents unnecessary network traffic from impeding other nodes.

Lastly, the PIO capability (PCI slave interface) of GT memory allows it to be accessed by any PCI bus master. For example, the DMA engine on a second PCI network device in the system could read or write its data directly to/from GT memory, eliminating copying to and from system memory, and processing steps that may otherwise be required to complete the same objective.

Summary of PIO access characteristics:

- No API, driver, or interrupt overhead
- CPU (a PCI bus master) reads and writes data
- One byte transfer granularity
- System-memory-like access

DMA Access to GT Memory

The DMA transfer method allows GT memory to be written and read in contiguous blocks, and produces results equivalent to a memory copy. During DMA transfers, the GT hardware initiates the movement of data over the PCI bus, between host memory (or other PCI-visible memory) and GT memory. GT hardware transfers data directly to or from an application provided buffer. The data is not first copied by the driver into a driver/kernel memory buffer.

The GT hardware has two DMA engines; one executes read requests and the other write requests. The GT driver synchronizes access to the GT hardware and DMA engines, serializing the execution of DMA transfer requests in each engine. No more than one DMA request will be in progress in a DMA engine at any given time, but a write DMA and a read DMA can be in progress concurrently. PIO accesses to GT memory can be initiated, accepted, and completed by the GT hardware during DMA transfers. The PIO accesses are independent of the DMA engine and are not synchronized or serialized with DMA engine transfers.

The GT DMA engines have 4-byte (32-bit) granularity. As such, transfer sizes must always be a multiple of four bytes, and the system memory and GT memory addresses must be aligned to natural 32-bit word boundaries.

In general, system throughput will be maximized when data is transferred in large blocks using DMA, as there is API, driver, and interrupt overhead associated with initiating and completing DMA transfers. As transfer size decreases, this overhead has an increasing impact on throughput. Applications that transfer small blocks of data can eliminate DMA overhead by using PIO, and may thereby achieve higher throughput with PIO.

During DMA transfers, the CPU is available to perform other system tasks, such as executing other processes. A single application can take advantage of the CPU availability by using multiple threads. For example, thread A may be dedicated to retrieving data from GT memory with DMA. The data will be stored in one of two system memory buffers. While one buffer is updated, thread B can process the data in the other buffer. This would streamline the application by using processing time more efficiently.

Summary of DMA access characteristics:

- ♦ GT hardware performs data transfer
- ♦ CPU free for other system tasks
- ♦ API and driver overhead
- ♦ Four byte transfer granularity
- ♦ Serialized and synchronized by driver and hardware

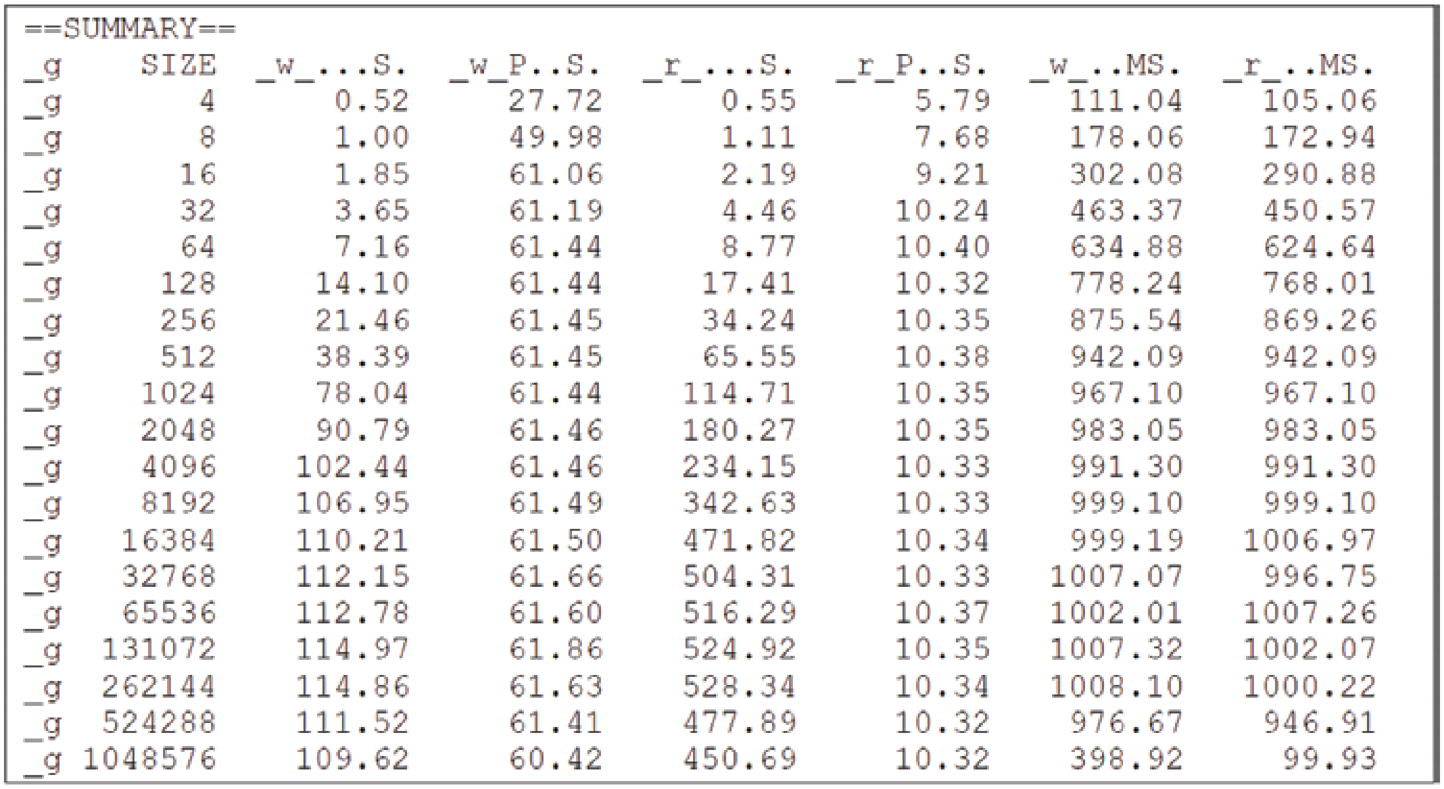
Analysis with the gttp Utility

An intelligent selection of GT memory access method can be made using the characteristics given above. However, each system and design can still benefit from some analysis. The **gttp** application provided with the GT software distribution can assist in performance analysis of the GT within the target system. The **gttp** throughput graphing option(s) **-GA**, can be used to collect textual performance numbers for varying transfer sizes. These numbers provide a good starting point for evaluating system performance. A case study follows.

Table 1 shows the GT throughput summary report[1] obtained with command '**gttp –GA**' on a computing system hereafter referred to as SystemA The throughput numbers are in MB/s (106 bytes/second), and were collected with minimal system load from other applications and no other GT network traffic. The column headers are interpreted as follows (if neither P nor M are present, then DMA access was used):

SIZE	contiguous and continuous transfer buffer size in bytes
w	indicates write throughput to GT memory
r	indicates read throughput from GT memory
P	indicates PIO access (32-bit accesses)
M	indicates system memory throughput (GT memory not involved)
S	indicates the test repeatedly transferred the buffer for 1 second

Table 1: A gttp throughput summary



```
==SUMMARY==
_g    SIZE   _w_...S.  _w_P..S.  _r_...S.  _r_P..S.  _w_..MS.  _r_..MS.
_g       4      0.52     27.72      0.55      5.79    111.04    105.06
_g       8      1.00     49.98      1.11      7.68    178.06    172.94
_g      16      1.85     61.06      2.19      9.21    302.08    290.88
_g      32      3.65     61.19      4.46     10.24    463.37    450.57
_g      64      7.16     61.44      8.77     10.40    634.88    624.64
_g     128     14.10     61.44     17.41     10.32    778.24    768.01
_g     256     21.46     61.45     34.24     10.35    875.54    869.26
_g     512     38.39     61.45     65.55     10.38    942.09    942.09
_g    1024     78.04     61.44    114.71     10.35    967.10    967.10
_g    2048     90.79     61.46    180.27     10.35    983.05    983.05
_g    4096    102.44     61.46    234.15     10.33    991.30    991.30
_g    8192    106.95     61.49    342.63     10.33    999.10    999.10
_g   16384    110.21     61.50    471.82     10.34    999.19   1006.97
_g   32768    112.15     61.66    504.31     10.33   1007.07    996.75
_g   65536    112.78     61.60    516.29     10.37   1002.01   1007.26
_g  131072    114.97     61.86    524.92     10.35   1007.32   1002.07
_g  262144    114.86     61.63    528.34     10.34   1008.10   1000.22
_g  524288    111.52     61.41    477.89     10.32    976.67    946.91
_g 1048576    109.62     60.42    450.69     10.32    398.92     99.93
```

[1] The throughput numbers presented in this document are provided for illustrative purposes, and are valid for the stated computing system. Throughput results vary from system to system.

The following graphs were generated using the GT memory throughput numbers listed in Table 1 for SystemA.

Figure 1: A graphical representation of Table 1

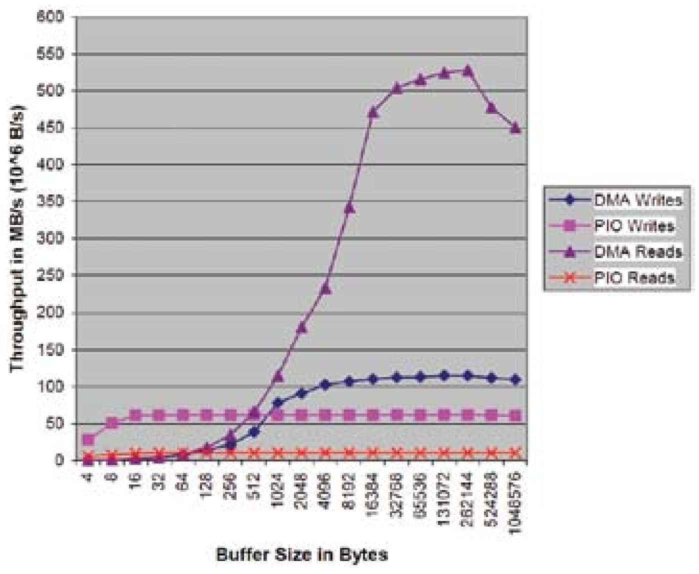
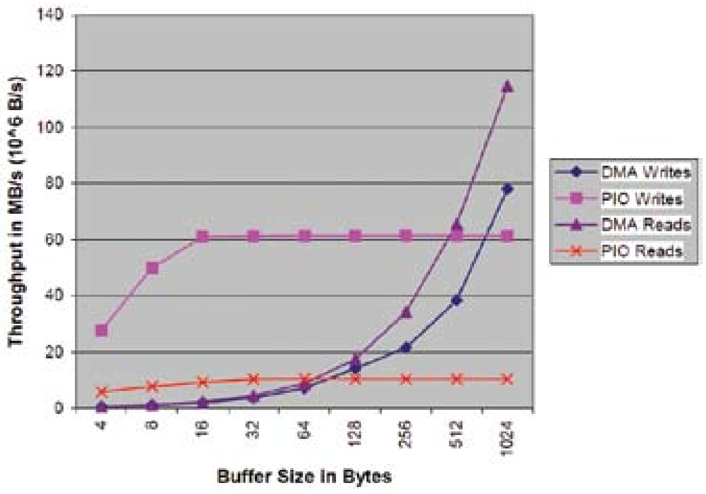


Figure 2: A graphical representation of Table 1, zoomed to DMA vs. PIO performance thresholds



The throughput numbers provide useful information regarding DMA and PIO performance on this computer. Let the buffer sizes at which DMA access outperforms PIO access be referred to as the DMA thresholds. Here, the write DMA threshold is 1024 bytes, and the read DMA threshold is 128 bytes. So, when accessing GT memory under conditions similar to those when the throughput numbers were taken, the access method can be selected based on the read and write DMA thresholds. PIO can be efficiently used for buffer sizes below the respective DMA threshold, and DMA can be efficiently used for buffer sized above the respective DMA threshold.

Numbers such as those presented above are helpful in selecting the correct access method for use in an application. However, they provide only a snapshot of performance capabilities under low system load conditions. Many factors can have considerable impact on overall performance. Such factors include CPU load, PCI bus performance, memory speed, GT network load, and the timing of events within the system. Analysis of performance under expected operating conditions is recommended.

Additional instances of gttp (or other GT utility applications) can be executed to simulate expected operating conditions. For example, executing an additional gttp instance on the target computer will impose resource sharing and addition CPU and memory load. In addition, writing to GT shared memory with gttp from remote nodes will impose network bandwidth sharing.

An Application with Potential

Assume that a shared memory application named gfixit utilizes 16 small contiguously located 32-byte segments of GT memory for sharing status and control information among 16 nodes on the GT network. Each node is responsible for updating the 32-byte segment containing its control and status information. Let each GT node be identical and have performance numbers equal to those presented in the previous section, notably a DMA read threshold of 128 bytes.

Gfixit has a main execution loop that checks the status of each other node, one by one, by reading their control/status segments. If the node has new data available, then the data is retrieved and processed. The following is pseudo-C-code that implements this.

```
int readData( int length, ... )
{
    if ( length >= 128 )
    {
    /* read data from GT memory with
    DMA */
    }
    else
    {
    /* read data from GT memory using
    PIO */
    }
    return 0;
}
int main(int argc, char **argv)
{
    int i;
    while ( 1 )
    {
       for( i=0; i < 16; i++ )
       {
          if ( i != /* my status and
          control index */ )
          {
             readData (32, ... ); /*
             read control/status
             info for node 'i'.
             This will use PIO. */
             /* process control/status
             info */
          }
          else
          {
             /* update my control/
             status info in GT memory*/
          }
       }
    }
    return 0;
}
```

In the simple example above, control data is retrieved from the other nodes, one by one using PIO. The application does not retrieve its own data, but instead refreshes its data in GT memory. Functionally, the application works, but it is performing a little more sluggishly than the designer would like. Fortunately, a modification can be made that will boost performance.

PIO was selected to transfer the 32-byte control segments because it performs more efficiently for 32-byte transfers (10.24 MB/s with PIO vs. 4.46 MB/s with DMA). PIO was the correct choice for the current design. The total time required to transfer the 15 32-byte segments with the current code is:

$$\frac{\text{32 bytes/node x 15 nodes}}{\text{10.24e6 bytes/second}} \simeq 0.000046875 \text{ sec} = 46.875\mu\text{sec}$$

Notice, that the 32-byte control segments are stored contiguously in shared memory. In general, throughput performance improves as the transfer size increases. Instead of reading the control segments one by one, they can be read all at once. Of course, each node will read its own control data, but this simplifies the code. This also permits the data to be transferred in one large transfer, instead of two smaller ones[2]. Reading our own control data is harmless, aside from adding a little overhead to the transfer. The following is the modified pseudo-C-code.

[2] The nodes having the first and last indices require only one transfer even if they don't read their own control data. This is ignored in the interest of simplicity and commonality.

```
int main(int argc, char **argv)
{
    int i;
    while ( 1 )
    {
       readData (32*16, ... ); /* read
       control/status info for
       all nodes. This will use DMA.
       */
       for( i=0; i < 16; i++ )
       {
          if ( i != /* my status and
          control index */ )
          {
             /* process control/status
             info */
          }
          else
          {
             /* update my control/
             status info in GT memory*/
          }
       }
    }
    return 0;
}
```

DMA was selected to transfer the control segments in the new code because the total number of bytes read at once is now 32 x 16 = 512 bytes. DMA performs more efficiently for 512-byte transfers (65.55 MB/s with DMA vs. 10.38 MB/s with PIO). The total time required to transfer the 512-byte segment with the new code is:

$$\frac{512 \text{ bytes}}{65.55\text{e}6 \text{ bytes/second}} \sim 0.000007811 \text{ sec} = 7.811 \mu\text{sec}$$

The new code transfers the data approximately 6 times faster, even while unnecessarily retrieving its own 32 control bytes.

Conclusion

PIO and DMA access to GT memory each offer unique characteristics, and finding the correct mix of DMA and PIO access can improve the performance of an application, and the efficiency of the entire system. Many factors can affect application performance, so in any case, a design will benefit from some analysis.