Using the Cray Gemini Performance Counters

Kevin Pedretti, Courtenay Vaughan, Richard Barrett, Karen Devine, K. Scott Hemmert
Sandia National Laboratories
Albuquerque, NM 87185
Email: {ktpedre, ctvaugh, rfbarre, kddevin, kshemme}@sandia.gov

***Abstract*—This paper describes our experience using the Cray Gemini performance counters to gain insight into the network resources being used by applications. The Gemini chip consists of two network interfaces and a common router core, each providing an extensive set of performance counters. Based on our experience, we have found some of these counters to be more enlightening than others. More importantly, we have performed a set of controlled experiments to better understand what the counters are actually measuring. These experiments led to several surprises, described in this paper. This supplements the documentation provided by Cray and is essential information for anybody wishing to make use of the Gemini performance counters. We demonstrate the use of the Gemini tile performance counters to quantify the reduction in network congestion due to an improved process mapping scheme for the MiniGhost miniapp.**

I. Introduction

The Cray Gemini [1] network processing unit provides a large set of hardware performance counters that have many potential uses, including understanding individual application communication behavior and understanding the behavior of the system as a whole (e.g., network congestion). This is a unique capability compared to most commodity networks. Unfortunately, when we had a need to use these counters, we quickly concluded that doing so would not be "easy". The available documentation [2] provided basic descriptions of the counters but left out many necessary details, and provided no examples showing how to directly access the counters without using CrayPat. Our hope in writing this paper is to fill in some of the missing gaps to jump start others who want to make use of the Gemini counters.

The remainder of this paper is organized as follows: Section II describes our approach for directly accessing Gemini performance counter information from MPI applications. Section III and IV describe how to derive logical link counters and gather job-wide counter information, respectively. Results of experiments designed to understand the operation of the counters are described in Section V, followed by a demonstration of using the counters to measure network congestion in Section VI. Future work is discussed in Section VII and Section VIII concludes the paper.

II. Directly Accessing the Gemini Tile Counters

Our first step was to gain direct access to the Gemini counters from application code. At the time, CrayPat [3] had the ability to access Gemini network interface counters but had no support for accessing the Gemini router tile counters. Since our main interest was in the tile counters and we wanted to avoid the extra steps needed to use CrayPat, we began looking for other options.

Fortunately our search was not long. While looking through the Gemini Linux kernel driver source code,[1] we found the source code for the Gemini Performance Counter Daemon (GPCD) kernel module and some example code showing how to access its interface from user-level. This is presumably the same interface that CrayPat uses. We converted the sample user-level code into a library, wrote a test program, and verified that we could access the tile counters.

Figure 1 shows an example of using the GPCD library. The first step is to create a counter context that includes the six fixed counters for each tile (lines 8 through 18). The code uses the counter's human readable name (line 14) to look up the memory mapped register descriptor for the counter (line 16). The descriptor is then added to the context (line 17). Overall, 288 counters are added to the context, since there are 48 tiles and each tile has six fixed counters. The next step is to read the counter values into the context (line 21). Internally, this makes an ioctl() call to the GPCD kernel module, the kernel module reads the counter values specified in the context, and then returns the values to user space. The counter values are stored in the context itself. The last step is to print the counter values by walking the descriptor list stored in the context (lines 24-26).

The six per-tile fixed counters are briefly mentioned in Cray's documentation [2]. However, the mapping between the counter names in the documentation and the GPCD library counter names is not specified. We performed a set of simple experiments similar to those described in Section V to determine the mapping is as shown in Table I. The table also gives a brief description of each tile counter.

III. Aggregating Tile Counters to Link Counters

Our next step was to figure out how to aggregate the individual tile counters into logical link counters. For our purposes the router tiles were just an implementation detail– what we really cared about were the network links connecting the Geminis together in the 3-D torus and the host link connecting the Gemini to its two local hosts. Each of these logical links is made up of a set of tiles.

We were not able to find a way to determine the tile to link mapping from a login node or compute node. Instead, we used Cray's rtr tool that is available on the Service Management Workstation (SMW) to dump out the mapping to a text file:

```
rtr --interconnect > interconnect.txt
```

[1] For the Cray 4.0.36 release, this source code is available in cray-gni-1.0-1.0400.4123.8.8.gem.src.rpm and is licensed under GPLv2.

TABLE I: Mapping of tile counter names used in Cray's Gemini documentation [2] to the names used in the GPCD interface.

Cray Documentation Name	GPCD Library Name	Description
GM_TILE_PERF_VC0_PHIT_CNT:n:m	GM_n_m_TILE_PERFORMANCE_COUNTERS_0	Request virtual channel phit count.
GM_TILE_PERF_VC1_PHIT_CNT:n:m	GM_n_m_TILE_PERFORMANCE_COUNTERS_1	Response virtual channel phit count (phit = 3 bytes)
GM_TILE_PERF_VC0_PKT_CNT:n:m	GM_n_m_TILE_PERFORMANCE_COUNTERS_2	Request virtual channel packet count
GM_TILE_PERF_VC1_PKT_CNT:n:m	GM_n_m_TILE_PERFORMANCE_COUNTERS_3	Response virtual channel packet count (packet = 8 to 32 phits)
GM_TILE_PERF_INQ_STALL:n:m	GM_n_m_TILE_PERFORMANCE_COUNTERS_4	Count of input stall cycles
GM_TILE_PERF_CREDIT_STALL:n:m	GM_n_m_TILE_PERFORMANCE_COUNTERS_5	Count of output stall cycles (router operates at 800 MHz)

```
1  gpcd_context_t *ctx;
2  gpcd_mmr_desc_t *desc;
3  gpcd_mmr_list_t *p;
4  int i, j, k;
5  char name[128];
6
7  // Create a counter group, all tile counters
8  ctx = gpcd_create_context();
9  for (i = 0; i < 6; i++)      // TILE_ROWS
10   for (j = 0; j < 8; j++)    // TILE_COLS
11     for (k = 0; k < 6; k++) // TILE_COUNTERS
12     {
13       sprintf(name,
14         "GM_%d_%d_TILE_PERFORMANCE_COUNTERS_%d",
15         i, j, k);
16       desc = gpcd_lookup_mmr_byname(name);
17       gpcd_context_add_mmr(ctx, desc);
18     }
19
20 // Sample the tile counters
21 gpcd_context_read_mmr_vals(ctx);
22
23 // Print the counter values
24 for (p = ctx->list; p; p = p->next)
25   printf("Counter %s: Value=%lu\n",
26          p->item->name, p->value);
```

Fig. 1: Example user-level code showing how to sample and print the six static counters from each of the Gemini's 48 router tiles using the native Gemini Performance Counter Daemon interface.

Fig. 3: Gemini tile to logical link mapping for the first node defined in the `interconnect.txt` for Cielo. Links in the X dimension are shown in orange, Y links in green, Z links in blue, and host links in yellow.

This command requires administrative privileges to run. We then copied the text file to a login node so that it was available to our tools.

The rtr tool output contains one line per source tile to destination tile connection (i.e., a connection between two Gemini chips). Figure 2 shows an example of the tool's output. Each line includes the direction of the link (X+, X-, Y+, Y-, Z+, Z-), the 3-D coordinate of the source and destination Gemini, and the type of link (backplane, mezzanine, or cable). Only information for the 40 network link tiles per Gemini is included in the output. The eight tiles per Gemini that are not included make up the host link that connects the Gemini to its two hosts. For the 9216 node DOE/NNSA Cielo XE6 system, the `interconnect.txt` file is about 14 MB and contains 184,320 lines (40 tiles * 4,608 Geminis = 184,320 lines).

Figure 3 graphically shows an example tile to logical link mapping for a node in Cielo. Other large XE and XK systems will use a similar mapping, with eight tiles being assigned to links in the X and Z dimensions and four tiles being assigned to Y dimension links. In general, it is not possible to assume that the mapping is identical on all nodes of a given system because, at a minimum, the direction of a given tile will be different on the source and destination Geminis. For example, in Figure 3 tile 0 is assigned to the Z+ link. This tile connects to tile 0 on the destination Gemini, and is therefore associated with the Z- link. The link type assigned to each Gemini also changes from node to node, depending on the Gemini's position in the 3-D torus. Our tools parse the full `interconnect.txt` file so these differences are automatically picked up without having to make any assumptions.

Table II lists the bandwidth of each link type. This is useful for calculating the aggregate bandwidth of a logical link. For example, in Figure 3 there are eight tiles making up the X+ logical link, each using a cable link. The aggregate bandwidth of the link is therefore $8*1.17 = 9.4$ GBytes/s (uni-directional, 18.8 GB/s bi-directional). For large XE and XK systems, all X links are cable based with bandwidth 9.4 GB/s. Links in the Y dimension alternate every other between mezzanine (within a board) and cable (between boards) links, with bandwidths $4 * 2.34 = 9.4$ GB/s and $4 * 1.17 = 4.7$ GB/s, respectively. Links in the Z dimension are mostly backplane links (within a cage) with bandwidth $8 * 1.88 = 15$ GB/s. Every eighth link in the Z dimension is a cable link (between cages) with

```
c0-0c0s0g0l00[(0,0,0)] Z+ -> c0-0c0s1g0l32[(0,0,1)]    LinkType: backplane
c0-0c0s0g0l01[(0,0,0)] Z+ -> c0-0c0s1g0l21[(0,0,1)]    LinkType: backplane
c0-0c0s0g0l02[(0,0,0)] X+ -> c1-0c0s0g0l02[(1,0,0)]    LinkType: cable11x
c0-0c0s0g0l03[(0,0,0)] X+ -> c1-0c0s0g0l03[(1,0,0)]    LinkType: cable11x
c0-0c0s0g0l04[(0,0,0)] X- -> c2-0c0s0g0l41[(15,0,0)]   LinkType: cable18x
c0-0c0s0g0l05[(0,0,0)] X- -> c2-0c0s0g0l31[(15,0,0)]   LinkType: cable18x
c0-0c0s0g0l06[(0,0,0)] Z- -> c0-0c2s7g0l26[(0,0,23)]   LinkType: cable15z
c0-0c0s0g0l07[(0,0,0)] Z- -> c0-0c2s7g0l35[(0,0,23)]   LinkType: cable15z
```

Fig. 2: First 8 lines of the `interconnect.txt` file for Cielo (16x12x24 topology).

TABLE II: Tile link type to bandwidth conversions.

Link Type	Bandwidth
Mezzanine	2.34 GB/s
Backplane	1.88 GB/s
Cable	1.17 GB/s
Host	1.33 GB/s (est.)

bandwidth $8 * 1.17 = 9.4$ GB/s. As can be seen, there is a fairly heterogeneous distribution of interconnect link speeds on Gemini based systems.

IV. Gathering Job-wide Information

Once we understood how to aggregate the tile counters into logical links, our next step was to develop a higher-level MPI library to aggregate the counter information across an entire application run. This library, which we named the Gemini monitor library (libgm.a), gets linked with the target application to be profiled and provides a simple API for sampling and printing application-wide counter information. Currently there are three API calls:

```
// Initialize the library
gemini_init_state(comm, &state)

// Sample the gemini counters
gemini_read_counters(comm, &state)

// Output delta of last two samples
gemini_print_counters(comm, &state)
```

A typical usage scenario would be to surround a code region of interest with calls `gemini_read_counters()` and then call `gemini_print_counters()` to output the counter difference from the start to the end of the region.

Internally, the library forms an MPI communicator with one process per Gemini, which we call the leader process. Reading the counters from multiple processes per Gemini would be redundant and add unnecessary overhead. Each Gemini leader process parses the `interconnect.txt` information (rank 0 broadcasts the file to all Gemini leader processes) to find its own tile to logical link mapping. No information about other Geminis is retained. When an application requests that the counters be sampled, the individual tile counters are aggregated to logical link counters, as described in Section III. Each Gemini leader stores multiple logical link counter samples. When the application requests that the counters be output, each Gemini leader calculates the delta between the last two counter samples and sends the result to rank 0 using an `MPI_Gather()` collective. Rank 0 then outputs the aggregated counter information to a text file.

Figure 4 shows an example of the library's output. The first two comment lines are included for convenience, and are not normally included in the output. Each Gemini block begins with a line indicating the Gemini's coordinate in the 3-D torus and is followed by up to seven lines, one per logical link (including the host link, HH). Each logical link line includes the direction of the link, the coordinate of the Gemini at the remote end of the link, the speed of the link in Gbytes/s, and the values of the six fixed tile counters for the link (the delta between the last two samples). To allow for maximum flexibility for post processing, we chose to output the full six counters for each of the seven logical links on each Gemini, rather than trying to condense or summarize the information within the library.

We have tested the library on up to 131,072 process runs, comprised of 8,192 nodes running 16 processes per node. We found that the Gemini leader process communicator size is typically slightly larger than half the number of nodes for a given run, due to the Cray node allocator sometimes only allocating one of a Gemini's two nodes to the job (e.g., for all of the 131,072 process runs, the same 4,118 Geminis were used). The memory overhead of the library on non-rank 0 processes is very small, since only local information is stored. The rank 0 process requires some buffer space to gather information from all Gemini leader processes. This overhead grows linearly with the number of Geminis and is approximately 6 MBytes for a hypothetical system with 10,000 Geminis.

V. Sonar Experiments

We performed a number of simple tests to better understand the operation of the Gemini tile counters. The basic theme of these experiments was to send out a known "ping", such as a single MPI message of known size, and then inspect the tile counters to determine what happened – the "pong" echo. These experiments gave us a clear picture of what was actually transmitted on the wire for put and get transactions, revealed the directionality of the tile counters, helped us understand the routing scheme used on our system, and allowed us to measure the bandwidth efficiency of the Gemini network for MPI point-to-point messages. These topics are discussed in the following sections.

```
# DEST_COORD
#    SRC_COORD  GB/s   VC0_PHITS  VC1_PHITS VC0_PKTS  VC1_PKTS   INQ_STALLS OUTQ_STALLS

( 0, 1, 1 )
 X+ ( 1, 1, 1)  9.38   1626452284 304999266 101662806 101666422  3533598961  2689080952
 X- (15, 1, 1)  9.38       100506     38796      9780     12932          83           0
 Y+ ( 0, 2, 1)  4.69   1627257610 305156760 101726643 101718920  1702270109           0
 Y- ( 0, 0, 1)  9.38   1153554135 216313236  72105559  72104412  1925883229  2366983378
 Z+ ( 0, 1, 2) 15.00    815234359 152948952  50988260  50982984   133047991   776502961
 Z- ( 0, 1, 0) 15.00      1743043    378399    156635    126133         580   992022669
 HH ( 0, 1, 1) 10.40   1834489368 344019696 114672167 114673232 10585723107  2263990777

( 0, 0, 1 )
 X+ ( 1, 0, 1)  9.38   1966685020 368797209 122929393 122932403  3317929506  3063532486
 X- (15, 0, 1)  9.38       122194     43005     11983     14335           9           0
 Y+ ( 0, 1, 1)  9.38   1154016206 216417552  72141025  72139184  3589170400  1097189607
 Y- ( 0,11, 1)  4.69        96911     20538      9646      6846       56244           0
 Z+ ( 0, 0, 2) 15.00   2477453033 458007486 153779007 152669162   952487628  2209098748
 Z- ( 0, 0, 0) 15.00      2071415   3684912    128723   1228304      464902   387186094
 HH ( 0, 0, 1) 10.40   2174662127 407809092 135934105 135936364 10604254673  2216827070
```

Fig. 4: Example output of the Gemini Monitor library for two Geminis, (0, 1, 1) and (0, 0, 1).

A. Put and Get Transactions

As described in the Cray's Gemini white paper [4], every network transaction is a single "request" packet from source to destination, followed by a single "response" packet from destination to source. The maximum transaction contains 64 bytes of user data payload. Everything is built on top of these small transactions. For example, a large MPI message will get broken down into many individual 64 byte transactions. Request packets always are sent on virtual channel 0 (VC0) and response packets are always sent on virtual channel 1 (VC1).

A typical PUT transaction that sends 64 bytes of data from a source to a destination consists of a 32 phit request packet (96 bytes) followed by a 3 phit response packet (9 bytes) from destination to source. The total traffic on the network is $96 + 9 = 105$ bytes. A typical GET transaction requesting 64 bytes of data from a remote node consists of a 8 phit request packet (24 bytes) followed by a 27 phit response packet (81 bytes). Total traffic on the network is $24 + 81 = 105$ bytes, which is the same as for PUT transactions.

We performed a series of experiments to confirm that the tile phit counters were correctly measuring the expected values for PUT and GET transactions, as just described. Our original test consisted of sending a 1 MB message from a source to a destination, sampling the tile counters before and after to calculate the delta. The results we got were confusing at first. While the packet counts were consistent with a PUT based protocol, the phit counters were much too low, by about a factor of three.

After discussing the issue with Cray, it turns out that the Gemini compresses packets with runs of zero bits or runs of one bits. Our benchmark was sending a zeroed message, leading to the confusing phit counts. After we initialized our test message to a random bit pattern, the phit counters measured the expected values. This compression is something to be aware of when using the Gemini's tile counters.

B. Tile Counter Directionality

The documentation provided by Cray did not discuss the directionality of the tile counters. It was not clear if the phit and packet counters measured incoming or outgoing transmissions, or whether the directionality of the VC0 and VC1 counters were the same. Our experiments revealed that the VC0 and VC1 counters both measured packets and phits coming into the Gemini tile (i.e., they measure packets and phits that arrive at the destination end of a network link). This means that the traffic flowing into a Gemini router can be determined with only local information. Remote tile counter information from neighboring Geminis is needed to understand the traffic flowing out of a given Gemini.

Like the packet and phit counters, the input stall counter measures stall cycles for incoming packets. These stall cycles occur when a packet is not able to move to the destination tile's output queue, due to lack of credits or some other Gemini-internal resource being unavailable. The output stall counter measures stall cycles for outgoing packets. This occurs when a tile's output queue does not have enough credits for the destination Gemini tile's input queue.

C. Routing

Gemini based systems use a static routing scheme where all traffic from a given source to a given destination follows the same path through the 3-D torus interconnect. While this is obvious in retrospect, our experiments revealed that the path taken by a given transaction's request packet is, in general, different from the path taken by the response packet. The request packet will use the static route from the source to the destination. The response packet will return via the static route from the destination to the source. It is important to be aware of this since PUT ACK response and GET REPLY response traffic is significant. System models that attempt to calculate the load on a given link should also consider the induced PUT

ACK and GET REPLY traffic on the return route from the destination.

Our experiments also revealed that the routing scheme used on our systems is performed by traversing the X dimension first, Y next, then Z last. All hops needed in a given dimension are completed before moving onto the next dimension. The shortest number of hops possible is taken in each dimension (i.e., either by traversing in the positive or negative direction). The systems that we examined had no network link failures. It is likely that the routing algorithm is more flexible when there are link failures that must be routed around, but we did not perform experiments to examine this.

D. MPI Bandwidth Efficiency

As a final experiment, we used the host tile counters to measure the actual number of bytes transferred for MPI point-to-point messages of varying sizes. This revealed the different protocols used by Cray's MPI implementation and allowed us to quantify the overall bandwidth efficiency of the system (MPI-Message-Size / Bytes-Transferred). We were surprised by the precision of the tile phit and packet counters. Even for the smallest messages sizes, we observed the counts to be nearly identical from run to run and for the values obtained to be appropriate given the test message size.

The results of the experiment are shown in Figure 5. The plot on the left (Fig. 5a) only includes the bytes injected into the network by the host on the source Gemini, either as PUT requests or GET reply responses. It does not include the the PUT acknowledgments or GET request packets received by the source Gemini. The plot on the right (Fig. 5b) considers the total number of bytes injected and received by the source Gemini due to the MPI message being sent, including PUT acknowledgments and GET requests received by the source. The calculated bandwidth efficiency percentage is shown as the purple line and corresponds to the scale on the right Y axis. For large MPI messages, the overall bandwidth efficiency is calculated based on the empirical tile counters to be approximately 61%, which corresponds to the expected value (105 bytes transferred for every 64 bytes of the MPI message). For smaller messages, the MPI header appears to be around 64 bytes, resulting in low bandwidth efficiency until this cost is amortized.

By examining the jumps in the graph as well as the distribution of traffic over VC0 and VC1, the results suggest that Cray's MPI implementation is using four different protocols depending on message size. The first protocol is for messages from 1 to 16 bytes and is based on PUT transactions from source to destination. Messages from 32 bytes to 4 KB also use a PUT based protocol, but multiple transactions are used, as indicated by the rising number of VC0 bytes received. Messages from 8 KB to 256 KB use a GET based protocol where the destination node pulls the message data from the source using GET transactions. This is indicated by the high VC1 traffic at the source. Finally, 512 KB messages and larger again use a PUT based protocol. The increasing number of VC1 bytes shown on the left plot indicates that a small number of destination to source GET transactions are used in this protocol, but most of the data is transmitted using source to destination PUT transactions. The Gemini's block transfer engine, which is optimized for large transfers, is likely used by this protocol.

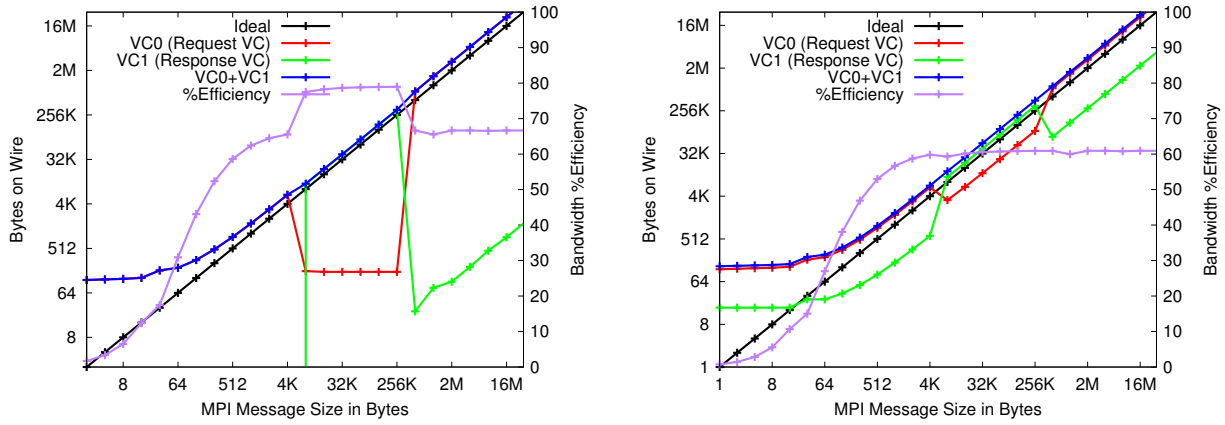
VI. Large-scale MiniGhost Experiments

Our original motivation for using the Gemini's performance counters was to understand why the MiniGhost [5], [6] miniapp from Sandia's Mantevo project [7] was not scaling well on Cielo. MiniGhost is a proxy application, meaning it is designed to be a simpler version of a real application, that implements an explicit time-stepping scheme where the communication is synchronous and is needed to update ghost cells for a 27 point stencil across a regular 3-D grid. Performance benchmarking showed that MiniGhost configured for weak scaling (constant problem size per process) scaled well up to 16K processes running 16 processes per node (1K nodes total), but poorly for larger runs. Runtime was increasing significantly with process count when it should have been staying relatively constant. Our thought was that the Gemini performance counters could tell us if network congestion in the 3-D torus was becoming an issue for large scale runs.

Before we got our Gemini performance counter tools up and running, we discovered that a simple remapping of MiniGhost processes to physical compute nodes could produce much better scaling results. Originally a 1x1x16 group of processes in the overall MiniGhost input problem 3-D grid was distributed to each node. The remapping scheme changed this to place a 2x2x4 group of processes on each node, which intuitively reduces the surface area that must be communicated with neighboring nodes. We used the Gemini tile performance counters to confirm this intuition. Figure 6 shows the number of bytes injected into the interconnect by each Gemini for three different MiniGhost process mapping schemes. The `Random` scheme randomly assigns MiniGhost processes to physical nodes, `No-Remap` is the default 1x1x16 mapping, and `Remap` is the improved 2x2x4 mapping. As can be seen in the figure, `Random` results in by far the most remote communication and the `Remap` scheme results in more than a factor of two reduction in remote communication compared to `No-Remap`. This confirmed our intuition with empirical evidence.

Access to the tile counters also gave us an opportunity to understand overall network congestion by using the input and output stall counters. Figure 7 shows the observed stall counters for MiniGhost running on 128K processes with 16 processes per node (8,192 nodes). The left plot (Fig. 7a) shows the input stall counters, which count the cycles a packet must wait in a Gemini tile's input queue before it can move to the destination tile's output queue. The right plot (Fig. 7b) shows the output stall counters, which count the cycles a packet must wait in a tile's output queue before it can move to the destination remote Gemini.

Both input and output stalls are typically due to some form of flow control, which is a side-effect of network congestion. The left plot shows that the host links experience a high number of input stalls due to network congestion making it harder to inject packets into the network. The right plot shows that host links experience a low number of output stalls, which is because destination nodes are always able to sink packets from the network quickly.



(a) Initiator Transmit Only

(b) Overall (Transmit + Receive)

Fig. 5: Measured bandwidth efficiency for MPI point-to-point messages.

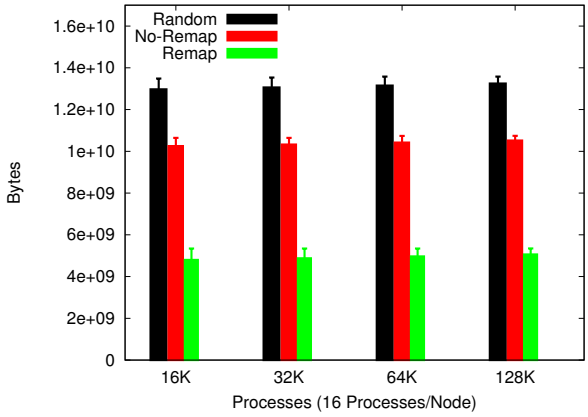


Fig. 6: MiniGhost per-Gemini bytes injected on host links. Filled bars represent the average per-Gemini remote communication for all Geminis in a run. The error bars represent the Gemini with the most remote communication.

The plots show that X dimension network links encounter the highest maximum congestion. We believe this is due to the routing algorithm traversing the X dimension first, resulting in added contention with traffic waiting to move into the Y and Z dimensions. For all dimensions (X, Y, and Z), the `Remap` scheme results in about a factor of two reduction in maximum congestion compared to the `No-Remap` case. This correlates with the observed decrease in MiniGhost communication time. The maximum congestion on any link is the bottleneck for MiniGhost, since it is a bulk-synchronous parallel application. For X dimension output stalls, the average stall count actually increases slightly for the `Remap` scheme, but the reduction in maximum link congestion significantly improves MiniGhost performance.

VII. Future Work

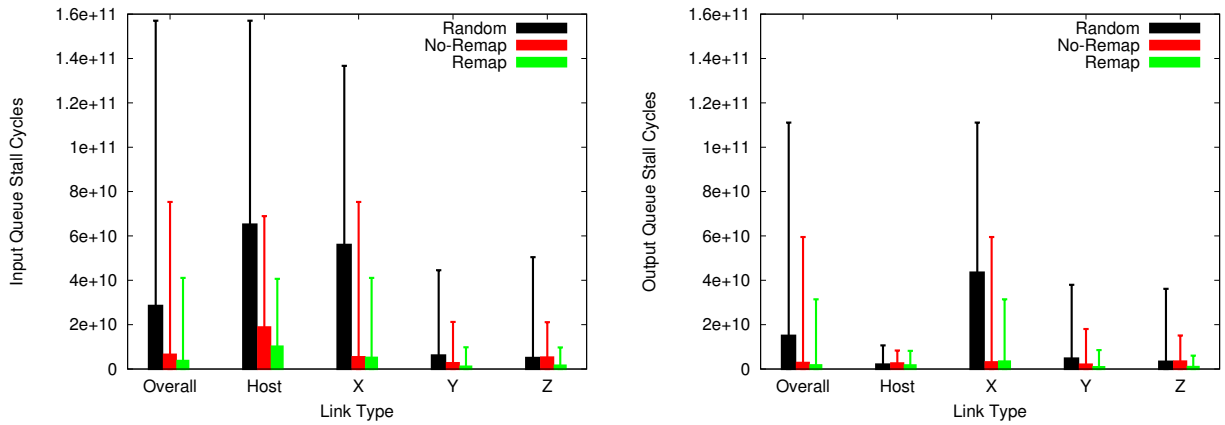
Now that we understand the Gemini's router tile performance counters and have tools for analyzing them, we plan to use the capability to quantify the benefit of different task mapping strategies with respect to network congestion. Specifically, we would like to make use of MPI scalable graph topology communicators [8] along with existing and new topology mapping algorithms [9] to improve the performance of DOE applications running on Cray Gemini and Aries based systems. Cray has recently added support for Gemini and Aries performance counters to PAPI [10], so we will be investigating whether our tools can use this interface rather than the low-level GPCD library described in Section II. A longer term goal is to create a portable interface for accessing and aggregating network performance counter information available on high-end HPC systems [2], [11], and apply this information for dynamic task mapping based on network congestion.

VIII. Conclusion

In this paper we have described our method for accessing the router tile performance counters available on the Cray Gemini network processing unit. This included a description of how to directly access the counters from a user application, aggregate the individual tile counters into logical links, and gather application-wide counter information. We presented the results of several experiments designed to understand the meaning of the counters, including their directionality, and to understand the operation of the system as a whole in terms of network transactions, routing, and MPI point-to-point messaging. Finally, we demonstrated the use of the Gemini's performance counters to quantify the reduction in network congestion due to an improved process mapping scheme for the MiniGhost miniapp. We plan to perform similar studies for other applications and task mapping algorithms in the future.

Acknowledgment

The authors would like to thank Bob Alverson at Cray for his assistance in understanding the Gemini tile counters.



(a) Input Stalls

(b) Output Stalls

Fig. 7: Measured network congestion for MiniGhost running on 128K processes. Filled bars represent the average per-Gemini stall counts for all Geminis in a run. The error bars represent the Gemini with the highest stall count.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, 2010, pp. 83–87.

[2] *Using the Cray Gemini Hardware Counters*, Cray Inc., 2010. [Online]. Available: http://docs.cray.com/books/S-0025-10/

[3] *Using Cray Performance Measurement and Analysis Tools*, Cray Inc., 2013. [Online]. Available: http://docs.cray.com/books/S-2376-610/

[4] "The Gemini Network," Cray Inc., Tech. Rep., Aug. 2010. [Online]. Available: http://wiki.ci.uchicago.edu/pub/Beagle/SystemSpecs/Gemini_whitepaper.pdf

[5] R. F. Barrett, C. T. Vaughan, and M. A. Herox, "Minighost: A miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing," Sandia National Laboratories, Tech. Rep. SAND 2011-5294832, May 2011.

[6] R. F. Barrett, P. S. Crozier, D. W. Doerfler, S. D. Hammond, M. A. Heroux, H. K. Thornquist, T. G. Trucano, and C. T. Vaughan, "Summary of work for ASC L2 milestone 4465: Characterize the role of the mini-application in predicting key performance characteristics of real applications," Sandia National Laboratories, Tech. Rep. SAND 2012-4667, Jun. 2012.

[7] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, Sep. 2009.

[8] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Traeff, "The scalable process topology interface of mpi 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, Aug. 2010.

[9] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *ACM International Conference on Supercomputing (ICS'11)*, Jun. 2011.

[10] *Using the PAPI Cray NPU Component*, Cray Inc., Mar. 2013. [Online]. Available: http://docs.cray.com/books/S-0046-10/

[11] H. Jagode, S. Moore, and D. Terpstra, "Performance Counter Monitoring for the Blue Gene/Q Architecture," in *The 18th Annual Meeting of ScicomP, the IBM HPC Systems Scientific Computing User Group*, ser. ScicomP'12, May 2012.