CHAPTER 5

Xeon Phi Cache and Memory Subsystem

The preceding chapter showed how the Intel Xeon Phi coprocessor uses a two-dimensional tiled architecture approach to designing manycore coprocessors. In this architecture, the cores are replicated on die and connected through on-die wire interconnects. The network connecting the various functional units is a critical piece that may become a bottleneck as more cores and devices are added to the network in a chip multiprocessor (CMP) design such as Intel Xeon Phi uses. The interconnect design choices are primarily determined by the number of cores, expected interconnect performance, chip area limitation, power limit, process technology, and manufacturing efficiencies. The manycore interconnect technology—although it has benefited from existing research on other interconnect topologies in multiprocessor systems and the close interaction among cores, cache subsystem, memory, and external bus—makes interconnect design for coprocessors especially challenging.[1]

The Interconnect Topologies for Manycore Processors

Various topologies can be used to connect the cores in a multicore chip. The most common interconnect topologies in current use are described in the following sections.

Bidirectional Ring Topology

The simplest interconnect topology is *bidirectional ring topology*, where the cores connect with one another through one or multiple hops in both directions (Figure 5-1). The low complexity of the implementation is attractive for a low number of cores. The average number of hops is N/4, where N is the number of cores.

[1]Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. "Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling." *Proc. Intl. Symp. on Computer Architecture* (ISCA), 2005, pp. 408–419; and D. N. Jayasimha, Bilal Zafar, and Yatin Hoskote. "On-Chip Interconnection Networks: Why They Are Different and How to Compare Them." Technical Report, Intel Corp., 2006.

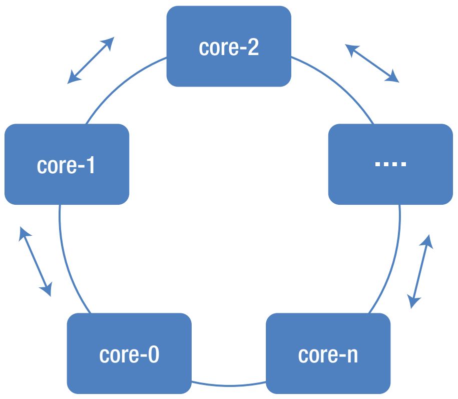


***Figure 5-1.** Bidirectional ring topology*

This ring design's simplicity is offset by latency, bottleneck, and fault-tolerance issues:

- As the number of cores N increases, the number of hops and hence the latency increases.
- A single bidirectional path to move data from one core to another can easily become a bottleneck as the data transfer load increases.
- Any link failure will cause the chip to become nonfunctional.

Two-Dimensional Mesh Topology

Two-dimensional (2D) *mesh topology* (Figure 5-2) is a popular solution for interconnecting cores in a multicore design thanks to being more scalable than a ring network and a more simplified layout in 2D tiled architecture. The routing protocol plays a significant role in the performance of such a network.

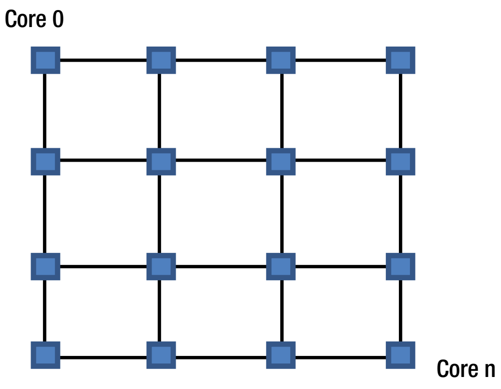


***Figure 5-2.** 2D mesh network*

2D mesh topology has the following drawbacks:

- The power efficiency of a 2D mesh network is relatively low compared to that of a ring network.
- The 2D mesh topology is nonuniform, because the cores at the edges and corners have fewer communication channels and hence less bandwidth available to them.

Two-Dimensional Torus Topology

Two-dimensional torus topology improves on 2D mesh by adding wrap-around wires to the mesh network, as shown in Figure 5-3. This topology removes the nonuniformity of the edge nodes in a 2D mesh, reduces the maximum and average hop counts, and doubles the bisection bandwidth. The downside is the extra wiring that spans the length of the die. Folded torus topology can be used, however, to reduce the wire length to two tiles.

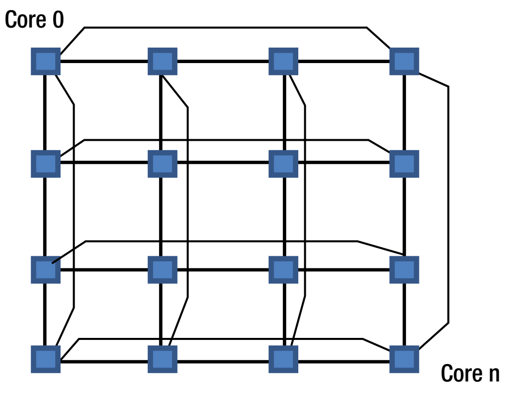


Figure 5-3. *2D torus topology*

Other Topologies

Other networks—such as *3D mesh*, *3D torus*, *fat tree*, and *hierarchical ring networks*—have their respective technological merits, but their wiring density and power requirements are impractical for on-chip interconnects at the current level of technology.

The Ring Interconnect Architecture in Intel Xeon Phi

The Xeon Phi coprocessor implemented bidirectional high performance on the chip ring carrying data and instructions to various agents, such as core, GDDR controllers, and *tag directory* (TD).[2] These agents are connected to the ring through ringstops, as shown in Figure 5-1. Ringstops handle all traffic coming on and off the ring for the attached agents.

[2]*Tags* are numbers used to uniquely identify the cache lines. The *tag directory* is a structure in the cache design that holds information about the cache lines residing in the processor cache.

In the Xeon Phi architecture, there are three pairs of independent rings traveling in two opposite directions (bidirectional), as shown in Figure 5-4. The three pairs carry data, addresses, and acknowledgments. The data ring is 64 bytes wide to feed the high data bandwidth required by a large number of cores. Messages placed on the ring are deterministically delivered to the destination or may remain (bounce) on the ring until they are captured by the destination. The address ring carries address and read/write commands and the acknowledgment ring carries flow control and coherence messages.

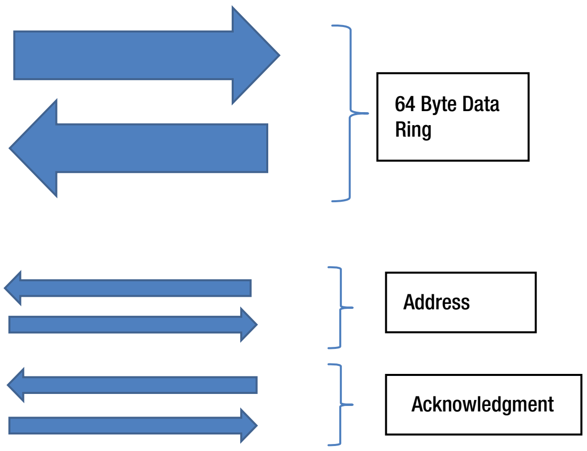


Figure 5-4. *Intel Xeon Phi bidirectional bus*

L2 Cache

The L2 cache is the secondary cache for the core. The L2 cache is inclusive of L1 cache. It is okay to have a cache line in L2 only, but L1 cache lines must have a copy in L2. TD, used for cache coherency, only tracks L2 entries. L2 has ECC, which implements single error correction and double error detections. With 32 GB (35 bits) of address range, each L2 cache associated with a core has 512 kB size divided into 1024 sets and 8-way associativity per set with 64 bytes/1 cache line per way. The cache is divided into two logical banks. There are 32 read buffers of one cache line each. There is a 16-entry write-out/snoop-out buffer of one cache line each, so that a minimum of four of these buffers is guaranteed to be available to snoop for outgoing data.

L2 latency is 11 cycles. If there is an L1 miss but an L2 hit, the latency is 17 cycles if the requesting thread is waiting for the data and 21 cycles if the thread is put to sleep. L2 cache uses pseudo-LRU implementation as a replacement algorithm.

Tag Directory

The Xeon Phi coprocessor implements a physically distributed TD for data coherency among the cores on the ring. It filters and forwards requests to appropriate receiving agents on the ring. It is also responsible for sending the snoop requests to various cores on behalf of the requesting core and returns global L2 line state to the requesting core. It also initiates the communication with the main memory via on-die memory controllers.

The TD is physically attached to each core and gets an equal portion of the whole address space to provide balanced load on the address ring. The TD that is referenced on a L2 miss is not necessarily collocated on the same core that generated the miss but is collocated based on the address. Every physical address is uniquely mapped through a reversible *one-to-one* mapping hash function. L2 caches are kept coherent through TDs and referenced on an L2 miss. A TD tag contains the address, state, and an ID for the owner of the cache line needed to perform the coherency functions.

Data Transactions

On an L2 cache miss by an executing instruction, the core sends the address to TDs through the address ring. If the data are found in another core's L2, a forwarding request is sent to that core's L2 over the address link and the data are returned through the data ring. If the requested data are not found in any of the core's cache, a request is sent to the memory controller from the TD.

The memory controllers are distributed evenly on the ring and the address has *all-to-all* mapping between the TD and memory controllers. The addresses are evenly distributed among the memory controllers to reduce bottlenecks and to provide optimal bandwidth.

Once a memory controller retrieves the requested 64-byte size cache line, it is returned over the data ring to the requesting core.

The Cache Coherency Protocol

The Xeon Phi coprocessor uses a modified MESI protocol for cache coherency control between distributed cores using a TD-based *globally owned, locally shared* (GOLS) protocol. To start with, let's review what an unmodified MESI protocol looks like. The standard MESI state diagram and policies are shown in Table 5-1.

Table 5-1. *Standard MESI Protocol*

L2 Cache State	State Definition
M	*Modified*: Cache line is modified relative to memory. Only one core can have a given line in M state at a time.
E	*Exclusive*: Cache line is consistent with memory. Only one core can have a cache line in E state at a time.
S	*Shared*: Cache line is shared and consistent with other cores, but may not be consistent with memory. Multiple cores can have a given cache line in S state at a time.
I	*Invalid*: Cache line is not present in the cores L1 or L2.

Initially all cache lines are in the *invalid* (I) state. If the data are loaded for writing, the corresponding cache line changes to the *modified* (M) state. If the data are loaded for read and hits the cache in another core, it is marked *shared* (S) state; otherwise it is marked *exclusive* (E) state. If a modified cache line is read or written from the same core, it stays in the M state. If a second core reads this modified cache line, the data are sent to the second core and the GDDR memory update can be delayed due to the global coherency GOLS protocol with the help of TD. If a modified cache line has to be evicted due to a local capacity miss, or if a remote core tries to update the same cache line by a *read for ownership* (RFO),[3] the cache line is written back to GDDR memory and goes to the I state. If an eviction is caused by the remote core's write request, the corresponding cache line in the remote core gets to the E state first and then to the M state after the store retires. If a cache line in the E state is locally written to, it changes to the M state. If a cache line in the S state is requested by another core for write, the state changes to the I state and the cache line is sent directly to the requesting core. The state machine representing MESI policies implemented in a local L1/L2 cache is shown in Figure 5-5.

[3]The RFO is the process common to many cache-based processor architectures of reading a cache line from the memory into the cache before it can be written to.

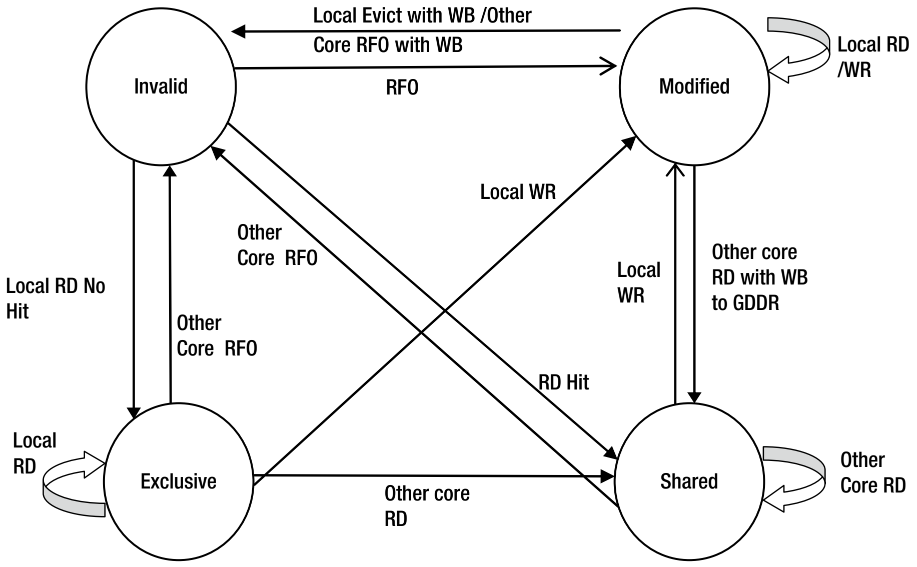


Figure 5-5. *Standard MESI cache coherency policies. RD = read; WR = write; WB = writeback to GDDR; RFO = read for ownership*

In order to remove the potential performance bottleneck resulting from its lack of the *owner* (O) state that is a component of the MOESI protocol, the Xeon Phi coprocessor has implemented TD to manage the global state so the modified cache lines can be shared between the cores without writing back to GDDR memory, thus reducing shared cache-line access between cores. The TD implements the GOLS protocol. By complementing the MESI protocol with the GOLS protocol, it is possible to emulate the O state. Table 5-2 and Figures 5-6a and 5-6b show the augmented MESI and GOLS protocols.

Table 5-2. *GOLS Protocol*

TD State	State Definition
GOLS	*Globally owned locally shared*: Cache line is present in one or more cores but inconsistent with memory (GDDR).
GS	*Globally shared*: Cache line is present in one or more cores and consistent with memory.
GE/GM	*Globally exclusive/modified*: Cache line is owned by one and only one core and may or may not be consistent with the memory. The TD does not know whether the core has modified the cache line or not.
GI	*Globally invalid*: Cache line is not present in any core.

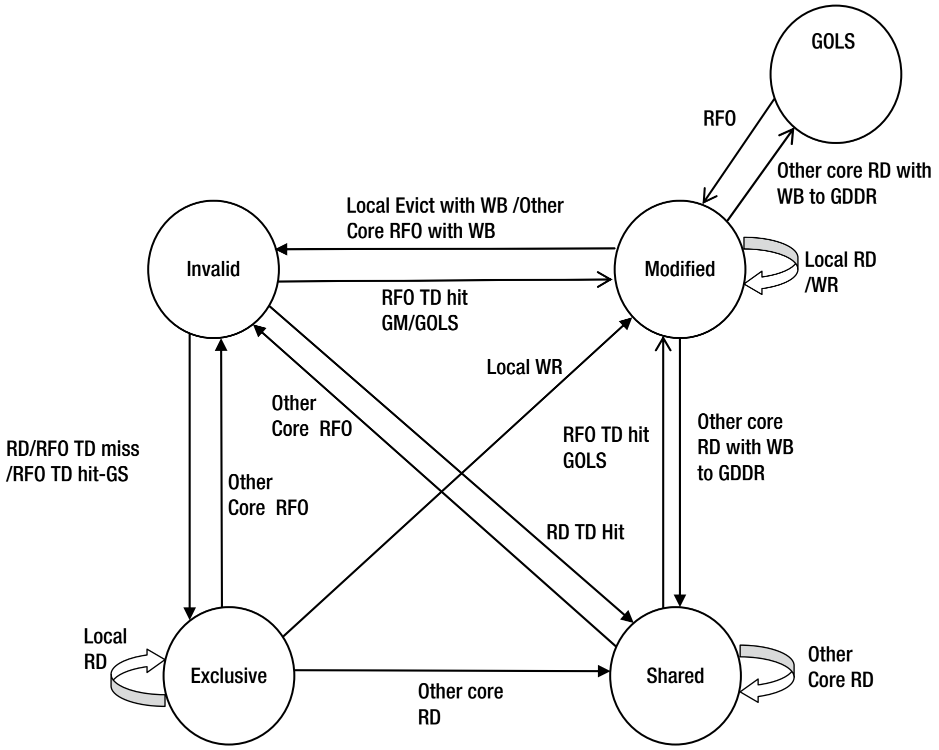


Figure 5-6a. *Intel Xeon Phi augmented MESI with GOLS protocol in the core. RD = read; WR = write; WB = writeback to GDDR; RFO = read for ownership; TD = tag directory*

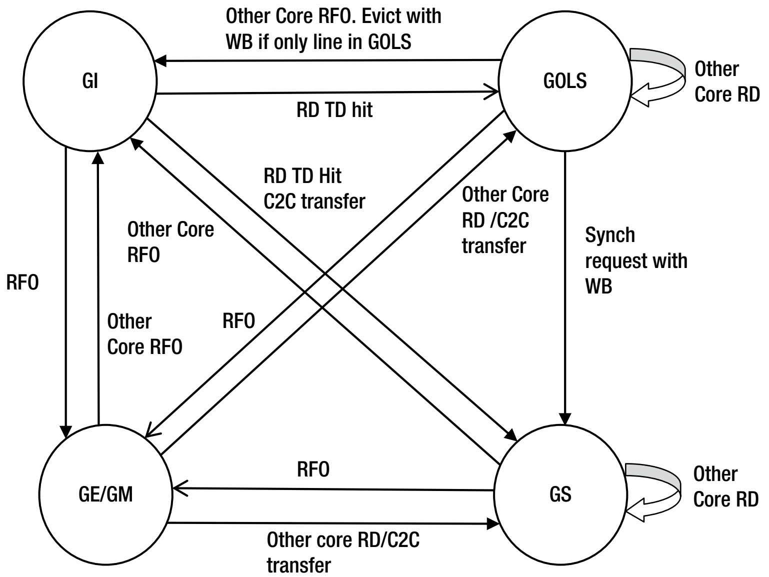


Figure 5-6b. *Intel Xeon Phi augmented MESI with GOLS protocol in the TD. RD = read; WR = write; WB = writeback to GDDR; RFO = read for ownership; TD = tag directory; C2C = cache to cache*

The augmented MESI with GOLS protocol is depicted in Figure 5-6 and works as follows. Initially, all the cache lines in the cores and TD state are invalid (I state for L1/L2 and GI state for TD). If one of the cores is read or RFO'd (to be written in the future), that core's cache line state becomes E state and the TD state becomes GE/GM. If the first core's data are in the shared state and a second core reads the data, the TD state becomes GS state (*globally shared*) and gets the data directly from the first core marked as C2C transfer, as shown in Figure 5-6b. However, if the first core modified the data (acquired through RFO), the modified data are sent to the requesting core and individual core's cache-line states become S state. The TD state becomes GOLS, indicating the cache lines are shared between the cores and may be inconsistent with the GDDR memory.

If the cache line that is in the S state is evicted and there are other copies of the line in another core's cache, the TD state remains GOLS and the evicted cache-line state becomes I state. However, if the evicted cache line is the only cache line and TD state is marked as GOLS, the cache line is written back to GDDR memory and the TD state becomes GI state.

Hardware Prefetcher

The Intel Xeon Phi coprocessor implements a second-level cache hardware prefetcher that is part of the uncore and responsible for fetching cache lines into the L2 cache. It selectively prefetches code, read, and read for ownership data into L2. It has 16 stream entries and allocates a stream on a demand miss to a new 4-kB page. Subsequent requests that hit and miss the same page within a certain address range train the stream entry. Once a stream direction is detected, forward or backward, the prefetcher issues up to four multiple prefetch requests.

Code streams are always prefetched forward. When a stream is at the end of the prefetch 4-kB page boundary, it kick-starts an extra prefetch for the new page, assuming new page allocation of the existing stream.

The Memory Controllers

The Intel Xeon Phi coprocessor comes with eight memory controllers with two channels, each communicating with GDDR5 memory at 5.5 GT/s. The memory controllers are connected to a ring bus using ringstops on one end and GDDR5 on the other end. This provides approximately 352 GB/s of memory bandwidth. Memory access requests are directed to appropriate memory controllers to access data from corresponding GDDR5 memory modules. The cores support two types of memory: *un-cacheable* (UC) and *write-back* (WB).

The memory controllers interface with the ring bus at full speed and receive a physical address with each request. They translate the read/write requests from core to GDDR5 protocols and submit the commands to the memory devices. The memory controllers are also responsible for scheduling GDDR5 requests to optimize the memory bandwidth available from GDDR memory and for guaranteeing bounded latency for special requests from the system interface unit providing data over the PCI express bus.

■ **Note** Sample Calculation of the Theoretical Memory Bandwidth of an Intel Xeon Phi Coprocessor

Given eight memory controllers with two GDDR5 channels running at 5.5 GT/s

Aggregate Memory Bandwidth	= 8 memory controllers × 2 channels × 5.5 GT/s × 4 bytes/transfer
	= 352 GB/s

The memory addresses are interleaved between the GDDR5 devices to enable better memory bandwidth and ring bus utilization. The memory requests are 4-kB lines at a time and are interleaved between memory modules. This means consecutive memory locations are distributed among the memory modules and there is no way to place the memory close to a core for optimal memory bandwidth.

Memory Transactions Flow

A good understanding of how memory requests are handled by the memory controller may help in root-causing application performance issues running on the Xeon Phi coprocessor. This section will look at the details of how memory transactions are generated and handled by various servers and consumers of the data inside the Xeon Phi coprocessor.

Cacheable Memory Read Transaction

The memory address ranges can be divided into two broad categories: the cacheable and uncacheable memory ranges. Memory marked as *uncacheable* is not saved into cache for later access and directly delivered to the requester. For performance reasons, most memory transactions of interest to developers of applications running on Xeon Phi are cacheable. The memory transactions for address ranges that are cacheable in the core L1 and L2 caches are described below and shown in Figure 5-7.

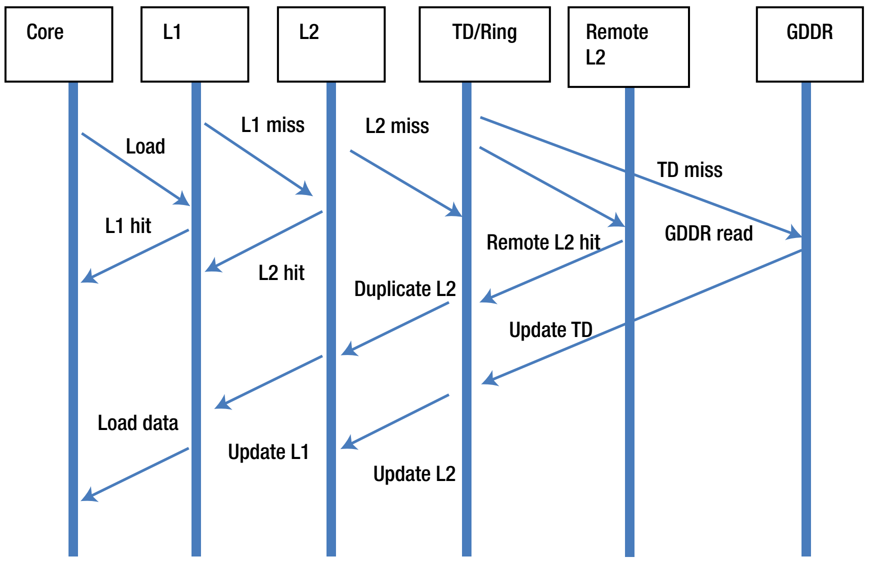


Figure 5-7. *Core/VPU load operation*

1. A read transaction request for data is generated from a core or vector unit.
2. If the data are found in L1 cache, the data are returned to the core, or else a local L2 lookup happens.
3. On a local L2 miss, a lookup to the TD happens. The TD contains all of the L2 occupancy information. The TD entry looked up might not be local to the core suffering the cache miss and sent through ring interconnect.
4. If the data are found in the L2 cache, the data are returned to the requesting core through an L1 update.
5. If not found in the L2 cache, the data have to be fetched from the GDDR memory to the L2 cache. The TD converts the address into a physical address and submits the physical addresses to the memory controller.

6. The memory controller converts the physical address to the appropriate memory access information channel/bank/row/column needed by the hardware.

7. There is a read queue for each memory channel and an entry is allocated when a read request is made to the memory controller. It uses a credit-based mechanism to prevent overflow of the read buffers.

8. The memory controller sends the read operation to GDDR memory, which returns the data to the read buffer.

9. The returned data update the TD, update the L2 cache with the appropriate cache, evict if needed, and return the data to the requesting core by appropriate cache level updates to L2 and L1.

Figure 5-7 shows the control and data flow for a memory read operation.

Managing Cache Hierarchy in Software

Because Intel Xeon Phi is a cache-based machine, any software trying to extract performance out of this machine must strive hard to make data available in the cache when the computation needs it. When optimizing code for this or any cache-based architecture, it is of the utmost importance to manage the cache hierarchy properly—through loop reorganization, data structure modifications, code and data affinity, and so on—to make sure the data are available in the cache as soon as needed.

To work with memory hierarchy complexity, the Intel Xeon Phi coprocessor includes various instructions and constructs such as prefetch, gather prefetch, clevict instructions, and nontemporal hints. These instructions allow one to pull in data to the appropriate cache level to hide memory access latency. The cache line eviction instructions—'clevict'—are also useful when you want to remove unnecessary data from the cache to make space for new data or to keep useful data in the cache by managing the LRU state of cache lines. When the cache controller needs to select a cache line for eviction, the LRU state is used to select the victim cache line. These instructions are often useful when the hardware prefetcher fails due to irregularly spaced data access patterns. The vprefetch* instructions are implemented for performing cache line prefetches that the hardware might not be able to issue automatically. Xeon Phi also implements gather prefetch instructions—vgatherpf*. Prefetch instructions can be used to fetch data to L1 or L2 cache lines. Since this hardware implements an inclusive cache mechanism, L1 data prefetched is also present in L2 cache-but not vice versa.

For prefetch instructions, if the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetch instructions can specify invalid addresses without causing a *general protection* (GP) fault because of their speculative nature.

The Intel compiler provides `-opt-prefetch switch` to tell the code generator to insert prefetch instructions in the code and the `-opt-prefetch-distance switch` to globally define the L1 and L2 prefetch distances. You can also tell the compiler not to generate prefetch instructions by setting the `-no-opt-prefetch` compiler switch or setting `-opt-prefetch=0`.

Let's look at how the compiler uses these flags to add instructions to your generated code. Code Listing 5-1 shows a small loop that runs on a single thread and performs memory access on an array of structures.

Code Listing 5-1. Compiler Prefetch Example

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE  1000000
#define ITER       20


```

```
typedef struct pointVal {
              double x, y, z;
              double value;
}POINT;

__declspec(align(256)) static POINT  a[SIZE];

extern double elapsedTime (void);

int main()
{
        double startTime, duration, tmp[SIZE];
        int i, j;
//initialize
          for( j=0; j<SIZE;j++){
              a[j].x=0.1;
           }

       startTime = elapsedTime();

        for(i=0; i<ITER;i++) {
           for( j=0; j<SIZE;j++){
             tmp[j]+=a[j].x;
           }
        }
        duration = elapsedTime()-startTime;

        double MB = SIZE*sizeof(double)/1e+6;
        double MBps = ITER*MB/duration;
        printf("DP ArraySize =  %lf MB, MB/s = %lf\n", MB, MBps);

  return 0;
}

```

In order to turn off the prefetch instructions being generated by the compiler, I first compiled the code with the –no-opt-prefetch option and send the output gather.out code to the mic card using "scp gather.out mic0:/tmp" as follows:

```
Command_prompt-host >icpc  -mcmodel=medium -O3 -no-opt-prefetch -mmic   -vec-report3  gather.cpp
gettime.cpp -o gather.out
```

If I run the code on the mic card, I see that the code is able to achieve ~381 MB/s on the single-threaded run of this code.

```
command_prompt-mic0 >./gather.out
DP ArraySize =  8.000000 MB, MB/s = 381.794093
```

Now, I recompile the same code by removing the –no-opt-prefetch compiler option and upload the file to the mic card, as described above. The same run with the –no-opt-prefetch restriction removed will generate prefetch instructions and will cause substantial improvement in memory access performance, as shown below. The performance has gone up from 381 to 481 MB/s.

```
command_prompt-mic0 >./gather.out
DP ArraySize =  8.000000 MB, MB/s = 484.896942
command_prompt-mic0 >
```

You can generate the assembly file corresponding to Code Listing 5-1 by using –S switch as shown below. A fragment of the generated assembly code is shown in Code Listing 5-2. I also added the –unroll0 compiler switch to turn off code unrolling optimization so that the generated assembly is easier to read. Although I am not going to walk through the assembly listing here, I would like to point out some of the prefetch instructions highlighted in Lines 170 and 172 of the assembly in Code Listing 5-2, where it is prefetching the array a to Cache Level 1. You may also use the compiler option `-opt-report-phase=hlo` when building the code, so that the compiler will provide you diagnostics on where it generated the software prefetch instructions, if any:

```
Command_prompt-host >icc -mmic -vec-report3 -O3 -c -S -unroll0 -opt-report-phase=hlo gather.cpp
```

If you look at the kernel loop in Code Listing 5-1, you will notice that the code accesses nonconsecutive data elements: in this case, coordinate x in a[].

```
 62           for( j=0; j<SIZE;j++){
 63             tmp[j]+=a[j].x;
 64           }
```

This requires the compiler to use gather instructions as shown in Lines 180 and 183 of Code Listing 5-2. The Intel Xeon Phi ISA supports gather/scatter instructions to help sparsely placed data to move in/out of a vector register. These instructions simplify vector code generation for complex data structures and allow further hardware optimization of the instructions in future coprocessors. There are also prefetch instructions available corresponding to vector gather/scatter instructions. These are `vgatherpf0dps` for L1 and `vgatherpf1dps` for L2 prefetch.

Code Listing 5-2. Compiler Generated Prefetch Instruction

```
170         vprefetch0 a(%rip)                                  #63.22 c17
171 ..LN46:
172         vprefetch0 256+a(%rip)                              #63.22 c21
173         .align    16,0x90
174 ..LN47:
....
177 ..LN48:
178         kmov      %k1, %k2                                  #63.22 c1
179 ..LN49:
180         vprefetche1 512(%rsp,%rcx,8)                        #63.14 c1
181 ..L13:                                                      #63.22
182 ..LN50:
183         vgatherdpd a(%rdx,%zmm1), %zmm3{%k2}                #63.22
184 ..LN51:
185         jkzd      ..L12, %k2    # Prob 50%                  #63.22
186 ..LN52:
187         vgatherdpd a(%rdx,%zmm1), %zmm3{%k2}                #63.22
188 ..LN53:
```

```
189         jknzd     ..L13, %k2    # Prob 50%                        #63.22
190 ..L12:                                                            #
191 ..LN54:
192         vaddpd    (%rsp,%rcx,8), %zmm3, %zmm4
194         vprefetch0 256(%rsp,%rcx,8)
196         vprefetch1 2048+a(%rdx)
197 ..LN57:
```

If `vgatherpf0dps` misses both L1 and L2, the resulting prefetch in L1 is nontemporal, but the prefetch into L2 is a normal prefetch.

The gather instruction 'vgatherd' is able to access up to 16 32-bit elements. The actual number of elements accessed is determined by the number of bits set in the vector mask provided as the source. In Intel Xeon Phi, vgatherd can load multiple elements with a single 64-byte memory access if all the elements fall in the same 64-byte cache line.

The gather instruction guarantees at least one element to be gathered for each call. In Lines 185 and 187 of Code Listing 5-2, the compiler uses a mask register, k2 to determine whether all the required elements are gathered or not.

Probing the Memory Subsystem

This section investigates the GDDR memory characteristics of the Xeon Phi coprocessor.

Measuring the Memory Bandwidth on Intel Xeon Phi

One key performance metric related to the cache subsystem is the GDDR memory bandwidth, seen by a computational code. To measure the GDDR memory bandwidth, we write and examine the small benchmark in Code Listing 5-3.

Code Listing 5-3. Measuring GDDR Memory BW as Seen by Cores

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>


#define SIZE    (180*1024*1000)
#define ITER       20

__declspec(align(256)) static double  a[SIZE],  b[SIZE],   c[SIZE];


extern double elapsedTime (void);

int main()
{
        double startTime,  duration;
        int i, j;

        //initialize arrays
        #pragma omp parallel for
54         for (i=0; i<SIZE;i++)
55         {
56                 c[i]=0.0f;
57                 b[i]=a[i]=(double)1.0f;
58         }
59
60         //measure c = a*b+c performance
61         startTime = elapsedTime();
62         for(i=0; i<ITER;i++) {
63         #pragma omp parallel for
64            for( j=0; j<SIZE;j++){
65               c[j]=a[j]*b[j]+c[j];
66            }
67         }
68         duration = elapsedTime() - startTime;
69
70         double GB = SIZE*sizeof(double)/1e+9;
71         double GBps = 4*ITER*GB/duration;
72         printf("Running %d openmp threads\n", omp_get_max_threads());
73         printf("DP ArraySize =  %lf MB, GB/s = %lf\n", GB*1000, GBps);
74
75   return 0;
76 }
```

The code consists of three double-precision arrays, a, b, and c. The size of arrays is set to (180*1024*1000). That number was chosen so that I can divide the work among 180 threads on the 60 available compute cores of the coprocessor. In order to create optimal code, I selected the number of elements to be 124 so that each vector access could be cacheline-aligned. The 1000 multiplier makes the size of the array large enough (each array ~ 1.4 GB) to go outside the cache. The ITER in Line 40 sets the number of times I need to run the bandwidth loop to account for run-to-run performance variations. The benchmark uses the timing routine described in Chapter 4 for core computational flops measurements.

The computation is a simple c = a*b+c operation, where each of the arrays is double-precision. At each iteration, we read in three DP numbers—a,b,c—and write out a single DP number 'c'. This is shown in Line 65 of Code Listing 5-3. Lines 53 to 58 initialize the array to some random numbers. I also make sure the arrays are aligned by declaring them as 256-byte align by __declrspec(align(256)) in Line 42.

In order to measure the BW, I run the inner compute loop ITER time and capture the start and end time during the whole run at Line 68. The amount of memory used by each array can be calculated with (GB = `SIZE*sizeof(double)/1e+9;`).

The total number of memory operations is 4 (3 reads and 1 write). Hence, the BW, as seen by this compute kernel, can be computed as:

```
4*GB*ITER/duration
```

where GB equals the gigabyte size of each of the arrays, ITER is the number of times the BW kernel iterates inside the timing count, and duration is obtained by collecting the total execution times from start to the end of the ITER loop.

This code is cross-compiled with the –mmic switch as follows:

```
icpc -mcmodel=medium -O3  -mmic  -openmp -vec-report3  bw.cpp gettime.cpp -o bw.out
```

Since the code uses three large arrays ~1.75GB each, we needed to use the switch `-mcmodel=medium`, which tells the compiler to expect the data size to be above 2GB and handle that accordingly, as it is for this case. This compile command will generate a bw.out binary that can run on the Intel Xeon Phi coprocessor.

Once the binary is generated, follow the commands in Chapter 4 to copy the binary over to the Intel Xeon Phi card native virtual drive using:

```
command-prompt-host>scp bw.out mic0:/tmp
```

Also, upload the necessary openmp file from the compiler to the card using the following command:

```
command-prompt-host>scp /opt/intel/composerxe/lib/mic/libiomp5.so mic0:/tmp
```

Once the binary and the corresponding files are on the card's virtual drive, you can execute the code by logging in to the card by ssh:

```
Command-prompt-host > ssh mic0
```

Inside the Intel Xeon Phi card, export LD_LIBRARY_PATH to point to the appropriate folder with openmp runtime library:

```
Command-prompt-mic0 > export $LD_LIBRARY_PATH=/tmp:$LD_LIBRARY_PATH;
```

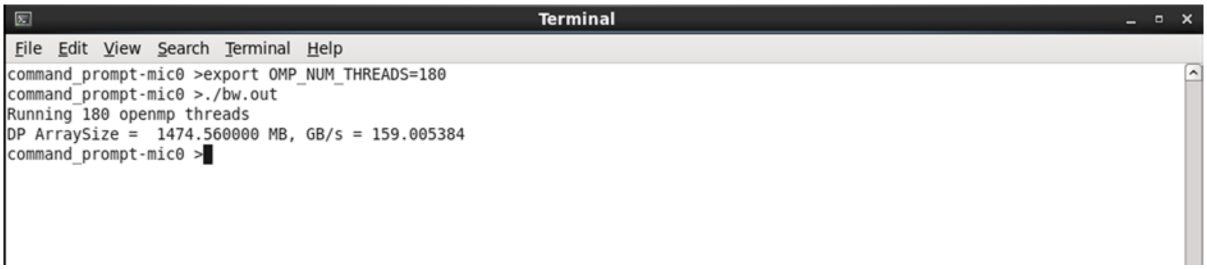
Now set the number of openmp threads to 180 by "export OMP_NUM_THREADS=180" and execute the command ./bw.out as follows:

```
Command_prompt-mic0> export OMP_NUM_THREADS=180
Command_prompt-mic0>./bw.out
```

This should output the following on the terminal window:

```
Command_prompt-mic0> ./bw.out
Running 180 openmp threads
DP ArraySize = 1475.56 MB and GBs = 159.005
Command_promot-mic0 >
```

The output (Figure 5-8) shows that the code ran with 180 threads and was able to measure ~ 159 GB/s overall BW for memory access on this kernel.



```
command_prompt-mic0 >export OMP_NUM_THREADS=180
command_prompt-mic0 >./bw.out
Running 180 openmp threads
DP ArraySize =  1474.560000 MB, GB/s = 159.005384
command_prompt-mic0 >
```

Figure 5-8. *Output of BW benchmark run*

Summary

This chapter covered the memory subsystem of the Xeon Phi coprocessor. You learned about the interconnect ring that connects various components of Xeon Phi, such as cores and memory controllers. You looked at various cache levels and how the cache coherency protocol MESI with GOLS extension is used to maintain data consistency among the caches associated with different cores. You saw how to measure the memory bandwidth on a Xeon Phi coprocessor.

The next chapter will explain the PCIe bus and power management support on the Xeon Phi coprocessor.