Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENCE IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel, the Intel logo, Xeon®, Intel® Xeon Phi™, are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright 2013 Intel Corporation. All rights reserved.

Revision History
<table>
<thead>
<tr>
<th>Document Number</th>
<th>Revision Number</th>
<th>Description</th>
<th>Revision Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXXXX-XXX</td>
<td>1.0.0</td>
<td>Initial release.</td>
<td>October 2013</td>
</tr>
</tbody>
</table>
# Contents

1. Introduction .......................................................................................................................... 1

2. Intel® MPSS Considerations for Optimal Performance ..................................................... 2
   2.1 General Considerations .................................................................................................... 2
       2.1.1 Intel® Xeon Phi™ Coprocessor Clock Sources ......................................................... 2
       2.1.2 Huge Pages .................................................................................................................. 3
           2.1.2.1 Transparent 2MB Page Support ........................................................................ 4
           2.1.2.2 Manual 2MB Pages and Libhugetlbfs ................................................................. 5
       2.1.3 Thread Affinity Mapping ............................................................................................ 6
           2.1.3.1 APIC Mapping and Linux* Thread Enumeration .................................................. 6
           2.1.3.2 Setting Affinities on Linux* .............................................................................. 8
           2.1.3.3 The Linux* Boot Strap Processor (BSP) ............................................................. 8
       2.1.4 Avoiding Page Faults ................................................................................................. 8
           2.1.4.1 Memory Allocations in Linux* .......................................................................... 8
           2.1.4.2 Using MMAP ..................................................................................................... 8
           2.1.4.3 Using Other Allocators .................................................................................... 9
   2.2 Considerations for Offload Applications .......................................................................... 9
       2.2.1 Symmetric Communication Interface (SCIF) .......................................................... 10
           2.2.1.1 Data Transfer Mechanisms .............................................................................. 10
           2.2.1.2 Buffer Alignment ............................................................................................... 10
   2.3 Power Management and NUMA .................................................................................... 14
       2.3.1 Power Management ................................................................................................. 14
       2.3.2 NUMA Issues ........................................................................................................... 15
List of Figures

Figure 1  Performance Benefits: Comparing 2M, 64K, and 4K Page Size ......................... 4
Figure 2  MIC Offload Application Flow ................................................................. 9
Figure 3  Case 1 where source and destination buffers are cacheline aligned and mutually aligned ................................................................. 11
Figure 4  Case 1 data transfer bandwidth at 8K and 16K payload .............................. 11
Figure 5  Case 2 where source and destination buffers are not cacheline aligned but are mutually aligned ................................................................. 12
Figure 6  Case 2 data transfer bandwidth at 8K payload is only half of Case 1 .......... 13
Figure 7  Case 3 with the extra “Shadow Buffer” ......................................................... 13
Figure 8  Case 3 performance degrades due to additional increase in overheads ....... 14
Figure 9  Intel® Xeon Phi™ coprocessor attached to local socket ............................ 15
Figure 10 Intel® Xeon Phi™ coprocessor attached to remote socket ....................... 16

List of Tables

Table 1  Intel® Xeon Phi™ Coprocessor Clock Sources Compared .......................... 3
Table 2  Core to Pthread Mappings for a 57-Core Part ............................................. 7
Table 3  Core to Pthread Mappings for a 61-Core Part ............................................. 7
1 **Introduction**

This document captures best known methods (BKMs) and settings to get the most out of your Intel® Xeon Phi™ coprocessor, with specific focus on the Intel® Manycore Platform Software Stack (Intel® MPSS).

This document captures the runtime settings for optimal performance. Compiler or build time BKMs are not captured, nor are Intel® MIC Architecture-optimal algorithmic design BKMs.
2 Intel® MPSS Considerations for Optimal Performance

This section describes Intel® MPSS setup and settings important to achieving optimal application performance. This section is divided into two categories: General Considerations and Considerations for Offload Applications.

2.1 General Considerations

2.1.1 Intel® Xeon Phi™ Coprocessor Clock Sources

Intel® Xeon Phi™ coprocessor users have access to the following main clock sources:

- TSC ("Time Stamp Counter")
- MICETC (MIC “Elapsed Time Counter”)
- Jiffies counter in Linux®

The Intel® Xeon Phi™ coprocessor does not have the same set of traditional clock sources found in the Intel® Xeon® family of processors.

For the Intel® Xeon Phi™ coprocessor running a Linux® micro-OS, the available and current clock sources can be determined as shown below:

```
user_prompt> cat \
/sys/devices/system/clocksource/clocksource0/available_clocksource
tsc micetc jiffies
```

```
user_prompt> cat \
/sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

The TSC has the finest granularity among this list. There is also a TSC in each core accessible to the 4 threads in that core, and a mechanism has been implemented to synchronize all the TSCs across the chip. Unlike its Intel® Xeon® processor variant, the TSC in the Intel® Xeon Phi™ coprocessor is not frequency-invariant. On the normal operating frequency/state, the TSC is the best clock source for performance timing. Since the Intel® MPSS Gold Update 3 release, recalibration code was added to recalibrate the TSC counters when a throttling event occurs, making TSC frequency-invariant to the user. TSC has become the default clock source in Intel® MPSS.
The MICETC is a clock source from the PCIe unit. There is only one MICETC and it is primarily designed to support PCIe clock needs. Thus, it has lower resolution than the TSC. Moreover, getting a clock signal to and from each core and the MICETC unit will involve multiple hops, and possibly multiple contentions from any of the core requesters, and is also serialized before reaching the PCIe unit. The minimum penalty of using MICETC to do gettimeofday() involves making a ring transition and a serialization in PCIe unit. Hence, the use of the MICETC is not encouraged for performance measurement. The table below shows some examples using the MICETC versus the TSC. Because it is frequency-invariant, it is the most reliable clock source and is used as a reference for recalibrating TSC after returning from a non-normal or non-P1 states.

### Table 1  Intel® Xeon Phi™ Coprocessor Clock Sources Compared

<table>
<thead>
<tr>
<th></th>
<th>MICETC</th>
<th>TSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_gettime()</td>
<td>gettimeofday()</td>
<td>clock_gettime()</td>
</tr>
<tr>
<td>Avg</td>
<td>1550.46</td>
<td>1612.81</td>
</tr>
<tr>
<td>Std</td>
<td>304.19</td>
<td>239.45</td>
</tr>
<tr>
<td>Min</td>
<td>1370.15</td>
<td>1470.21</td>
</tr>
<tr>
<td>Max</td>
<td>2665.89</td>
<td>2399.03</td>
</tr>
</tbody>
</table>

The jiffies is an increment counter in the Linux* kernel and does not have the fine granularity of the TSC. It can be used as a back-up timer if the kernel deems the TSC has drifted sufficiently to become an unstable clock source.

### 2.1.2 Huge Pages

Huge pages are important to some application performance as described best in this LWN.net article: [Transparent huge pages in 2.6.38](https://lwn.net/Articles/915884/). To paraphrase, they can “improve performance through reduced page faults” and reduce “the cost of virtual to physical address translation (fewer levels of page tables must be traversed to get to the physical address). But the real advantage comes from avoiding translations altogether. If the processor must translate a virtual address, it must go through as many as four levels of page tables, each of which has a good chance of being cache-cold, and, thus, slow. For this reason, processors maintain a “translation lookaside buffer” (TLB) to cache the results of translations, but TLB silicon resources are scarce. To reduce TLB pressure, the main kernel address space is mapped with huge pages -- a single 2MB huge page will only require a single TLB entry while the same memory, in 4KB pages, would need 512 TLB entries. So it’s not too surprising that the use of huge pages can potentially make programs run faster.
Studies done by the Intel® Many Integrated Core Architecture (Intel® MIC Architecture) team showed that the average benefits over several workloads ranges from >8% benefit in using 64K pages, and >15% benefit from 2MB pages (see Figure 1).

![Performance Benefits: Comparing 2M, 64K, and 4K Page Size](image)

Prior to the Intel® MPSS Gold Update release, the only way for user-space to take advantage of huge pages in current the Linux* kernel was through the hugetlbfs. Dynamically linked performance critical computing applications can use malloc() without code modifications and still get huge pages through the use of libhugetlbfs. But with a kernel upgrade to 2.6.38 in the Intel® MPSS Gold Update version for the micro-OS, transparent huge pages are now supported.

### 2.1.2.1 Transparent 2MB Page Support

Beginning with the Intel® MPSS Gold Update 1 version, the Intel® Xeon Phi™ coprocessor micro-OS is based off the 2.6.38 kernel which supports transparent 2MB pages. From the LWN.net documentation, “Transparent Hugepage Support is an alternative means of using huge pages for the backing of virtual memory with huge pages that supports the automatic promotion and demotion of page sizes and without the shortcomings of hugetlbfs.”

The current enabled state of the OS transparent huge pages can be seen as shown below.

```
user_prompt> cat \
/sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```
These can be enabled/disabled with the following commands:

```
user_prompt> echo always \
> /sys/kernel/mm/transparent_hugepage/enabled
user_prompt> echo madvise \
> /sys/kernel/mm/transparent_hugepage/enabled
user_prompt> echo never \
> /sys/kernel/mm/transparent_hugepage/enabled
```

See [Documentation/vm/transhuge.txt](Documentation/vm/transhuge.txt) for a full documentation of Transparent Hugepage Support in the Linux* 2.6.38 kernel.

### 2.1.2.2 Manual 2MB Pages and Libhugetlbfs

Prior to the Intel® MPSS Gold Update release, there was no transparent huge page support but 2MB pages can still be manually enabled. There are two ways to enable 2MB pages:

1) Write a value to `/proc/sys/vm/nr_hugepages` which will allocate a number of “persistent” huge pages.
   a) This requires root access.
   b) The value written is the number of 2MB pages that will be set aside.
   c) When a value is written to this file, pages are actually faulted in by the OS. This means that this pool of memory is physically backed and not available to the system unless a value of zero is written to `/proc/sys/vm/nr_hugepages` at which time the memory is freed back to the system.

2) Write a value to `/proc/sys/vm/nr_overcommit_hugepages`
   a) This requires root access, however `/proc/sys/vm/nr_overcommit_hugepages` is now set to cover all of the memory on the device when the mpss service startup completes.
   b) The value written is the number of 2MB pages that will be set aside.
   c) This tells the OS to allow that many number of 2MB pages specified to be allocated on the fly by the standard allocator. Only the needed memory is used and it is automatically returned to the system when it is freed by the application.

In both cases, mmap is used while specifying the MAP_HUGETLB flag to allocate pages from the huge pool, and no mount is required. On the upside, if the mmap call succeeds you are guaranteed that the allocated memory is using 2MB pages. With transparent 2MB page support, there is no guarantee of allocated page sizes.

For user apps that uses `malloc()`, or a dynamically linked performance critical computing applications that cannot afford code modifications, there is the libhugetlbfs. From [http://libhugetlbfs.sourceforge.net/](http://libhugetlbfs.sourceforge.net/): “libhugetlbfs is a library which provides easy access to huge pages of memory. It is a wrapper for the hugetlbfs file system. Applications can use huge pages to fulfill `malloc()` requests without being
recompiled by using LD_PRELOAD. Alternatively, applications can be linked against libhugetlbfs without source modifications to load text or BSS or BSS, data, and text segments into large pages. The library also comes with several user space tools to help with huge page usability, environment setup, and control.”

The most common usage here is to use LD_PRELOAD to preload the library before the process makes any allocation calls and then transparently serve 2MB page backed allocations any time the standard malloc() gets called. Note that the application does not have to be modified in any way to take advantage of this.

In a normal glibc malloc() call, the system tries to fulfill the request by using the current heap. If it cannot service the request, morecore() is called which by default is a call to sbrk(). This creates more memory space for the current process, thereby growing the heap. The allocation can then be fulfilled and the pointer to the new allocation is returned to the user.

When using libhugetlbfs, the system tries to fulfill the request by using the current heap. If it cannot (this will always be true of the first allocation), it calls morecore() which has been overridden by libhugetlbfs. The library uses the native hugetlb support in the OS to mmap 2MB pages in order to grow the heap. The allocation can then be fulfilled (with 2MB backed pages) and the pointer to the new allocation is then returned to the user.

The constant use of libhugetlbfs is also not a panacea. Libhugetlbfs makes every dynamic allocation backed by 2MB pages. For workloads that use only large allocations, this is probably OK. But imagine an application that does many small allocations, this will mean using many 2MB memory. This can cause a lot of pressure on the 2MB TLBs and the application will suffer from numerous page walks due to insufficient TLB resources. It is best to try balancing allocations to take advantage of both the 4K and 2MB TLB resources on the hardware.

2.1.3 Thread Affinity Mapping

There can be various reasons why threads in a multicore processor may need to be affinitized. For example, threads working together on a cache block as a “team” should probably be on the same core (which shares the L2$) to avoid cache line bouncing and take advantage of data locality. In some cases, one may not want the kernel load balancer to move threads around and thread affinitization can explicitly pin them to a specific CPU hardware thread. These are some simple examples of thread affinitization that can have performance impact.

2.1.3.1 APIC Mapping and Linux* Thread Enumeration

On Linux*, the pthread IDs of the threads are not topologically consistent. In an HPC scenario, threads on the same core may be designed to take advantage of data sharing between them, and this enumeration can lead to some performance issue as explained below.
Note that Linux* chooses its boot thread to be the first thread on the last core of the device. For a 57-core device with 4 threads per core, this is the hardware thread 224, but for a 61-core part with 4 threads per core, this is the hardware thread 240. Both these BSP threads are then mapped to pthread id 0. The remaining pthread ids are linearly enumerated starting with the first thread on the first core as being 1. See the following tables below as examples.

### Table 2  Core to Pthread Mappings for a 57-Core Part

<table>
<thead>
<tr>
<th>APIC ID</th>
<th>Core</th>
<th>pthread ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2, 3</td>
<td>0</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>4, 5, 6, 7</td>
<td>1</td>
<td>5, 6, 7, 8</td>
</tr>
<tr>
<td>224, 225, 226, 227</td>
<td>...</td>
<td>57 &quot;BSP core&quot;</td>
</tr>
</tbody>
</table>

### Table 3  Core to Pthread Mappings for a 61-Core Part

<table>
<thead>
<tr>
<th>APIC ID</th>
<th>Core</th>
<th>pthread ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2, 3</td>
<td>0</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>4, 5, 6, 7</td>
<td>1</td>
<td>5, 6, 7, 8</td>
</tr>
<tr>
<td>240, 241, 242, 243</td>
<td>...</td>
<td>61 &quot;BSP core&quot;</td>
</tr>
</tbody>
</table>

The core to pthread mappings for a 61-core part looks like below:

<table>
<thead>
<tr>
<th>Core</th>
<th>pthread ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>1</td>
<td>5, 6, 7, 8</td>
</tr>
<tr>
<td>2</td>
<td>9, 10, 11, 12</td>
</tr>
<tr>
<td>3</td>
<td>13, 14, 15, 16</td>
</tr>
<tr>
<td>4</td>
<td>17, 18, 19, 20</td>
</tr>
<tr>
<td>5</td>
<td>21, 22, 23, 24</td>
</tr>
<tr>
<td>6</td>
<td>25, 26, 27, 28</td>
</tr>
<tr>
<td>7</td>
<td>29, 30, 31, 32</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>237, 238, 239, 240</td>
</tr>
<tr>
<td>61</td>
<td>0, 241, 242, 243</td>
</tr>
</tbody>
</table>

Hence if you are counting on pthread IDs 0, 1, 2, 3 being on the same core, this is not definitely the case with the Intel® Xeon Phi™ coprocessor system. So in general, when one calls sched_affinity, pthread_setaffinity_np, or sets the variable KMP_AFFINITY=proclist=[…], check to make sure that thread affinity is set to what is expected. Did one really mean to have KMP_AFFINITY=explicit, proclist=[0-243] or KMP_AFFINITY=explicit, proclist[1-243, 0]?
2.1.3.2 Setting Affinities on Linux*

It is entirely possible to “correct” for the enumeration as shown below:

```c
// for a 61 core part
#define MAX_THREADS 244
for (int i = 0; i < nthreads; i++) {
    cpu_set_t cpuset;   // (1) Create affinity mask
    CPU_ZERO (cpuset);  // (2) Clear all bits in affinity mask
    // (3) Calculate corrected affinity
    int corrected_affinity = (i+1)%MAX_THREADS;
    // (4) Set 1 bit in affinity mask, corresponding to thread# i
    CPU_SET (corrected_affinity, &cpuset);
    pthread_attr_setaffinity_np (attr, sizeof(cpu_set_t), &cpuset);
    pthread_create ( &tid[i], &attr, doWork, (void*) i );
}
```

where CPU_ZERO, CPU_SET are macros defined in sched.h header, and
pthread_attr_setaffinity_np is a part of pthread library (-lpthread).

2.1.3.3 The Linux* Boot Strap Processor (BSP)

On Linux*, there are some tasks that are specific to the BSP, pthread id 0 or the physical
hardware thread 240 (for a 61-core part). Compute-intensive applications, especially
ones that require barriers, can be adversely affected by this. There are ongoing
investigations on how to mitigate this issue. For now, one of the best known methods is
simply to stay off the BSP core if your application is impacted.

2.1.4 Avoiding Page Faults

Page faults impact application performance due to the cost in servicing that fault. Since
memory allocations are not automatically backed with physical pages, this is one of the
most common sources of page faults.

2.1.4.1 Memory Allocations in Linux*

All memory allocations are mapped down to a mmap call. The addresses get set aside,
but no physical pages have been allocated or faulted in. Thus a fault results on access to
this memory and could have detrimental performance impact especially if it is in the
performance critical section of the code. Even worse, DMA calls referencing this
memory will fail. The simple answer is to explicitly fault (“pre-faulting”) those memory
preferably outside of the critical code section.

2.1.4.2 Using MMAP

When using mmap, specifying the MAP_POPULATE flag will fault the memory
requested.
2.1.4.3 Using Other Allocators

When using other allocators, touch every page that is being allocated. When reading in data or when the values are initialized, the memories are already touched and no extra steps are required; reading and initialization will fault the pages.

The other option is to use the `memset()` call to set the memory to some default value which effectively will cause the memory to fault.

2.2 Considerations for Offload Applications

An Offload application in the Intel® Xeon Phi™ coprocessor context essentially follows the flow shown in Figure 2. The Offload compiler uses the Intel® Coprocessor Offload Infrastructure (Intel® COI) runtime which in turn calls Intel® Symmetric Communication Interface (SCIF) to perform the necessary lower level functions. This section will delve into these lower level components that need to be taken into consideration for optimal application performance.

![Figure 2 MIC Offload Application Flow](image)
2.2.1 Symmetric Communication Interface (SCIF)

SCIF is the communication backbone between the host processors and the Intel® MIC architecture devices in a heterogeneous computing environment. It is intended to provide communications capabilities within a single platform. SCIF enables communications between host and Intel® Xeon Phi™ coprocessor cards, and also between Intel® Xeon Phi™ coprocessor cards within the platform. It provides a uniform API for communicating across the platform’s PCI Express® system busses while delivering the full capabilities of the PCI Express® transport hardware. SCIF directly exposes the DMA capabilities of Intel® Xeon Phi™ coprocessor for high bandwidth transfer of large data segments, as well as the ability to map memory of the host or an Intel® Xeon Phi™ coprocessor device into the address space of a process running on the host or any Intel® Xeon Phi™ coprocessor device. Please refer to the Intel® Xeon Phi™ Coprocessor System Software Developers Guide (https://www.ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html) for more details.

2.2.1.1 Data Transfer Mechanisms

There are a couple of ways to move data between the host and the Intel® MIC device – CPU copy and using the DMA engines. For large data buffers, DMA is good since the set-up cost gets “amortized.” But for small buffers, the overhead of programming DMA plus ring transition for the driver could result in unacceptable effective bandwidth or latency. For small data transfer, low latency mechanisms exist (e.g. scif_mmap).

The initiator of the DMA transfer also matters – e.g. host vs. MIC. Since the host Intel® Xeon® processors are generally more powerful running a single threaded process, it can thus program the DMA engine faster.

In general, getting the host Intel® Xeon® processor to initiate or perform a write data transfer to the MIC device gives better performance for the following reasons: faster programming of the DMA engine, write is faster than read, and the Intel® MIC address space is already mapped to the host CPU memory.

2.2.1.2 Buffer Alignment

The alignment of source and destination buffer matters a lot when it comes to data transfer performance. There are 3 possible scenarios that will be illustrated below:

1) Source and destination buffers are cacheline aligned and mutually aligned – meaning the source and destination have the same offset from the cacheline boundary.

2) Source and destination buffers are both not aligned to a cacheline boundary but are still mutually aligned.

3) Source and destination buffers are both not aligned to a cacheline boundary and are no longer mutually aligned – source and destination have different offset to the cacheline boundaries.

The first case (see Figure 3) obviously gives the best performance (see Figure 4).
Figure 3  Case 1 where source and destination buffers are cacheline aligned and mutually aligned

Figure 4  Case 1 data transfer bandwidth at 8K and 16K payload
The second case where the source and destination buffers not aligned to a cacheline boundary but are still mutually aligned will incur additional overhead to copy the leading bytes before DMA kicks-in to move the remaining cacheline aligned data. In Figure 5, 48 bytes of data will have to be moved by a `memcpy()` call. The bandwidths for lower payloads are degraded as it amortizes the `memcpy()` overhead cost (Figure 6). Each `memcpy()` operation happens on the “local side” from source to shadow, and involves an `ioremap()` to map the remote buffer before the CPU copy to the remote end.

![Case 2 diagram](image)

**Figure 5**  Case 2 where source and destination buffers are not cacheline aligned but are mutually aligned
Finally, the third case incurs an additional overhead and gives the poorest performance. When the source and destination buffers are not cacheline aligned and also not mutually aligned, an extra step is involved to convert it to a mutually aligned situation (Case 2) by creating a “shadow buffer” (Figure 7) which incurs another memcpy() operation.
As Figure 8 shows, the added overhead in creating a “Shadow Buffer” to convert it into Case 2 furthers degrade the performance as clearly shown for both the 8K and 16K payload. It is not until the payload size gets to about 1MB before the bandwidth recovers and the overhead costs have been effectively amortized.

2.3 Power Management and NUMA

2.3.1 Power Management

Power Management on the platform, code named Romley can affect the performance of data transfers. Certain power management states such as C1E detect that a socket is idle, and once detected throttle the CPUs on the socket to their minimum value as well as scaling the voltage down as well. There can be a situation where a data transfer is initiated and then the calling thread blocks waiting for an interrupt to signal the completion of the transfer. Since the host CPU is idle while waiting the system detects this and enters C1E. The frequency throttling also affects the un-core where the memory controller resides and therefore effects the DMA operation already in progress. A similar problem can occur when the Linux* on demand CPU frequency scaling driver detects idle CPUs and steps down the frequency trying to conserve power. There are 2 steps that can be taken to mitigate this issue.
Busy wait – When a transfer is initiated it is possible to spin on a volatile memory location which will be written to when the transfer completes. This is currently the default behavior for the offload runtime. This keeps the calling CPU busy and keeps the CPU from entering C1E. This is achieved in the SCIF API by using the scif_fence_signal API. In the COI API this can be achieved by passing 0 to the time out parameter in COIEventWait. The offload compiler does this by default, but can be overridden by defining the environment variable OFFLOAD_ACTIVE_WAIT=0.

Performance governor - Enabling the Linux* performance scaling governor will mitigate any issues caused by the on demand performance governor. The performance scaling governor keeps the CPUs at maximum (non-turbo) frequency when not in C1E. Since the prior step keeps the CPUs out of C1E, this guarantees the CPUs and un-core are operating at maximum frequency during transfers. The following steps will enable the Linux* performance governor on Red Hat* Enterprise Linux* 6 as well as SUSE* Enterprise Linux* 11 distributions (SLES 11):

```
user_prompt> sudo modprobe acpi-cpufreq
user_prompt> echo "performance" | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

2.3.2 NUMA Issues

**Card to Host/CPU Transaction**

**Local**

![Diagram of Card to Host/CPU Transaction](image)

*Figure 9 Intel® Xeon Phi™ coprocessor attached to local socket*

Further PCIe performance issues can arise as a result of NUMA (non-uniform memory access) configuration. When the Intel® Xeon Phi™ coprocessor is attached to the local socket as in *Figure 9*, maximum performance can be achieved. However, if the calling process is on a socket remote to the Intel® Xeon Phi™ coprocessor, as in *Figure 10*, noticeable performance degradations can occur.
The solution to this is to pin the calling thread to whichever socket is local to the Intel® Xeon Phi™ coprocessor. This can be done in a variety of ways. It can be accomplished using affinitization with pthreads, Cluster OpenMP* product, or Intel® MPI Library. Linux® also provides cpuset and numactl commands to pin processes to certain CPUs or sockets. Since this is dependent on the physical location of the Intel® Xeon Phi™ coprocessor in the host system, the correct CPU/socket to bind to will differ by system configuration.