Symmetric Communications Interface (SCIF) For Intel® Xeon Phi™ Product Family Users Guide

Revision 1.03

February 2014

General Notices and Disclaimers

Benchmark and Performance Disclaimers

Table of Contents

1 Introduction

1.1 About This User Guide

This user guide describes the Symmetric Communication Interface (SCIF) for the Intel® Xeon Phi™ Product Family. SCIF is a component of the Intel® ManyCore Platform Software Stack (MPSS). The goal of this document is to present SCIF concepts and usage. Refer to the SCIF header file, scif.h, and the SCIF man pages for detailed information on the SCIF API.

1.2 Target Audience

The target audience includes tools developers and application developers. After reading this document, the reader will be able to use the SCIF interface for communication between the components of a distributed application.

1.3 Related Documents

Document Title	Revision Number	Availability
MPI overview and specification		http://www.mpi-forum.org/
OFED* overview		http://www.openfabrics.org/OFED-Overview.html

1.4 Terminology and Acronyms

Term	Description
API	Application Programming Interface
HCA	(Infiniband) Host Channel Adapter
MIC	Intel® Many Integrated Core
MPSS	ManyCore Platform Software Stack
OFED	Open Fabrics Enterprise Distribution
RMA	Remote memory access
RDMA	Remote direct memory access

2 Product Overview

2.1 Goals and Objectives

SCIF provides a mechanism for inter-node communication within a single platform, where a node is an Intel® Xeon Phi™ coprocessor or an Intel® Xeon® host processor complex. In particular, SCIF abstracts the details of communicating over the PCIe bus while providing an API that is symmetric between the host and MIC Architecture devices. An important design objective for SCIF was to deliver the maximum possible performance given the communication capabilities of the hardware.

2.1.1 Portability and Platform Independence

The Intel® MIC software architecture supports a computing model in which the workload may be distributed across both the Intel® Xeon® host processor complex and Intel® MIC Architecture coprocessors. An important property of SCIF is symmetry; SCIF drivers must present the same interface on both the host processor and the Intel® MIC Architecture coprocessor in order that software written to SCIF can be executed wherever is most appropriate.

Since the Intel® MIC Architecture coprocessor may use a different operating system than that running on the host, the SCIF architecture is designed to be operating system independent. This ensures SCIF implementations on different operating systems can inter-communicate.

2.2 Product Environment

As mentioned earlier, the Intel® MIC software architecture supports a computing model in which the workload is distributed across both Intel® host processors and Intel® MIC Architecture coprocessors.

2.2.1 Hardware Environment

SCIF supports communication between Xeon host processors and Intel® MIC Architecture coprocessors within a single platform. Communication between such components that are in separate platforms can be performed using standard communication channels such as Infiniband and TCP/IP.

2.2.2 Software Environment

A SCIF implementation on a host or Intel® MIC Architecture coprocessor includes both a user mode (Ring 3) library and kernel mode (Ring 0) driver as shown in Figure 1. Most of the components in the Intel® MPSS use SCIF for communication. Refer to the *Intel® Xeon Phi™ Coprocessor (codename: Knights Corner) Software Developers Guide* for a discussion of the other components in the Intel® MPSS and their relationship to SCIF.

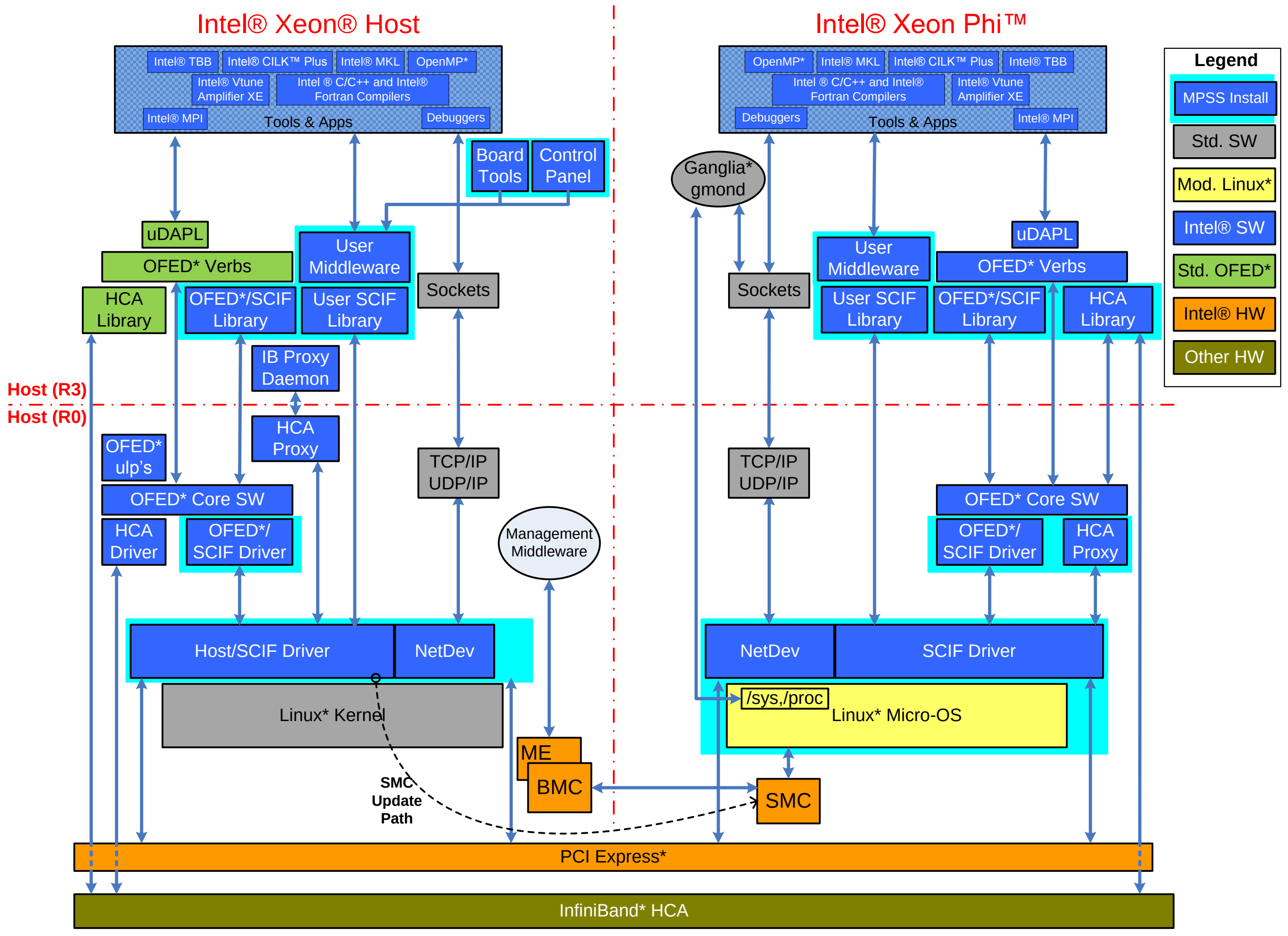

Figure 1: Intel® ManyCore Platform Software Stack (MPSS)

3 SCIF Programming Concepts

The SCIF driver provides a reliable connection-based messaging layer, as well as functionality which abstracts RMA operations. In the following sections we describe these architectural concepts in some detail. The SCIF API is documented in the SCIF header file, scif.h, and the SCIF man pages. A common API is exposed for use in both user mode (ring 3) and kernel mode (ring 0), with the exception of slight differences in signature, and several functions which are only available in user mode, and several only available in kernel mode.

3.1 Nodes

A *SCIF node* is a physical endpoint in the SCIF network. The host and MIC Architecture devices are SCIF nodes. From the SCIF point of view, all host processors (CPUs) under a single OS are considered a single SCIF (host) node.

We generally use "node" instead of "SCIF node" where this will not cause confusion.

Each node in the SCIF network has a node identifier that is assigned when the platform is booted. Node IDs are generally based on PCIe discovery order and, thus, may change across a platform reboot, however the host node is always assigned ID 0.

3.2 Ports

A *SCIF port* is a logical destination on a SCIF node. We generally use "port" rather than "SCIF port". Within a node, a SCIF port on that node may be referred to by its number, a 16-bit integer. This is analogous to an IP port; for instance, SSH usually talks over TCP port 22. We sometimes use "local port" to refer to a port that is on the same node as a particular point of reference.

A *SCIF port identifier* is unique across a SCIF network, comprising both a node identifier and a local port number. A SCIF port identifier is analogous to a complete TCP/IP address (for instance 192.168.1.240:22).

Analogous to Internet sockets, some ports may be "well-known", and monitored by service daemons launched with the local OS or later. Any such services are layered on SCIF and thus beyond the scope of this document.

3.3 Endpoints and Connections

The entity through which a port is accessed is called an *endpoint*. An endpoint can be *listening*, i.e. waiting for a connection request from another endpoint, or *connected*, i.e. able to communicate with a remote connected endpoint. A connection is an association established between two endpoints for the purpose of communication. The following functions are used during the connection process:

```
scif_epd_t scif_open(void);
int scif_bind(scif_epd_t epd, uint16_t pn);
int scif_listen(scif_epd_t epd, int backlog);
int scif_connect(scif_epd_t epd, struct scif_portID* dst);
int scif_accept (scif_epd_t epd, struct scif_portID* peer, scif_epd_t*
    newepd, int flags);
int scif_close (scif_epd_t epd);
```

The process for establishing a connection is similar to socket programming: A process calls `scif_open`() to create a new endpoint; `scif_open`() returns an endpoint descriptor that is used to refer to the endpoint in subsequent SCIF function calls. The endpoint is then bound to a port on the local node using `scif_bind`(). An endpoint which has been opened and bound to a port is made a listening endpoint by calling `scif_listen`(). To create a connection, a process opens an endpoint and binds it to a local port, and then requests a connection by calling `scif_connect`(), specifying the port identifier of some listening endpoint, usually on a remote node. A process on the remote node may accept a pending or subsequent connection request by calling `scif_accept`(). `scif_accept`() can conditionally return immediately if there is no connection request pending, or block until a connection request is received.
The `select`() and `poll`() functions can be used from Linux* user mode to determine when a connection request has been received on any of a set of listening endpoints. The `scif_poll`() function may be used from Linux* user and kernel modes, and from Microsoft Windows* user mode for this purpose.

When the connection request is accepted, a new connected endpoint is created, bound to the same port as the listening endpoint. The requesting endpoint and the new endpoint are now connected endpoints that form the connection. The listening endpoint is unchanged by this process. Multiple connections may be established to a port bound to a listening endpoint.

The following figure illustrates the connection process. In this example, a process on node i calls `scif_open`(), which returns endpoint descriptor epd_i. It then calls `scif_bind`() to bind the new endpoint to local port pm, and then calls `scif_connect`() requesting a connection to port `pn` on node j. Meanwhile, a process on node j calls `scif_open`(), getting back endpoint descriptor epd_j, binds the new endpoint associated with epd_j to local port pn, and calls `scif_listen`() to mark the endpoint as a listening endpoint. Finally, it calls `scif_accept`() to accept a connection request. In servicing the connection request, `scif_accept`() creates a new endpoint, with endpoint descriptor `nepd`, which is the endpoint to which epd_i is connected. The endpoints associated with epd_i `and nepd` are now connected endpoints and may proceed to communicate with each other. The listening endpoint associated with epd_j remains a listening endpoint and may accept an arbitrary number of connection requests.

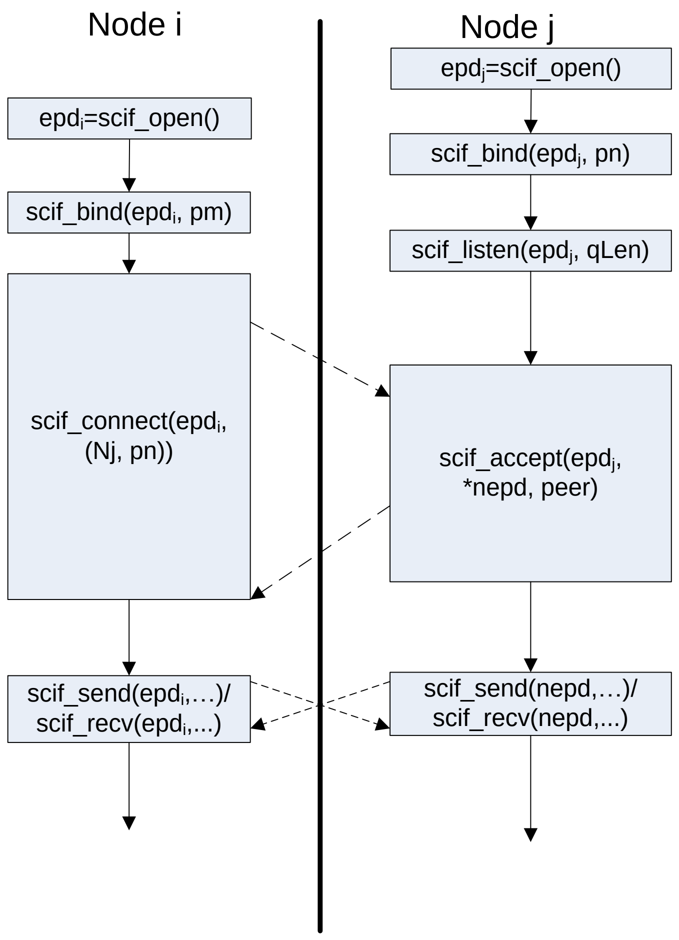
Node i
Node j
epd$_i$=scif_open()
scif_bind(epd$_i$, pm)
scif_connect(epd$_i$, (Nj, pn))
scif_send(epd$_i$,…)/ scif_recv(epd$_i$,...)
epd$_j$=scif_open()
scif_bind(epd$_j$, pn)
scif_listen(epd$_j$, qLen)
scif_accept(epd$_j$, *nepd, peer)
scif_send(nepd,…)/ scif_recv(nepd,...)

Figure 2: Connecting two endpoints

Normally the endpoints of a connection are on different nodes in the SCIF network. We therefore often refer to these endpoints as *local* and *remote* with respect to one end of the connection. In fact, SCIF fully supports connections in which both endpoints are on the same node, and we refer to this as a *loopback* connection.

A process may create an arbitrary number of connections, limited by system resources (memory). The following figure illustrates a SCIF network of three nodes. Two connections have been established between nodes 0 and 1, another between nodes 0 and 2. On node N2, a loopback connection has been established.

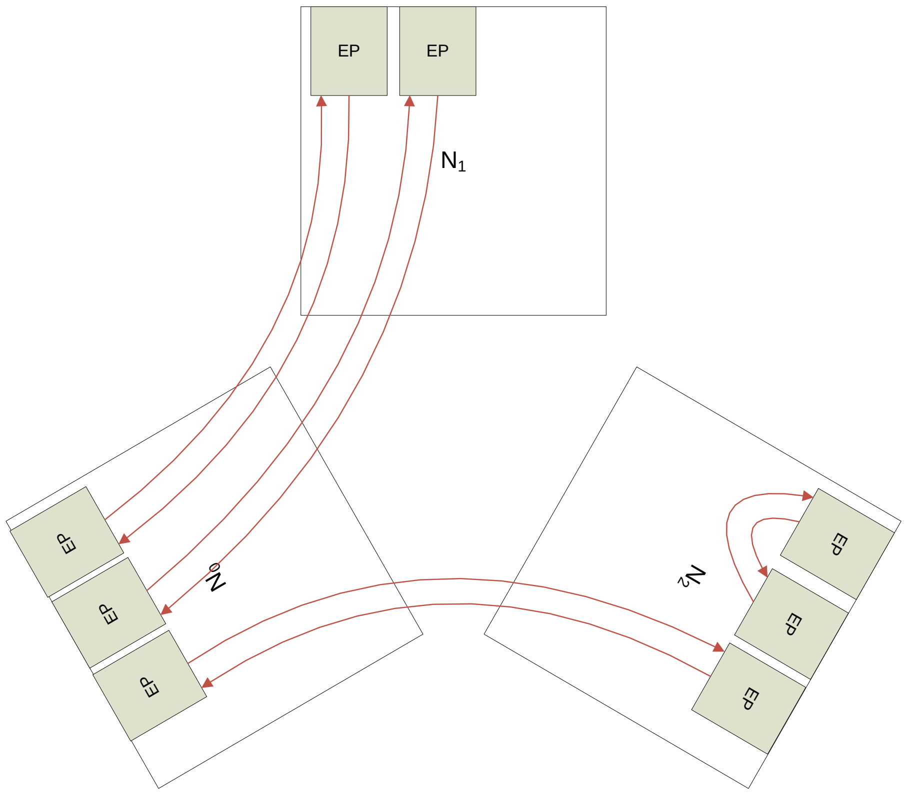

Figure 3: Connected endpoints

The endpoint pair comprising the connection are *peer endpoints* or just *peers*. Similarly, the processes which own the peer endpoints are *peer processes*, the node on which a peer endpoint resides is a *peer node*, and so on.

3.4 Messaging Layer

After a connection has been established, messages may be exchanged between the processes owning the connected endpoints. A message sent into one connected endpoint is received at the other connected endpoint. Such communication is bi-directional. The following functions comprise the messaging layer:

```
int scif_send(scif_epd_t epd,void* msg,int len,int flags);
int scif_recv(scif_epd_t epd,void* msg,int len,int flags);
```

Messages are always sent through a local endpoint for delivery at a remote connected endpoint. For each connected pair of endpoints, there is a dedicated pair of message queues – one queue for each direction of communication. In this way, the forward progress of any connection is not gated by progress on another connection, which might be the case were multiple connections sharing a queue pair.

A message may be up to 2^{31}-1 bytes long. In spite of this, the messaging layer is intended for sending short command-type messages, not for bulk data transfers. The messaging layer queues are relatively short; a long message is transmitted as multiple shorter queue-length transfers, with an interrupt exchange for each such transfer. Therefore it is strongly recommended that SCIF RMA functionality be used for sending larger units of data, e.g. longer than 4KiB.

Messages on any connection are received in the order in which they are sent. There are no guarantees regarding the order in which messages sent on different connections are received. Moreover, the PCIe bus is assumed to be a reliable transport. Therefore, SCIF makes no attempt to detect or correct lost or corrupted messages.

The content of a message is not interpreted by the messaging layer, and has meaning only to the sending and receiving processes. Therefore it is the responsibility of the application to impose any required structure or protocol.

The messaging layer supports both blocking and non-blocking behaviors. A blocking call to the `scif_send`() function will block (not return) until the entire message has been sent. A non-blocking call to the `scif_send`() function only sends as much data as there is room in the send queue at the time of the call. In both cases, the number of bytes sent is returned as the result of the call. The `select`() and `poll`() functions can be used from Linux* user mode to determine when it is possible to send more data on any of a set of connected endpoints. The `scif_poll`() function may be used from Microsoft Windows* and Linux* kernel mode, and from Microsoft Windows* user mode for this purpose.

Similarly, a blocking call to the `scif_recv`() function will block until all `len` bytes (where `len` is a parameter specifying the number of bytes to receive) have been received and copied to the application's buffer. A non-blocking call to the `scif_recv`() function only returns data that is currently in the receive queue (up to some application-specified maximum number of bytes). In both cases, the number of bytes received is returned as the result of the call. The `select`() and `poll`() functions can be used from Linux* user mode to determine when more data is available on any of a set of connected endpoints. The `scif_poll`() function may be used from Microsoft Windows* and Linux* kernel modes, and from Microsoft Windows* user mode for this purpose.

3.5 Memory Registration

Memory registration is the mechanism by which a process exposes ranges of its address space for controlled access by another process, typically a process on a remote node. Memory must be registered before it can be mapped to the address space of another process or be the source or target of an RMA transfer.

Each connected endpoint has a *registered address space*, a kind of address space managed by the SCIF driver, ranges of which can represent local physical memory. The registered address space is sparse in that only specific ranges which have been registered, called *registered windows* or just *windows*, can be accessed. It is an application error to attempt to access any range of a registered address space which is not within such a window.

We use the term *offset* to mean a location in a registered address space in analogy to the mapping from virtual address space to a shared memory object established by the Posix `mmap`() function. In the Posix `mmap`() function, an offset parameter specifies the offset, from the beginning of the memory object, of the range onto which the virtual address range is mapped. Essentially an offset is an address in some registered address space, therefore we sometimes talk about a *registered address*.

The following functions support registration:

```
off_t scif_register(scif_epd_t epd, void* addr, size_t len, off_t
      offset, int prot_flags, int map_flags);
int scif_unregister(scif_epd_t epd, off_t offset, size_t len);
```

The `scif_unregister`() function and window deletion is discussed in a later section.

[In this and subsequent sections, we talk about *ranges* in virtual and registered address spaces. The reader should understand that these are specified by the (`addr`,`len`) and (`offset`,`len`) parameter pairs respectively. Note, also, that registration granularity is 4KiB (a "small" page). Therefore `addr`, `offset` and `len` parameters to `scif_register`() must be multiples of 4KiB.]

The `scif_register`() function establishes a mapping between a range in the registered address space of some connected endpoint of the calling process and a set of physical pages. The physical pages are indirectly identified by specifying a range in the *user* virtual address space of the calling process. The mapping, then, is from the specified range in some registered address space to the physical pages which back the specified virtual address range. Note that this mapping between registered address space and physical memory remains even if the specified virtual address range is unmapped or remapped to some different physical pages or object.

In the following figure, the left diagram illustrates a registered window, W, at the time of its creation by `scif_register`(). The pages of W, a range in the registered address space of some local endpoint, represent some set, P1, of physical pages in local memory. P1 is the set of physical pages which backed a specified virtual address range, VA, at the time that `scif_register`() was executed. Even if the virtual address range, VA, is subsequently mapped to different physical pages P2 (right panel of Figure 4), W continues to represent P1. Of course, the process now has no way to access the registered memory in order to read or write RMA data unless those physical pages back some other virtual address range.

For simplicity, we show P1 and P2 as contiguous ranges in physical memory, whereas they may be discontiguous.

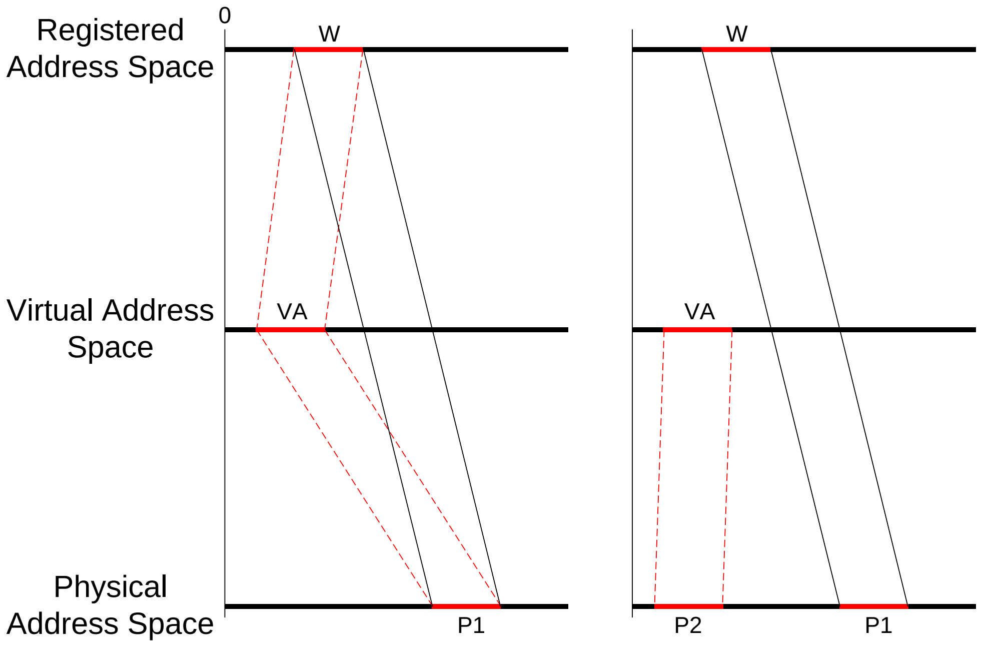

Figure 4: Registration mapping to memory objects

Though a window is a mapping in the mathematical sense, we generally say that the registered address space range of a window *represents* the corresponding physical pages. This is intended to avoid confusion with mappings created by `scif_mmap`() or `mmap`()) described later.

The physical pages which a window represents are pinned (locked) in memory so that they can be accessed from a remote SCIF node. Therefore it is an error to specify a virtual address range to `scif_register`() for which the backing pages cannot be pinned for whatever reason. The pages which a window represents remain pinned as long as the window exists. As will be explained below, a physical page may be represented by more than one window. Such a page will remain locked until all such windows are unregistered.

The `scif_unregister`() function is used to delete one or more windows and is discussed in more detail later.

Figure 5 illustrates several registered window configurations. It shows the physical space of a node which has two connected endpoints, possibly owned by different processes. Each endpoint has an independent registered address space associated with it (for simplicity, we do not illustrate the virtual memory ranges which the physical ranges back).

- Windows W1a and W2a represent the same physical address range but have different offsets in their respective registered address spaces.
- W1b and W2b have the same offset (the light gray dashed lines help show this) but represent different physical address ranges.
- W1c and W1d are disjoint windows in the same registered address space, but represent overlapping physical address ranges.

The extra degree of freedom offered by registered address spaces may be useful for solving various communication and programming problems.

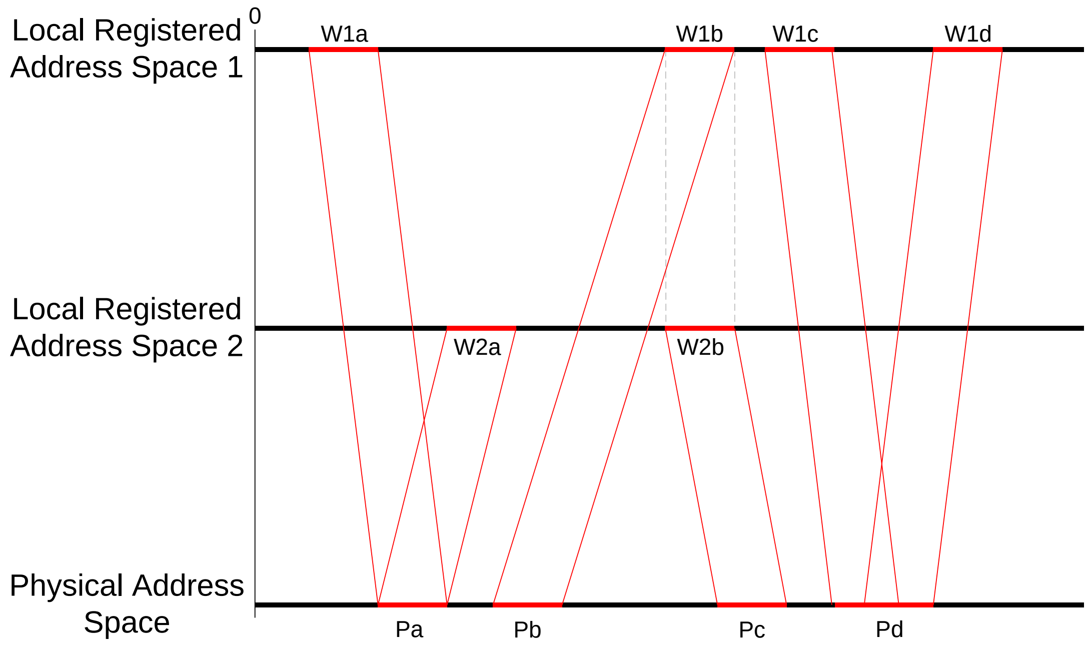
Local Registered Address Space 1
0
W1a
W1b
W1c
W1d
Local Registered Address Space 2
W2a
W2b
Physical Address Space
Pa
Pb
Pc
Pd

Figure 5: Registered window configurations

We refer to a window in the registered address space of the peer of a local endpoint as a *remote window*. Every window in the registered address space of a local endpoint is a remote window to the peer endpoint. Several SCIF functions (`scif_readfrom`(), `scif_writeto`(), `scif_vreadfrom`(), `scif_vwriteto`(), `scif_mmap`(), and `scif_get_pages`()) access remote windows or portions thereof, and specified as an offset and length in the registered address space of the peer of a specified local endpoint.

The management of a registered address space can be performed by SCIF, by the application or both, and is controlled by the `map_flags` parameter to `scif_register`(). When SCIF_MAP_FIXED is set in `map_flags`, SCIF attempts to allocate the window at the registered address specified in the offset parameter. Otherwise, SCIF selects a registered address at which to allocate the window.

In Figure 6 the application has create three windows at offsets 0x1000, 0x3000 and 0x5000 respectively (by passing the SCIF_MAP_FIXED flag), each 0x1000 bytes long. If these offsets are coded in the peer application, then it knows the offsets to use to access these windows, for example in performing an RMA.

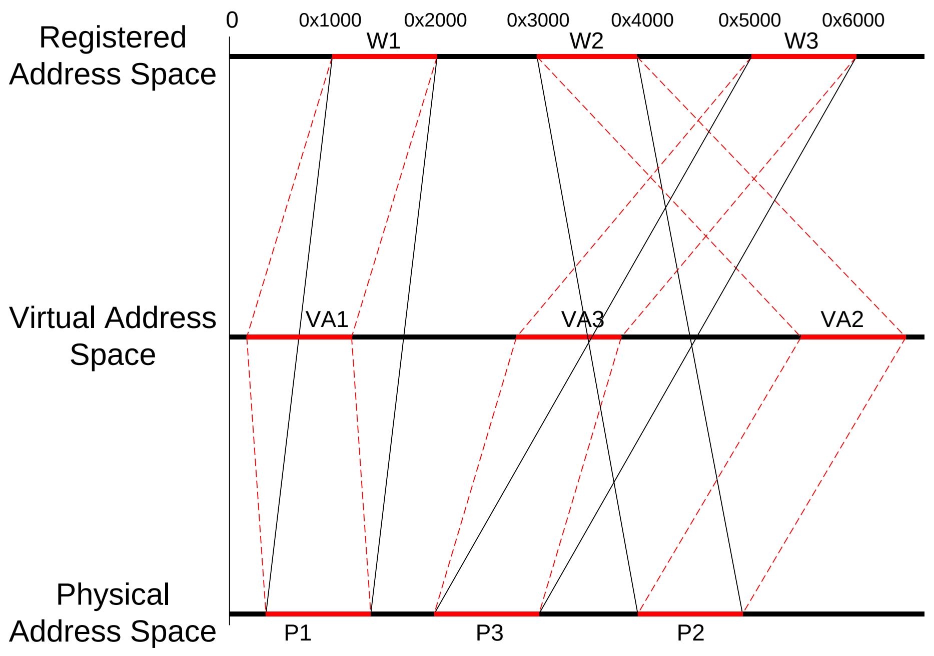

Figure 6: Hard coded registered addresses

As an alternative, an application can use the virtual address as the offset when registering a window. In this way the application need not "remember" the offset of the window corresponding to some virtual address. This is illustrated in Figure 8.

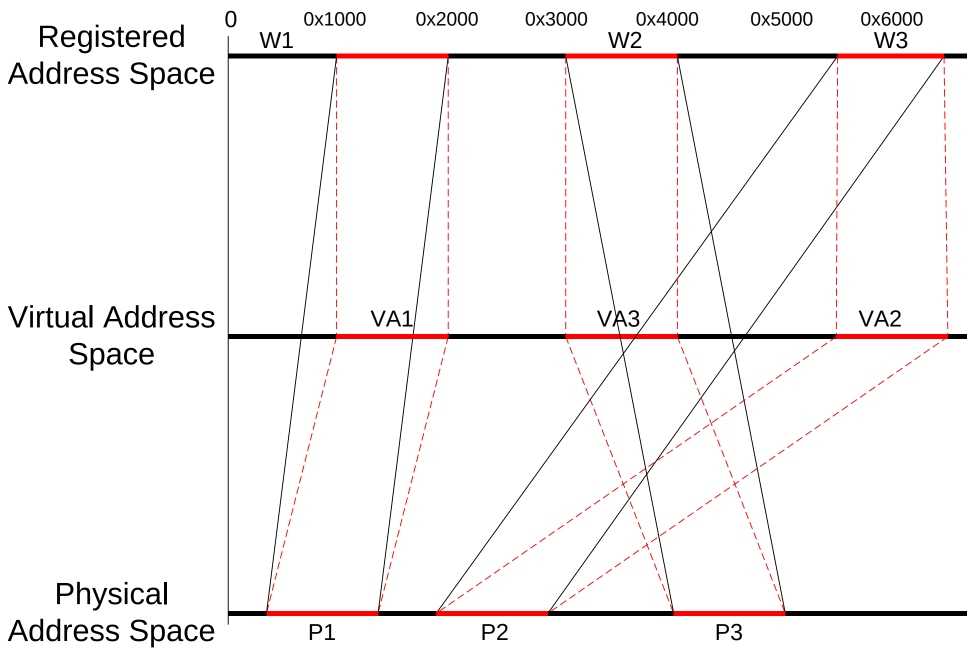

Figure 7: Registered addresses same as virtual addresses

The `scif_register()` function also takes a `prot_flags` parameter which controls access to the window being registered. The SCIF_PROT_READ flag marks a window as allowing read operations; specifically the window can be the source of an RMA operation. Similarly the SCIF_PROT_WRITE flag marks a window as allowing write operations; specifically the window can be the destination of an RMA operation.

The `scif_mmap()` function (described more fully later) also takes a `prot_flags` parameter. The SCIF_PROT_READ flag indicates that the mapped region is to be readable; it is an error if the referenced window was not also registered with the SCIF_PROT_READ flag. Similarly the SCIF_PROT_WRITE flag indicates that the mapped region is to be writable; it is an error if the referenced window was not also registered with the SCIF_PROT_WRITE flag.

These flags only control access to windows; they do not control access to the physical pages which a window represents where those pages back virtual memory. Thus, referring back to Figure 4, the process which registered window W has access to the pages P1 through the virtual addresses VA regardless of the protections on window W. Similarly, once a (portion of a) window has been mapped using scif_mmap(), the application may read or write to the mapped physical pages regardless of the `prot_flags` specified when scif_mmap() was called. Referring ahead to Figure 9, the process which mapped a range, RR, of remote window RW into a range of its address space at VA, can both read and write to pages P through VA, regardless of the value of `prot_flags`.

3.5.1 Duplication of Endpoint Descriptors Across a fork()

On Linux*, an endpoint is implemented as a file description, and an endpoint descriptor as a file descriptor. If an application opens an endpoint and then fork()'s, the parent and child will each have an endpoint descriptor (file descriptor) which refers the same endpoint. The parent and child then share the registered address space of this endpoint. Consider the following scenario:

```
Parent:
scif_epd_t epd = scif_open();
scif_connect(epd, pn);
fork();
off_t po =
scif_register(epd,addr1,0x1000,
   0x10000,3,0);
scif_readfrom(epd,0x20000,len1,
   roff1,flags);
```

```
Child:



off_t po =
scif_register(epd,addr2,0x1000,
   0x20000,3,0);
scif_readfrom(epd,0x10000,len2,
   roff2,flags);
```

After the fork(), both the parent and child have an endpoint descriptor, epd, which refers to the endpoint created by the parent. The parent now registers a window at offset 0x10000 that represents the physical page backing the page at its addr1. Similarly the child registers a window at offset 0x20000 that represents the physical page backing the page at its addr2. Because both windows are in the same registered address space, the child can access the parent's memory and vice versa. That is, any memory registered to this endpoint is shared by the two processes. For example, each can initiate an RMA which transfers data into the shared pages. This behavior, while perhaps surprising, is consistent with fork() semantics regarding duplication of file descriptors.

3.5.2 Registered Memory Across a fork()

Linux*' copy-on-write semantics mean that, following a fork(), both the parent and child process will have page table entries pointing to the same physical pages. Because those pages are write protected, when one of the processes, either parent or child, writes to a page, the hardware will trap the write event. The kernel will respond by allocating a new page and copying the contents from the original page, breaking the linkage to the physical page for that process.

Consider the case that a process registers a window and then fork()'s. Suppose the parent now writes directly to a virtual address corresponding to a page of the window; it will be allocated a new physical page. However, subsequent RMAs to or from the window offset that corresponds to that virtual address will, however, access the original physical page at the time of registration, not the newly allocated physical page; the physical pages that the window represents are unchanged. Thus, data which the parent process writes to the newly allocated page will not be sent when a `scif_writeto`() RMA is performed, and data received when a `scif_readfrom`() RMA is performed will not be read by the process.

To prevent this from happening, it is recommended that the parent mark the virtual address range of a registered window as MADV_DONTFORK, if the process will fork() after performing the registration. Doing this prevents the virtual address range from being seen by the child, so it is only seen by the parent, and copy-on-write semantics do not apply to that range.

A similar problem can occur if a process registers a window after a fork() in which the virtual address range was allocated before the fork(), since that virtual address range might now be subject to copy-on-write semantics. There are several possible solutions to this problem:

- Mark the virtual address range to be registered as MADV_DONTFORK before the fork(). The virtual address range will now only be available for registration by the parent.
- (After the fork…) In one or the other process, write to all the pages of the range to force new pages to be allocated

3.5.3 Kernel Mode Registration-Related API

Several additional functions are available in kernel mode to solve specific programming requirements:

```
int scif_pin_pages(void* addr, size_t len, int prot_flags, int
    map_flags, scif_pinned_pages_t* pages);
int scif_unpin_pages(scif_pinned_pages_t pinned_pages);
off_t scif_register_pinned_pages(scif_epd_t epd, scif_pinned_pages_t
    pinned_pages, off_t offset, int map_flags);
```

`scif_pin_pages`() pins the set of physical pages which back a range of virtual address space, and returns a handle which may subsequently be used in calls to `scif_register_pinned_-pages`() to create windows which represent the set of pinned pages. The windows so created are otherwise identical to windows created by `scif_register`(). The handle is freed by `scif_unpin_pages`(), but the physical pages themselves remain pinned as long as there is a window which represents the pages. Unlike `scif_register`() which interprets the address passed it as a user space address, `scif_pin_pages`() interprets the address passed it as a kernel space address if the `map_flags` parameter has the SCIF_MAP_KERNEL flag.

Figure 8 illustrates this process. In the leftmost panel, `scif_pin_pages()` pins the set of physical pages, P1, which back some range, VA, of virtual address space. In the center panel, a window, W1, is registered, using `scif_register_pinned_pages()`, at some offset in some Registered Address Space 1, and represents the physical pages P1. In the rightmost panel, a second window, W2, is registered, again using `scif_register_pinned_pages()`, at some offset in some Registered Address Space 2, and also represents the physical pages P1. At the same time, the mapping of VA has been changed to the set of physical pages, P2, but windows W1 and W2 continue to represent P1.

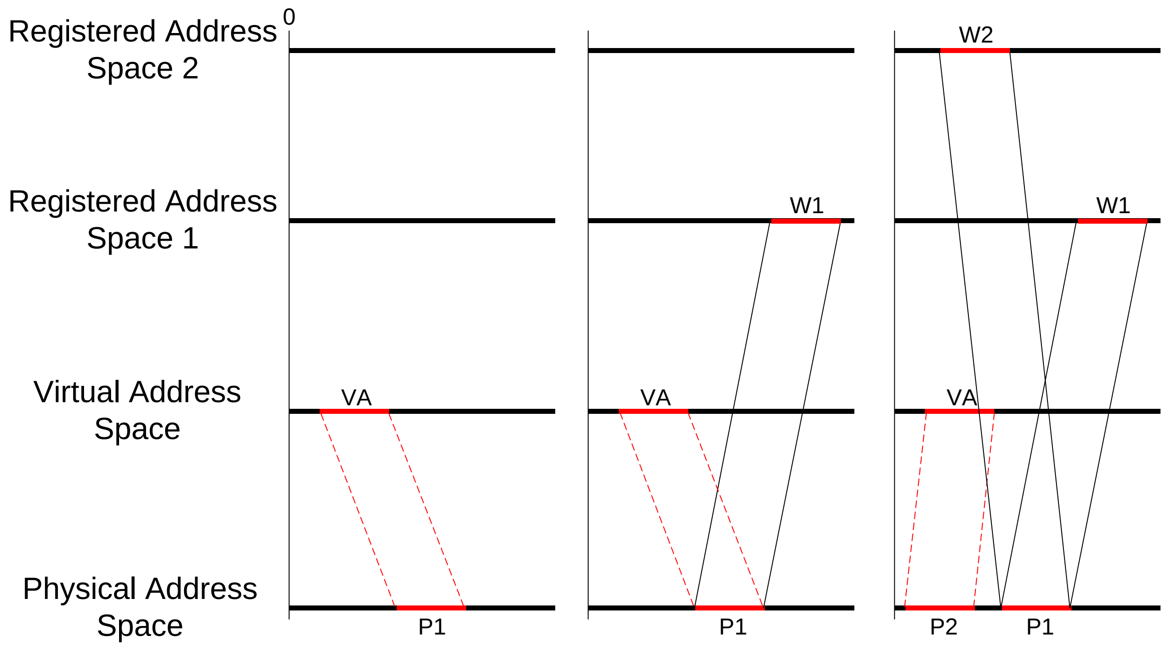

Figure 8: Registering windows using scif_pin_pages()

3.6 Mapped Remote Memory

The SCIF mapping functions enable mapping some physical memory on a remote node into the virtual address space of a process. Once established, a read or write access to such a mapped range of virtual address space will read or write to the corresponding mapped physical memory location. The mapping functions are:

```
void* scif_mmap(void* addr, size_t len, int prot_flags, int map_flags,
    scif_epd_t epd, off_t offset);
int scif_munmap (void* addr, size_t len);
```

Note that these functions are only available in the user mode API.

The mapping established by a `scif_mmap()` operation is illustrated in the following figure:

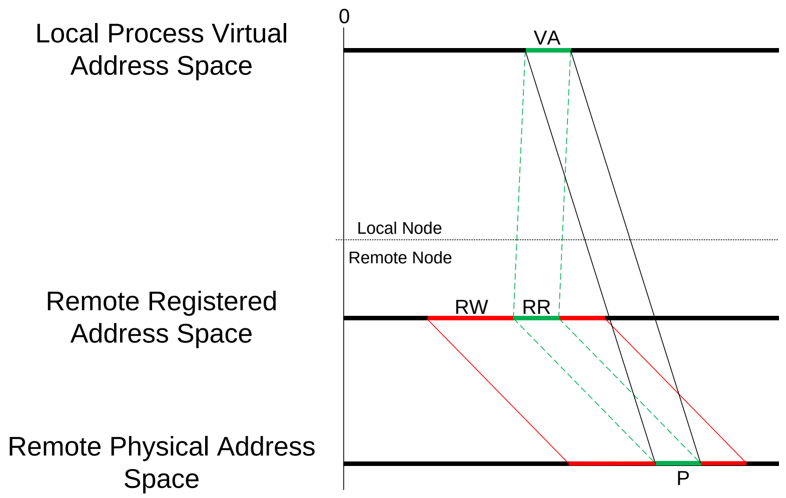
Local Process Virtual Address Space
0
VA
Local Node
Remote Node
Remote Registered Address Space
RW
RR
Remote Physical Address Space
P

Figure 9: Address space mapping of `scif_mmap()`

The process performing the `scif_mmap`() operation specifies a range, VA, within its local virtual address space, and a corresponding range, RR, of the same length within a peer remote registered address space. The composition of the mapping from VA to RR and the mapping from RR to P, the set of physical pages represented by RR, defines a mapping (black lines) from VA to P. `scif_mmap`() modifies the page table of the process according to this mapping. Hence, reads from and writes to VA will actually read from or write to corresponding locations in the physical pages P.

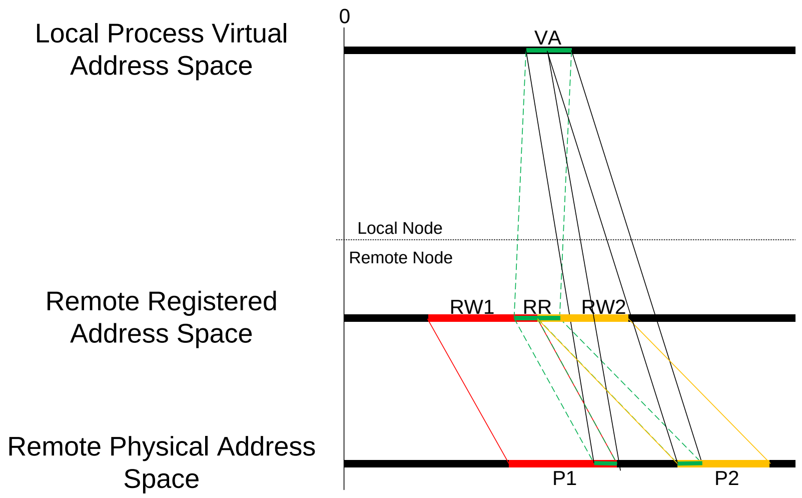
Local Process Virtual Address Space
0
VA
Local Node
Remote Node
Remote Registered Address Space
RW1
RR
RW2
Remote Physical Address Space
P1
P2

Figure 10: Virtual address space mapping that intersects multiple windows

The remote registered address range may not intersect any portion of the remote virtual address space which is not within a window, but may intersect multiple remote windows. Therefore those multiple windows must be contiguous in their registered address space. In Figure 10, RR intersects windows RW1 and RW2, which represent physical memory ranges P1 and P2 respectively. Thus access to an address in VA will be vectored to a page in P1 or P2 depending on whether the address in VA maps to RW1 or RW2.

While a remote mapping exists, the remote pages remain pinned and available for access, even if the peer endpoint referenced when the mapping was created is closed, either explicitly or because the peer process is killed. `scif_munmap()` unmaps some range of pages in the callers address space. Subsequent access to such virtual pages results in a segmentation fault. `scif_munmap()` does not take an endpoint parameter; if a page in the specified range was not mapped using `scif_mmap()`, the effect will be as if `mmap()` was called on that page.

3.6.1 Kernel Mode Mapping-Related API

The kernel mode API provides a similar capability to scif_`mmap()` through the `scif_get_-pages()` and `scif_put_pages()` functions. `scif_get_pages()` takes a range in some remote window and returns a structure listing the physical addresses of pages which are represented by the registered address space range. Those physical pages will continue to be available until the structure obtained from `scif_get_pages()` is returned in a call to `scif_put_pages()`.

3.7 Remote Memory Access

SCIF RMA operations are intended to support the one-sided communication model which has the advantage that a read/write operation can be performed by one side of a connection when it knows both the local and remote locations of data to be transferred. One-sided calls can often be useful for algorithms in which synchronization would be inconvenient (e.g. distributed matrix multiplication), or where it is desirable for tasks to be able to balance their load while other processors are operating on data.

The following functions comprise the RMA group:

```
int scif_readfrom(scif_epd_t epd, off_t loffset, size_t len, off_t
    roffset, int rma_flags);
int scif_writeto(scif_epd_t epd, off_t loffset, size_t len, off_t
    roffset, int rma_flags);
int scif_vreadfrom(scif_epd_t epd, void* addr, size_t len, off_t
    offset, int rma_flags);
int scif_vwriteto(scif_epd_t epd, void* addr, size_t len, off_t
    offset, int rma_flags);
```

The `scif_readfrom()` and `scif_writeto()` functions perform DMA or CPU based read and write operations, respectively, between physical memory of the local and remote nodes of the specified endpoint and its peer. The physical memory is that which is represented by specified ranges in the local and remote registered address spaces of a local endpoint and its peer remote endpoint. Specifying these registered address ranges establishes a correspondence between local and remote physical pages for the duration of the RMA operation. The `rma_-flags` parameter controls whether the transfer is DMA or CPU based.

Figure 11 below illustrates such a mapping. The process performing the operation specifies a range, LR, within the registered address of one of its connected endpoints, and a corresponding range, RR, of the same length within the peer endpoint's registered address space. Each specified range must be entirely within a previously registered window or contiguous windows of the corresponding registered address spaces. The solid green lines represent the correspondence between the specified ranges in the local and remote registered address spaces; the dashed green lines represent the projections into their respective physical address spaces. This defines an overall effective correspondence (black lines) between the physical address space of the local node and that of the remote node of the peer registered address space.

Hence, a DMA operation will transfer data between LP and RP (again, LP and RP are typically not contiguous).

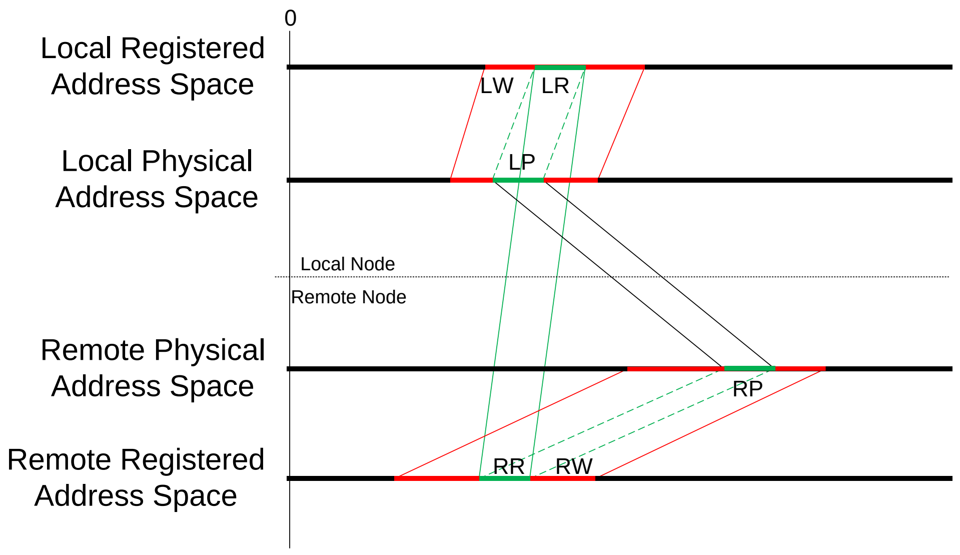
0
Local Registered Address Space
LW
LR
Local Physical Address Space
LP
Local Node
Remote Node
Remote Physical Address Space
RP
Remote Registered Address Space
RR
RW

Figure 11: `scif_readfrom()/scif_writeto()` address space mapping

`scif_vreadfrom`() and `scif_vwriteto`() are variants of `scif_readfrom`() and `scif_writeto`(). Rather than taking a local registered address space range parameter, these functions take a local user address space range, V. Transfers are then between the local physical pages, LP, which back V, and the remote physical pages, RP which are represented by RR. The resulting address space mapping is illustrated in Figure 12.

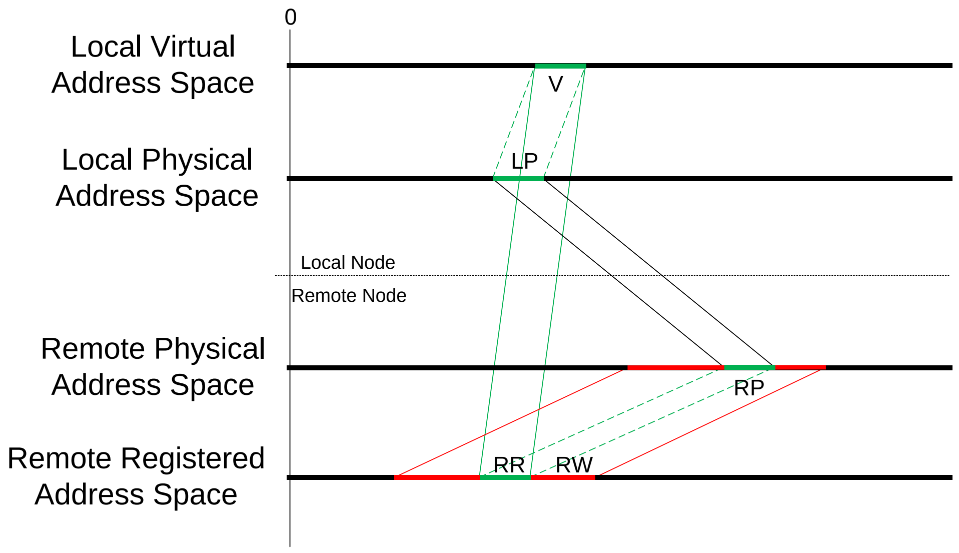
0
Local Virtual Address Space
V
Local Physical Address Space
LP
Local Node
Remote Node
Remote Physical Address Space
RP
Remote Registered Address Space
RR
RW

Figure 12: `scif_vreadfrom()/scif_vwriteto()` address space mapping

If it is known that a buffer will be used multiple times as the source or destination of an RMA, then it is typically beneficial to `scif_register`() the buffer and use `scif_readfrom`() and `scif_writeto`() to perform the transfers. However, if it's known that the buffer will only be used once, or if it is unknown if the buffer will be used multiple times (this might be the case in a library on top of SCIF), then using `scif_vreadfrom`() and `scif_vwriteto`() may provide a performance advantage as compared to registering some window in the local registered address space, performing a single RMA operation to or from that window, and then unregistering the window.

As mentioned above, in some cases it is not known whether a local buffer will be used in subsequent transfers. For this case, the `scif_vreadfrom`() and `scif_vwriteto`() functions have a caching option. When the `rma_flags` parameter includes the SCIF_RMA_USECACHE flag, physical pages that were pinned in order to perform the RMA may remain pinned after the transfer completes. This may reduce overhead if some or all of the same virtual address range is referenced in a subsequent execution of `scif_vreadfrom`() or `scif_vwriteto`() since pinning pages has relatively high overhead. A cached page is evicted from the cache in the event that it no longer backs the user space page that it backed when first cached.

3.7.1 DMA Ordering

The Intel® Xeon Phi™ Coprocessor DMA engine does not maintain write ordering. That is some written data may become visible before written data with a lower address. This might be an issue if the process to which data is being transferred polls the last byte of a buffer for some trigger value as an indication that the transfer has completed.

When the `rma_flags` parameter includes the SCIF_RMA_ORDER flag, the last cacheline or partial cacheline of the transfer is written after the all other data in the transfer is written. There is slight performance penalty for invoking this feature.

Similarly, the order in which any two RMA transfers complete is indeterminate. SCIF synchronization functions, described in the next section, can be used to synchronize to the completion of RMA transfers.

3.8 RMA Synchronization

SCIF supports the ability of a process to synchronize with the completion of RMA operations previously initiated against one of its endpoints, or against a peer of one of its endpoints. The following functions comprise the synchronization group:

```
int scif_fence_mark(scif_epd_t epd, int flags, int* mark);
int scif_fence_wait(scif_epd_t epd, int mark);
int scif_fence_signal(scif_epd_t epd, off_t loff, uint64_t lval, off_t
    roff, uint64_t rval, int flags);
```

There are two synchronization methods available. The first method uses both the `scif_fence_mark`() and `scif_fence_wait`() functions. The `scif_fence_mark`() function marks the set of RMAs previously initiated against a specified endpoint or against its peer, and which have not yet completed. `scif_fence_mark`() returns a handle to the application which the application can later pass to `scif_fence_wait`() in order to await completion of all RMAs in the marked set. If the `flags` parameter has the SCIF_FENCE_RAS_SELF flag, then `scif_fence_mark`() marks RMAs initiated through the local endpoint. If the `flags` parameter has the SCIF_FENCE_RAS_PEER flag, then scif_fence_mark() marks RMAs initiated through the peer endpoint. flags can have only one of these flags values.

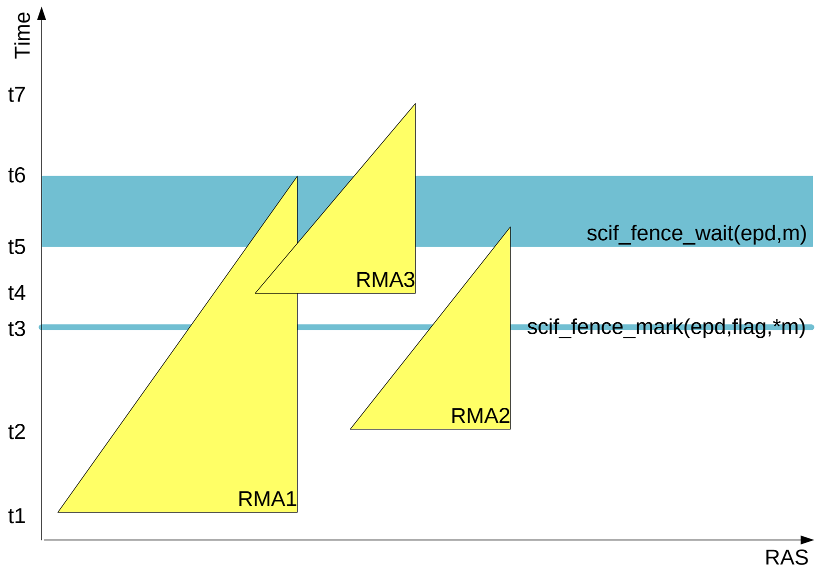

Figure 13: **scif_fence_mark()/scif_fence_wait()**

This is illustrated in Figure 13 (the triangles are meant to indicate RMA progress over time). RMA1 and RMA2 are initiated at times t1 and t2, respectively, against some endpoint descriptor epd. At time t3, scif_fence_mark() is called, marking RMA1 and RMA2 as members of some set, and returning a handle m to that set. At time t4, RMA3 is initiated. The application then calls scif_fence_wait() at time t5 to await the completion of RMAs in the set indicated by handle m. scif_fence_wait() then returns at time t6 when RMA1 completes.

The second synchronization method uses the scif_fence_signal(). This function returns after conceptually marking the set of RMAs previously initiated against a specified endpoint or against its peer endpoint, and which have not yet completed. Like scif_fence_mark(), if the flags parameter has the SCIF_FENCE_RAS_SELF flag, then scif_fence_mark() marks RMAs initiated through the local endpoint. If the flags parameter has the SCIF_FENCE_RAS_PEER flag, then scif_fence_mark() marks RMAs initiated through the peer endpoint. flags can have only one of these flags values.

When all the RMAs in the marked set have completed, an application specified value, lval, is written to a specified offset, loff, in the registered address space of a local endpoint and/or another application specified value, rval, is written to another specified offset, roff, in the registered address space of the peer of the local endpoint, as specified by the SCIF_SIGNAL_LOCAL and SCIF_SIGNAL_REMOTE flag values. Each specified offset must be within a registered window of the corresponding registered address space.

The local process and/or the peer process may poll the virtual address which maps to the specified registered address space offset waiting for the specified value(s) to be written.

scif_fence_signal() is illustrated in Figure 14 in which the same sequence of RMAs is initiated. The application calls scif_fence_signal() at time t3, passing a local offset, loff, and a value, v to be written to `loff`. Then `scif_fence_signal`() returns after marking RMA1 and RMA2, that were previously initiated and have not completed. At time t6, when all RMAs in the marked set have completed, a value v is written to the registered address space at offset `loff`. (For simplicity, we don't try to illustrate writing to values to both the local and remote registered address spaces.)

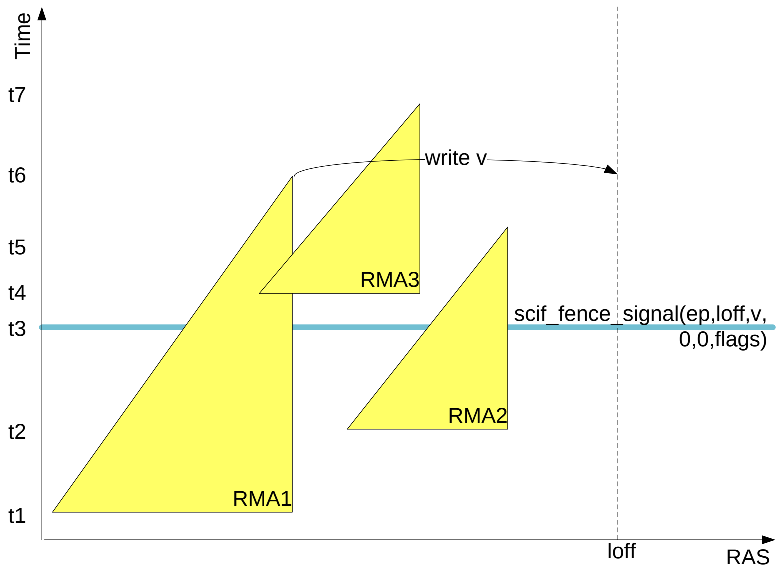

Figure 14: **`scif_fence_signal`()**

Note that marking a set of RMAs does not impose a barrier. That is, an RMA that is submitted after a set of RMAs is marked can begin transferring, and even complete its transfer, before the marked set completes. This is the case for both synchronization methods. For example, in the figure above RMA3 is shown to access some of the same registered address range as RMA1 while RMA1 is in progress. Thus if RMA1 is a transfer to some memory and RMA3 is a transfer out of some of the same memory, RMA3 would likely not transfer out the expected data in this case. It is the application's responsibility to order RMAs as needed by using SCIF synchronization functionality to await the completion of previous RMAs before subsequent RMAs are submitted. In this case, the application should wait until after RMA1 and RMA2 have completed by polling for v before initiating RMA3:

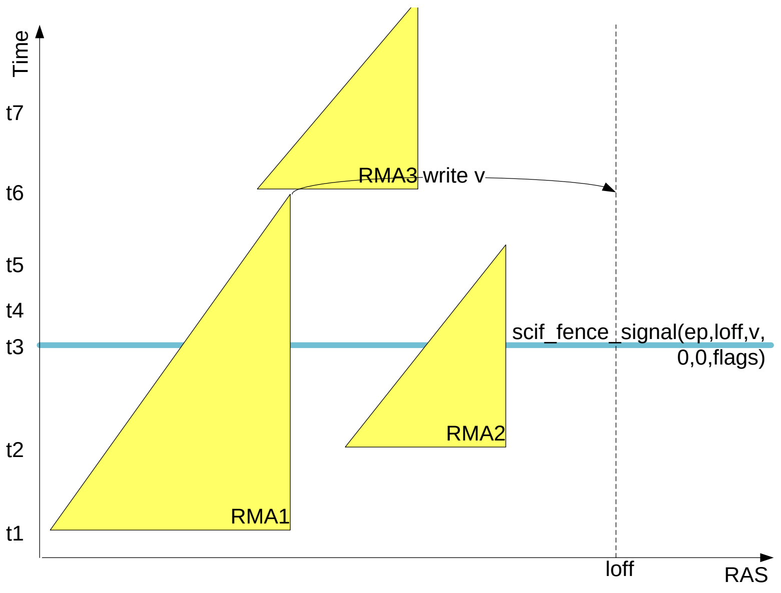

Figure 15: Using scif_fence_signal()

In the case that an application must wait for a DMA transfer to complete before it can do any other work, it can use either of the two fence mechanisms described above. Alternatively, if the `rma_flag`s parameter of any RMA API includes the SCIF_RMA_SYNC flag, then control will not return to the application until the RMA has completed.

3.9 Registered Window Deletion

The `scif_unregister`() function is used to delete one or more registered windows, as specified by a local endpoint and a range within that endpoint's registered address space. The range must completely encompass zero or more windows. Deleting a portion of a window is not supported.

After `scif_unregister`() is called to delete a window, the registered address space range of the window is no longer available for use in calls to `scif_mmap`(), `scif_get_pages`(), `scif_readfrom`(), `scif_writeto`(), `scif_vreadfrom`(), `scif_vwriteto`() and `scif_fence_signal`(). However, the window continues to exist until all references to the window are removed. A window is referenced if there is a mapping to it created by `scif_mmap`(), or if `scif_get_pages`() was called against the window (and the pages have not been returned via `scif_put_pages`()). A window is also referenced while an RMA, in which some range of the window is a source or destination, is in progress. Finally a window is referenced while some offset in that window was specified to `scif_fence_signal`(), and the RMAs marked by that call to `scif_fence_signal`() have not completed. Until the window is deleted, no portion of its registered address space range can be used to create a new window, and all the physical pages represented by that window remain locked.

A physical page can be represented by multiple windows; for example, see cases 1 and 3 in Figure 5 above. Such a page remains locked until all the windows which represent it are deleted.

3.10 Connection Termination

We distinguish between normal connection termination that is triggered by one of the processes at each end of a connection, and abnormal termination triggered when a node becomes "lost".

3.10.1 Normal Connection Termination

A connection is terminated when `scif_close`() is called on one of its endpoints. The following steps describe the process of closing an endpoint, and apply to both the local endpoint and its peer.

- Further operations through the closing endpoint are not allowed, with the exception described below.
- All previously initiated RMAs to or from windows of the endpoint are allowed to complete.
- Blocked calls to `scif_send`() or `scif_recv`() through the closing endpoint are unblocked and return the number of bytes sent or received, or return the ECONNRESET error if no data was sent or received.
- Each window of the closing endpoint is unregistered as described for `scif_unregister`(). In particular, the physical pages represented by each window remain locked until all references to the window are removed. Thus mappings to its windows previously established by `scif_mmap`() remain until removed by `scif_mmap`(), `scif_munmap`(), or standard functions such as `mmap`() and `munmap`(), or until the process holding the mapping is killed. In kernel mode, it is an error to call `scif_close`() on an endpoint for which there are outstanding physical page addresses obtained from `scif_get_pages`().

If an endpoint was closed because its peer was closed, `scif_recv`() can be called on the local endpoint while its receive buffer is non-empty and will return data until the receive queue is empty, at which time it returns the ECONNRESET error. This allows an application to send a message, and then close the local endpoint without waiting somehow for the message to be received by the remote endpoint.

In all other cases, a SCIF function call returns the ECONNRESET error if it references an endpoint that is no longer connected because the peer endpoint was closed.

3.10.2 Abnormal Connection Termination

When a node in the SCIF network is lost and must be reset for some reason, the SCIF driver on each other node will *kill*() any user mode process which has scif_mmap()'d pages from the lost node. This is done to prevent corruption of the memory of the lost node after it is reset.

Access to any remaining endpoint which was connected to an endpoint on the lost node now returns the ECONNRESET error. The application may `scif_close`() such an endpoint as part of cleaning up from the loss of the node.

Each kernel mode module that uses SCIF must register a callback routine with the SCIF driver:

```
void scif_event_register (scif_callback_t handler);
```

that is the routine to be called in the event that a node is added or is lost and must be reset. Upon being called with the SCIF_NODE_REMOVED event, and before returning, the event handler must return, using scif_put_pages(), all structures obtained using `scif_get_pages`() against an endpoint connected to the lost node. It is recommended and expected that the handler will also `scif_close`() all endpoints connected to the lost node.

3.11 Process Termination

When a process is terminated, either normally or abnormally, the following steps are performed:

- All remote mappings previously created by `scif_mmap`() are removed as if `scif_munmap`() were called on the mapping.
- Physical page addresses obtained from `scif_get_pages`() are effectively returned as if `scif_put_pages`() were called.
- Each endpoint owned by the process is closed as if `scif_close`() were called on the endpoint.

3.12 User Mode Utility Functions

Several utility functions are defined in the SCIF user mode API:

```
int scif_get_nodeIDs(uint16_t* nodes, int len, uint16_t* self);
static int scif_get_fd(scif_epd_t epd);
int scif_poll(struct scif_pollepd* epds, unsigned int nepds, long
    timeout);
```

The `scif_get_nodeIDs`() function may be called to obtain the IDs of the nodes currently in the SCIF network. This function also returns the ID of the node on which the calling process is executing.

`scif_get_fd`() returns the file descriptor which backs a specified endpoint descriptor, epd. The file descriptor returned can be used when calling poll() or select(). It should in this way. This function is only available in the Linux* user mode API

`scif_poll`() waits for one of a set of endpoints to become ready to perform an I/O operation; it is syntactically and semantically very similar to poll() . The SCIF functions on which `scif_poll`() waits are `scif_accept`(), `scif_send`(), and `scif_recv`(). Consult the SCIF header file, scif.h, and the SCIF man pages for details on `scif_poll`() usage.

3.13 Kernel Mode Utility Functions

The `scif_get_nodeIDs`() and `scif_poll`() functions are available in kernel mode. In addition, the `scif_pce_dev`() function:

```
int scif_pci_dev(uint16_t node, struct pci_dev** pdev);
```

returns the `pci_dev` structure pointer associated with specified SCIF node. This structure can then be used in standard Linux* kernel functions to refer to an Intel® Xeon Phi™ coprocessor. For example the `pci_dev` structure can be used to obtain system bus addresses from a virtual address or page pointer in calls to Linux* PCIe mapping APIs like `pci_map_single`() or `pci_map_page`().

4 Programming Considerations

4.1 Unaligned DMAs

The Intel® Xeon Phi™ Coprocessor DMA engine supports cacheline aligned transfers. That is, starting and ending addresses of DMA transfers must be a multiple of 64. SCIF RMA APIs (scif_readfrom(), scif_writeto(), scif_vreadfrom(), scif_vwriteto()) may be specified with any alignment: The source and destination may have any alignment, these alignments may differ, and the length of a transfer need not be a multiple of 64.

When a request is made to use DMA for a transfer that is not cacheline aligned, SCIF uses a combination of DMA and programmed I/O to implement the transfer. Such transfers will have lower performance than the cacheline aligned transfers. Therefore, optimal DMA performance will likely be realized if both source and destination base addresses are cacheline aligned. Lower performance will likely be realized if the source and destination base addresses are not cacheline aligned but are separated by some multiple of 64. The lowest level of performance is likely if source and destination base addresses are not separated by a multiple of 64.

A suggested workaround is to pad data allocations to ensure cacheline alignment of data structures that are to be DMA'd.

4.2 Synchronization Overhead

The `scif_fence_mark`() and `scif_fence_wait`() functions should be used somewhat judiciously in order to minimize overhead. For example, an application might call `scif_fence_mark`() after each RMA, and then later chose on which mark(s) to wait. Such a sequence can have a negative impact on BW, particularly where transfers are small.

4.3 Large pages

SCIF registration and DMA performance will be better if the buffers being registered are backed by huge pages. SCIF registration is improved because the driver requires fewer data structures to accurately store meta-data about huge pages which are contiguous in physical memory as compared to storing the meta data for every 4K page. SCIF DMA performance is improved since the software overhead for programming DMA descriptors is reduced. SCIF detects and optimizes for huge pages transparently. The user does not need to specify if a virtual address region is backed by huge pages or not. Maximum performance benefits will be seen if both source and destination buffers are backed by huge pages.