



SGI® Altix® UV GRU Development Kit
Programmer's Guide

007-5668-003

COPYRIGHT

© 2010, 2011, SGI. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of SGI.

LIMITED RIGHTS LEGEND

The software described in this document is "commercial computer software" provided with restricted rights (except as to included open/free source) as specified in the FAR 52.227-19 and/or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS

Altix, NUMAlink, SGI, the SGI logo, SHMEM, Silicon Graphics, and Supportfolio are trademarks or registered trademarks of Silicon Graphics International Corp. or its subsidiaries in the United States and other countries.

Intel and Xeon are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in several countries.

New Feature in This Manual

This rewrite of the *SGI Altix UV GRU Development Kit Programmer's Guide* supports the SGI Performance Suite 1.3 release.

Major Documentation Changes

Added additional information about the global reference unit (GRU) in "Introduction" on page 1.

Updated "Advantages Provided by Directly Programming the GRU" on page 2.

Record of Revision

Version	Description
001	June 2010 Original Printing.
002	July 2010 Updated to support the SGI ProPack 7 Service Pack 1 release.
003	October 2010 Updated to support the SGI Performance Suite 1.0 release.
004	November 2011 Updated to support the SGI Performance Suite 1.3 release.

Contents

About This Manual	ix
Obtaining Publications	ix
Related Publications	ix
Helpful Online Resources	x
Conventions	x
Reader Comments	xi
1. Altix UV GRU Direct Access API	1
Introduction	1
Advantages Provided by Directly Programming the GRU	2
Accessing the Altix UV GRU Direct Access API	4
SGI High Level APIs Supporting GRU Access	4
Overview of API for Direct GRU Access	4
GRU Resource Allocators	5
GRU Access Functions	6
GRU Memory Access Functions	9
XPMEM Library Functions	11
MPT Address Mapping Functions	12
MPI_SGI_gam_ptr Function	13
MPI_SGI_symmetric_addr Function	14
shmem_ptr Function	14
GRU Library Program Example	14
Simple GRU Program	17
2. GRU Driver and GRU Libraries Environment Variables	19
007-5668-003	vii

Contents

GRU_TLBMISS_MODE	19
GRU_CCH_REQUEST_SLICE	19
GRU_TLB_PRELOAD	20
GRU_STATISTICS_FILE	20
GRU_TRACE_FILE	20
GRU_TRACE_INSTRUCTIONS	21
GRU_TRACE_EXCEPTIONS	21
GRU_TRACE_INSTRUCTION_RETRY	21
GRU Files in /proc	22
grustats Command	24
3. GRU Software Functions	27
Checking the Status of GRU Operations	27
Displaying GRU Error Information	27
GRU Data Transfer Functions	27
xtype	28
exopc	28
Functions for GRU Instructions	29
Index	39

About This Manual

This publication documents the SGI Altix UV global reference unit (GRU) development kit. It describes the application programming interface (API) that allows an application direct access to GRU functionality.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: <http://docs.sgi.com>. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- You can also view man pages by typing `man title` on a command line.

Related Publications

This section describes documentation you may find useful, as follows:

- *Message Passing Toolkit (MPT) User's Guide*

Describes industry-standard message passing protocol optimized for SGI computers.

- *MPInside Reference Guide*

Documents the SGI MPInside MPI profiling tool.

- *Unified Parallel C (UPC) User's Guide*

Documents the SGI implementation of the Unified Parallel C (UPC) parallel extension to the C programming language standard.

- *SGI Altix UV 1000 System User's Guide*

This guide provides an overview of the architecture and descriptions of the major components that compose the SGI Altix UV 1000 system. It also provides the standard procedures for powering on and powering off the system, basic troubleshooting information, and important safety and regulatory specifications.

- *SGI Altix UV 100 System User's Guide*

This guide provides an overview of the architecture and descriptions of the major components that compose the SGI Altix UV 100 system. It also provides the standard procedures for powering on and powering off the system, basic troubleshooting information, and important safety and regulatory specifications.

Helpful Online Resources

Supportfolio is the SGI support web site, including the SGI Knowledgebase, has links for software supports and updates at: <https://support.sgi.com/login>.

For a complete list of SGI online resources, see the *SGI Performance Suite 1.x Start Here*.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
<code>command</code>	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
<code>manpage(x)</code>	Man page section identifiers appear in parentheses after man page names.
<i>variable</i>	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[]	Brackets enclose optional portions of a command or directive line.

...

Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:
techpubs@sgi.com
- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:
SGI
Technical Publications
46600 Landing Parkway
Fremont, CA 94538

SGI values your comments and will respond to them promptly.

Altix UV GRU Direct Access API

Note: This manual only applies to SGI® Altix UV 100 and SGI Altix UV 1000 series systems.

This chapter provides an overview of the SGI Altix UV global reference unit (GRU) development kit. It describes the application programming interface (API) that allows an application direct access to GRU functionality.

Introduction

The GRU is part of the SGI Altix UV Hub application-specific integrated circuit (ASIC). The UV Hub is the heart of the SGI Altix UV 1000 or Altix UV 100 system compute blade. It connects to two Intel® Xeon® 7500 series processor sockets through the Intel QuickPath Interconnect (QPI) ports and to the high speed SGI NUMALink® interconnect fabric through one of four NUMALink 5 ports. The Intel processor sockets can have two four-core, six-core, or eight-core processors with on-chip secondary caches.

This UV Hub acts as a crossbar between the processors, local SDRAM memory, and the network interface. The Hub ASIC enables any processor in the single-system image (SSI) to access the memory of all processors in the SSI.

The GRU is a coprocessor that is effective at assisting the transfer of data between compute nodes at a higher speed than socket-level instructions. In addition, the GRU has support for the following:

- Efficient atomic memory operations (AMOs)
- Internode messages sent to message queues located on remote nodes
- On partitioned systems, the GRU is able to access memory located on remote single system images (SSIs).

The GRU features are available to both the kernel and to user applications. Message Passing Toolkit (MPT), the SGI MPI library, is a primary user of the GRU. In addition, user applications can directly reference the GRU using header files and libraries that are provided by SGI.

The system architecture for the Altix UV 1000 and Altix UV 100 system is a fifth-generation NUMAflex distributed, shared memory (DSM) architecture known as NUMALink 5. In the NUMALink 5 architecture, all processors and memory can be tied together into a single logical system.

For more information on the SGI Altix UV hub, Altix UV compute blades, QPI, and NUMALink 5, see the *SGI Altix UV 1000 System User's Guide* or the *SGI Altix UV 100 System User's Guide*, respectively. This chapter covers the following topics:

- "Advantages Provided by Directly Programming the GRU" on page 2
- "Accessing the Altix UV GRU Direct Access API" on page 4
- "SGI High Level APIs Supporting GRU Access" on page 4
- "Overview of API for Direct GRU Access " on page 4
- "GRU Resource Allocators" on page 5
- "GRU Access Functions" on page 6
- "GRU Memory Access Functions" on page 9
- "XPMEM Library Functions" on page 11
- "MPT Address Mapping Functions" on page 12
- "GRU Library Program Example" on page 14

Advantages Provided by Directly Programming the GRU

The low level Altix UV GRU API provides direct access to the full set of GRU instructions. Most of these instructions are not available through the use of the MPT, SHMEM, or UPC APIs. The full benefit of the GRU is in the ability to have the GRU asynchronously executing instructions in the background while the user application performs other work.

The GRU instruction set and hardware architecture provides the following capabilities:

- Take advantage of the available bandwidth and NUMALink message efficiency with vector-like instruction
- Take advantage of specific hardware mechanisms aimed at reducing network traffic

- Expand the reach of the limited processor cores outstanding references by bringing latency tolerant remote references in the local hub resources, increasing sustained bandwidth in all but the smallest systems.
- Improve the apparent processor bus efficiency by compacting strided or random references into cache lines.
- Provide efficient synchronization and communication hardware assisted primitives aimed at improving latency for common synchronization and messaging operations (including MPI applications) by reducing the number of network traversals between GRU users and target references in system memory.

Figure 1-1 on page 3 shows the major capabilities of the GRU.

Global Reference Unit Overview

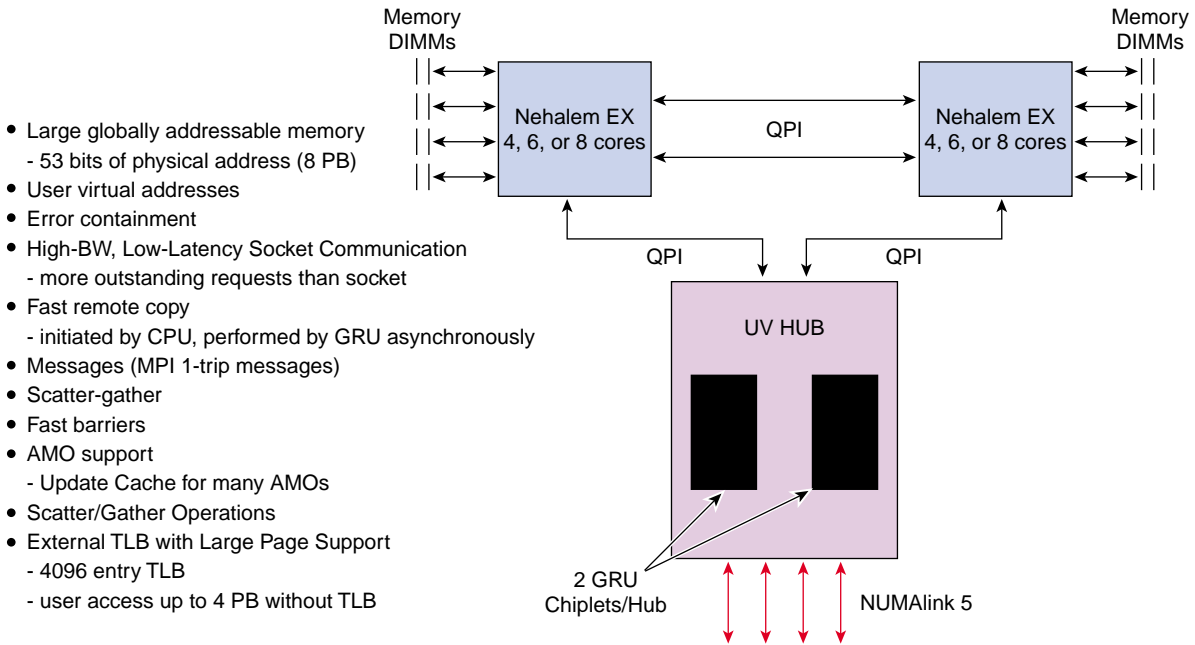


Figure 1-1 Global Reference Unit (GRU) Capabilities

Accessing the Altix UV GRU Direct Access API

In order to access and use the Altix UV GRU direct access API, you need to install the following RPMs on your SGI Altix UV system:

- `xpmem-devel`
- `gru-devel`
- `gru_alloc-devel`
- `libgru-devel`

Note: These RPMs are **not** installed by default.

SGI High Level APIs Supporting GRU Access

Message Passing Interface (MPI), SHMEM, and Unified Parallel C (UPC) high level APIs and programming models that are implemented and supported by SGI that support access to GRU functionality. For more information, see `mpi(1)`, `shmem(3)`, or `sgiupc(1)` man pages and the *Message Passing Toolkit (MPT) User's Guide* and *Unified Parallel C (UPC) User's Guide*.

Overview of API for Direct GRU Access

The Direct GRU Access API has four components, as follows:

- GRU resource allocators

The GRU resource allocator functions provide management of the GRU resources to allow independent software components in the same program access the GRU without oversubscribing the GRU resources.

- GRU memory access functions

The GRU memory access functions perform GRU operations that include memory read, memory write, memory-to-memory copies, and atomic memory operations and so on.

- XPMEM address mapping functions

The XPMEM address mapping functions set up mappings to target memory throughout the system into local GRU-mapped virtual addresses.

- MPT address mapping functions

The MPT address mapping functions are a layer on top of XPMEM, and expose mapped memory regions already set up for MPI and SHMEM to the user application.

GRU Resource Allocators

The UV global reference unit (GRU) has control block (CB) and data segment (DSEG) resources associated with it. User applications need to allocate CB resources and usually DSEG resources for use in GRU memory access functions.

There are two categories of GRU resources used by any thread: temporarily and permanently allocated. A program starts running with all the available GRU resources being in the temporary pool until some resources are allocated permanently via the `gru_pallocate()` function.

The preferred way to get access to all the GRU temporary CBs and DSEG is through the use of the lightweight `gru_temp_reserve()` and `gru_temp_release()` functions. These functions should wrap any use of the GRU memory access functions, with an exception to be described later.

```
#include <gru_alloc.h>

void gru_temp_reserve(gru_alloc_thdata_t *gat);

typedef struct {
    gru_segment_t      *gruseg;
    gru_control_block_t *cbp;
    void               *dsegp;
    int                 cb_cnt;
    int                 dseg_size;
} gru_alloc_thdata_t;
```

The `gru_alloc_thdata_t` structure returned from this function will describe the GRU resources available for use until the next call to `gru_temp_release()`.

The following code example shows a GRU memory access function `gru_gamirr()` being called after which the `gru_temp_reserve()` function reserves the GRU resources, and before the `gru_wait_abort()` function waits for completion of the operation. Then, followed by a call to `gru_temp_release()` to release the temporary GRU resources.

Example 1-1 GRU Memory Access Function (`gru_gamirr()`)

```
gru_alloc_thdata_t gat;
gru_temp_reserve(&gat);
gru_gamirr( gat.cbp, EOP_IRR_DECZ, address, XTYPE_DW, IMA_CB_DELAY);
gru_wait_abort(gat.cbp);
gru_temp_release();
```

The effect of the `gru_temp_reserve()` and `gru_temp_release()` functions is thread-private, so related POSIX threads or OpenMP threads could be executing the above sequence, concurrently.

An alternative allocation scheme is permanent allocation. The `gru_pallocate()` function returns CB and DSEG resources that can be used at any time thereafter. This can simplify the allocation strategy but it has the disadvantage of reducing the number of GRU resources that can be used by other software. An example would be a call to `gru_bcopy()` which allows you to pass a DSEG work buffer of any size. The achieved bandwidth for `gru_bcopy()` is higher with larger DSEG work buffers.

See the `gru(7)` man page for a complete list of GRU man pages. You can also use the `man gru` command to view these pages.

GRU Access Functions

The following functions are used to create and manage user access to the GRU. Each function has an associated *man(3)* page. For a list of GRU man pages, refer to the *SEE ALSO* section at the bottom of the `gru(7)` man page.

- `gru_create_context()`
Creates a GRU context to allow a user access to the GRU
- `gru_get_data_pointer()`
Gets a pointer to a GRU control block

- `gru_wait_abort()`
Waits for an active GRU instruction to complete. Aborts on error
- `gru_set_context_blade_chiplet()`
Selects GRU blade and chiplet for context
- `gru_unload_context()`
Unloads a GRU context
- `gru_check_status()`
Checks the status of a GRU instruction to complete
- `gru_wait()`
Waits for an active GRU instruction to complete
- `gru_get_cb_exception_detail_str()`
Gets string describing a GRU instruction exception
- `gru_print_cb_detail()`
Prints detailed error information for GRU instruction failure
- `gru_flush_tlb()`
Flushes a virtual address range from the GRU
- `gru_create_message_queue()`
Creates a GRU message queue
- `gru_abort()`
Causes abnormal process termination due to GRU instruction error
- `gru_destroy_message_queue()`
Frees resource allocated to a GRU message queue
- `gru_get_cb_substatus()`
Gets the GRU instruction sub-status
- `gru_get_amo_value()`

- Gets the AMO value from a GRU control block
- `gru_get_cb_status()`
- Gets the GRU instruction status
- `gru_free_message()`
- Frees a message from a GRU message queue
- `gru_get_amo_value_head()`
- Gets the head value for a GRU message queue AMO
- `gru_get_amo_value_limit()`
- Gets the limit value for a GRU message queue AMO
- `gru_get_next_message()`
- Gets the next message from a GRU message queue
- `gru_send_message()`
- Sends a message to a GRU message queue
- `gru_start_message()`
- Sends an asynchronous message to a GRU message queue
- `gru_wait_message()`
- Waits for asynchronous message sent to a GRU message queue
- `gru_mesq_head()`
- Returns a GRU message queue header value
- `gru_destroy_context()`
- Destroys a GRU context and free the GRU resources
- `gru_get_thread_gru_segment()`
- Gets GRU context identifier to use to access a GRU context
- `gru_get_cb_pointer()`
- Gets a pointer to a GRU control block

- `gru_get_tri()`
Gets a `tri0/tri1` index to a GRU data segment element

For detailed information, see the `gru(7)` man page.

GRU Memory Access Functions

The GRU memory access functions perform GRU operations that include memory read, memory write, memory-to-memory copies, and atomic memory operations. These functions use an ordinary virtual address or a GRU-mapped virtual address to reference the remote memory. Each function has an associated *man(3)* page.

The following in-line functions are provided by `gru_instructions.h` and are used to initiate GRU instructions:

- `gru_bcopy()`
Memory to memory copy using the GRU
- `gru_bstore()`
Stores data from the GRU into system memory
- `gru_gamer()`
GRU unregistered atomic memory operation with explicit data
- `gru_gamerr()`
GRU registered atomic memory operation with explicit data
- `gru_ivload()`
Indirectly load data into the GRU from system memory
- `gru_ivset()`
Indirectly store a defined data value into system memory using the GRU
- `gru_ivstore()`
Indirectly store data from the GRU into system memory
- `gru_nop()`

No operation, cancel active GRU instruction

- `gru_vflush()`

Flush cache lines from processor caches using the GRU

- `gru_vload()`

Load data into the GRU from system memory

- `gru_vstore()`

Store data from the GRU into system memory

- `gru_gamxr()`

GRU unregistered atomic memory operation with extended data

- `gru_mesq()`

Atomically send a message to a message queue using the GRU

- `gru_vset()`

Store a defined data value into system memory using the GRU

For detailed information, see the `gru(7)` man page.

The interfaces to these functions are viewable in the `uv/gru/gru_instructions.h` header file installed by the `gru-devel` RPM.

The following code example of a GRU memory access function illustrates the basic call structure.

Example 1-2 GRU Memory Access Function Basic Call Structure

```
static inline
void gru_vload(gru_control_block_t *cb, void *mem_addr,
              unsigned int tri0, unsigned char xtype, unsigned long nelem,
              unsigned long stride, unsigned long hints);
```

Arguments are:

`cb` - pointer to CB
`mem_addr` - address of targeted memory
`tri0` - index to DSEG buffer. Compute it using `gru_get_tri()`.
`xtype` - log2 of data type byte size (`XTYPE_B ...`)

```
nelem - number of elements to transfer
stride - memory stride, scaled in elements
hints - IMA_CB_DELAY is commonly used
```

All memory access operations are asynchronous. The wait functions, such as, `gru_wait_abort()`, specify the CB handle and are used to wait to completion.

XPMEM Library Functions

The XPMEM interface can map a virtual address range in one process into the GRU-mapped virtual address in another process. The XPMEM interface was designed to meet the needs of MPI and SHMEM implementations and provide ways to map any data region. As a GRU API user, you need to find a way to map the needed memory regions into the processes or threads involved. The Linux operating system offers many options for doing this, as follows:

- `mmap`
- System V shared memory
- memory sharing among `pthread`s
- memory sharing among OpenMP threads

These methods are the likely first choice for most potential GRU users.

The `sn/xpmem.h` header file installed by the `xpmem-devel` RPM has interface definitions for all the XPMEM functions.

The following example shows the main XPMEM functions:

Example 1-3 Main XPMEM Functions

```
extern __s64 xpmem_make_2(void *, size_t, int, void *);
extern int xpmem_remove_2(__s64);
extern __s64 xpmem_get_2(__s64, int, int, void *);
extern int xpmem_release_2(__s64);
extern void *xpmem_attach_2(__s64, off_t, size_t, void *);
extern void *xpmem_attach_high_2(__s64, off_t, size_t, void *);
extern int xpmem_detach_2(void *, size_t size);
extern void *xpmem_reserve_high_2(size_t, size_t);
extern int xpmem_unreserve_high_2(void *, size_t);
```

For more information on using XPMEM, see *SGI Altix UV Systems Configuration and Operations Guide*.

MPT Address Mapping Functions

The MPT `libmpi` library uses XPMEM to cross-map virtual memory between all the processes in an MPI job. Several functions are available to lookup mapped virtual addresses that are pre-attached in the virtual address space of a process by MPI. The addresses returned by the lookups may be passed to the GRU library functions.

Not all GRU API users can require their code to execute in an MPI job, but if you do, you may find the MPT address mapping functions are a convenient way to reference remote data arrays and objects.

The MPT address mapping functions are shown below. They reference ordinary virtual addresses or addresses of symmetric data objects. Symmetric data is static data or array-defined in the `intro_shmem(3)` man page.

The following example shows an `MPI_SGI_gam_type`:

Example 1-4 `MPI_SGI_gam_type`

```
#include <mpi_ext.h>

int
MPI_SGI_gam_type(int rank, MPI_Comm comm)
```

Return value is the XPMEM accessibility of the specified rank.

<code>MPI_GAM_NONE</code>	- not referenceable by load/store or GRU
<code>MPI_GAM_CPU_NONCOH</code>	- Altix 3700 noncoherent
<code>MPI_GAM_CPU</code>	- if referenceable by load/store only
<code>MPI_GAM_GRU</code>	- if referenceable by GRU only
<code>MPI_GAM_CPU_PREF</code>	- if referenceable by either load/store or GRU, preferred by load/store
<code>MPI_GAM_GRU_PREF</code>	- if referenceable by either load/store or GRU, preferred by GRU

The MPT address mapping functions are influenced by the `MPI_GSM_NEIGHBORHOOD` environment variable. This variable may be used to specify the "neighborhood size" for shared memory accesses. Contiguous groups of ranks within a host can be considered to be in the same neighborhood. The `MPI_GSM_NEIGHBORHOOD` variable specifies the size of these neighborhoods, as follows:

- MPI processes within a neighborhood will return `gam_type MPI_GAM_CPU_PREF`.
- MPI processes outside a neighborhood with a host will return `gam_type MPI_GAM_GRU_PREF`.
- MPI processes from a different host within a Altix UV system will return `gam_type MPI_GAM_GRU`.

When `MPI_GSM_NEIGHBORHOOD` is not set, the neighborhood size defaults to all ranks in the current host.

MPI_SGI_gam_ptr Function

The `MPI_SGI_gam_ptr` function is, as follows:

```
#include <mpi_ext.h>

void * MPI_SGI_gam_ptr(void *rem_addr, size_t len, int remote_rank,
    MPI_Comm comm, int acc_mode);
```

Given a virtual address in a specified MPI process rank, returns a general virtual address that may be used to directly reference the memory.

This function is for general users.

<code>acc_mode</code>	Chooses CPU or GRU addressable
<code>MPI_GAM_CPU</code>	Requests CPU address that can be referenced
<code>MPI_GAM_GRU</code>	Requests GRU address that can be referenced

This function prints an error message when error conditions occur and then aborts.

MPI_SGI_symmetric_addr Function

The `MPI_SGI_symmetric_addr` function is, as follows:

```
void *MPI_SGI_symmetric_addr(void *local_addr, size_t len,  
                             int remote_rank, MPI_Comm comm)
```

For symmetric objects, returns the virtual address (VA) of the corresponding object in a specified MPI process.

shmem_ptr Function

The `shmem_ptr` function is, as follows:

```
#include <mpp/shmem.h>  
  
void *shmem_ptr(void *target, int pe);
```

Returns a processor-referencable address that can be used to reference symmetric data object target on a specified MPI process. See `shmem_ptr(3)` for more details.

GRU Library Program Example

A GRU library programming example follows:

```
/* This SHMEM program uses the GRU API gru_bcopy function to read the bbb
 * variable on PE N+1. This accomplishes a global circular shift into aaa.
 */
#include <mpi_ext.h>
#include <mpi.h>
#include <mpp/shmem.h>
#include <uv/gru/gru_alloc.h>
#include <uv/gru/gru_instructions.h>
int aaa, bbb; /* static data is remotely accessible */
int main ()
{
    int *gptr;
    gru_alloc_thdata_t thd;
    int tri;
    start_pes (0);
    bbb = _my_pe ();
    shmem_barrier_all ();
    gru_temp_reserve (&thd); /* reserve temp GRU resources */
    gptr = MPI_SGI_gam_ptr (
        &bbb, /* address of source */
        1, /* number of elements */
        (_my_pe () + 1) % _num_pes (), /* PE owner of data */
        MPI_COMM_WORLD, /* SHMEM uses MPI_COMM_WORLD */
        MPI_GAM_GRU); /* get GRU-accessible address */
    tri = gru_get_tri (thd.dsegp); /* get offset to DSR buffer */
    gru_bcopy (
        thd.cbp, /* CB 0 will be used */
        gptr, /* GRU pointer for source of copy */
        &aaa, /* GRU pointer for destination of copy */
        tri, /* offset to DSR buffer */
        XTYPE_W, /* data type is 4 byte word */
        1, /* number of elements to copy */
        2, /* number of cache lines of DSR buffer */
        0); /* hints */
    gru_wait_abort (thd.cbp); /* wait for completion of gru_bcopy() */
    gru_temp_release (); /* release GRU resources */
    shmem_barrier_all ();
    printf ("pe %d aaa=%d bbb=%d\n", _my_pe(), aaa, bbb);
    return 0;
}
```

The GRU library programming example, shown in "GRU Library Program Example" on page 14, may be compiled and run on four processes, as follows:

```
% module load mpt
% cc prog.c -lmpi -lsma -lgru_alloc
% mpirun -np 4 ./a.out
```

Simple GRU Program

This section provides a very simple example of a GRU program, as follows:

```
/*
 * Very simple example of a program that:
 * - creates a GRU context
 * - uses VSTORE to store data to memory
 * - uses VLOAD to load data
 * - validates data.
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "uv/gru/gru.h"
#include "uv/gru/gru_instructions.h"

#define perrorx(s)      do {printf("%s: %s", s, strerror(errno)); exit(EXIT_FAILURE);} while (0)
#define Dprintf(s...)  do {if (verbose) printf(s);} while (0)

#define MAGIC 0xdeadbeef12345678UL

static int cbrs = 1;
static int dsrbytes = 64;

static unsigned long data;

int main(int argc, char **argv)
{
    gru_cookie_t cookie;
    gru_segment_t *gseg;
    gru_control_block_t *cb;
    unsigned long *dsr;

    /*
     * Create GRU context for accessing the GRU
     */
    if (gru_create_context(&cookie, NULL, cbrs, dsrbytes, 1, 0) < 0)
        perrorx("cant open gregs");
    if ((gseg = gru_get_thread_gru_segment(cookie, 0)) == NULL)
        perrorx("cant open gregs");
```

```
/*
 * Get pointers to CBR & DSR space
 */
cb = gru_get_cb_pointer(gseg, 0);
dsr = gru_get_data_pointer(gseg, 0);

/*
 * Initialize DSR0. Value equal MAGIC
 */
dsr[0] = MAGIC;

/*
 * Execute the VSTORE and wait for completion
 */
gru_vstore(cb, &data, 0, XTYPE_DW, 1, 1, 0);
gru_wait_abort(cb);

/*
 * Execute the VLOAD and wait for completion
 */
gru_vload(cb, &data, 64, XTYPE_DW, 1, 1, 0);
gru_wait_abort(cb);

/*
 * Validate data
 */
if (dsr[0] != dsr[8] || dsr[0] != MAGIC)
    printf("miscompare: expected 0x%lx, found 0x%lx\n", dsr[0], dsr[8]);

if (gru_destroy_context(cookie))
    perrorx("error closing gru segment");
}
```

GRU Driver and GRU Libraries Environment Variables

This chapter describes environment variables that can be used to specify options to the global reference unit (GRU) driver and GRU libraries. For a description of the GRU, see Chapter 1, "Altix UV GRU Direct Access API" on page 1.

GRU_TLBMISS_MODE

If an instruction references a virtual address that is not in the GRU translation lookaside buffer (TLB), a TLB miss occurs. TLB misses can be handled in several ways:

- `user_polling`

TLB dropins are done as a side effect of users calling `gru_wait` or `gru_check_status` on the coherence buffer request (CBR).

- `interrupt`

The GRU sends an interrupt to the CPU. The TLB dropin is done in the GRU interrupt handler.

- The default mode is "interrupt" although you can override this default using an option on the `gru_create_context()` request. The environment variable can be used to override both, as follows:

```
setenv GRU_TLBMISS_MODE [interrupt|user_polling]
```

GRU_CCH_REQUEST_SLICE

The GRU execution unit timeslices across all active instructions. By default, the GRU issues four NUMALink get/put messages for an active instruction, then switches the next active instruction. You can override the default, as follows:

```
setenv GRU_CCH_REQUEST_SLICE [0|1|2|3]
```

- 0 - issue 4 requests
- 1 - issue 8 requests
- 2 - issue 16 requests
- 3 - not sliced. All requests are issued

GRU_TLB_PRELOAD

The GRU driver can be configured to do anticipatory TLB dropins for GRU BCOPY instructions that take a TLB miss. When a TLB miss occurs, **and** the instruction is a BCOPY, the GRU driver will dropin multiple TLB entries. To configure the GRU driver to do anticipatory TLB dropins for GRU, perform the following:

```
setenv GRU_EXCEPTION_RETRY <num>
<num> number of consecutive retries before returning an error
```

GRU_STATISTICS_FILE

You can collect statistics of a task's usage of GRU contexts by using this option to specify a statistics file, as follows:

```
setenv GRU_STATISTICS_FILE <filename>
```

Whenever a task exits or a GRU context is destroyed, statistics are written to this file. A sample file is, as follows:

```
Pid: 23020                               Mon Oct 19 20:46:56 2009
Command: ./sgup2
CBRs: 4
DSRs: 24576 bytes
Gseg vaddr: 0x7fe3a1e80000
  46740 instructions
    23 instruction_wait
    0 exceptions
  9903 FMM tlb dropin
    1 UPM tlb dropin
  1040 context stolen
```

GRU_TRACE_FILE

You can collect detailed trace of GRU instructions. Use this option to specify the name of the file for the trace information. There are levels of tracing, as follows:

- All GRU instructions

- GRU instructions that return error EXCEPTIONS to users
- GRU instructions that fail and are automatically retried

To collect detailed trace of GRU instructions, perform the following:

```
setenv GRU_TRACE_FILE <filename>
```

GRU_TRACE_INSTRUCTIONS

Setting this option enables tracing of **every** GRU instruction, as follows:

```
setenv GRU_TRACE_INSTRUCTIONS
```

GRU_TRACE_EXCEPTIONS

This option enables tracing of GRU instruction that cause exceptions. Note that some exceptions for GRU MESQ instructions are automatically handled by the GRU `mesq` library routines. These exceptions are not traced if `<val>` is equal to 1 (or not specified). If you want to see these exceptions (`mesq_full`, `amo_nacked`, and so on), set `<val>` to 2.

```
setenv GRU_EXCEPTION_RETRY <num>
<num> number of consecutive retries before returning an error
```

GRU_TRACE_INSTRUCTION_RETRY

This option enables tracing of GRU instructions that fail due to transient errors. The GRU library routine normally retry the instruction and the failure is hidden from the user. If you want to see these failure that are retried successfully, enable this option, as follows:

```
setenv GRU_TRACE_INSTRUCTION_RETRY
```

An example output file is, as follows:

```
Pid: 25276 - gru_wait
    opc: NOP, xtype: BYTE, ima: ImmResp
    istatus: IDLE
```

2: GRU Driver and GRU Libraries Environment Variables

```
Pid: 25276 - gru_wait
    opc: VLOAD, xtype: DWORD, ima: DelResp, baddr0: 0x604450, tri0: 0x0, nelem: 0x1, stride: 0x1
    istatus: IDLE
Pid: 25276 - gru_wait
    opc: VSTORE, xtype: DWORD, ima: DelResp, baddr0: 0x604450, tri0: 0x0, nelem: 0x1, stride: 0x1
    istatus: IDLE
Pid: 25276 - gru_wait
    opc: IVLOAD, xtype: DWORD, ima: DelResp, baddr0: 0x0, tri0: 0x0, tril: 0x40, nelem: 0x1
    istatus: IDLE
Pid: 25276 - gru_wait
    opc: IVSTORE, xtype: DWORD, ima: DelResp, baddr0: 0x0, tri0: 0x0, tril: 0x40, nelem: 0x1
    istatus: IDLE
Pid: 25276 - gru_wait
    opc: VSET, xtype: DWORD, ima: DelResp, baddr0: 0x604450, value: 0x483966aa127ded1d, nelem: 0x1, stride: 0x1
    istatus: IDLE
Pid: 25284, Tid: 25289 - gru_wait
    opc: MESQ, xtype: CACHELINE, ima: DelResp, baddr0: 0x606000, tri0: 0x0, nelem: 0x1
    istatus: EXCEPTION, isubstatus: QLIMIT, avalue: 0f0000000f
        execstatus: EXCEPTION
        state: 0x1, exceptdet0: 0x606000, exceptdet1: 0x8
Pid: 25284, Tid: 25288 - gru_wait
    opc: MESQ, xtype: CACHELINE, ima: DelResp, baddr0: 0x606000, tri0: 0x0, nelem: 0x1
    istatus: EXCEPTION, isubstatus: AMO_NACKED, avalue: 00
        execstatus: EXCEPTION
        state: 0x1, exceptdet0: 0x606000, exceptdet1: 0x8
```

GRU Files in /proc

The `/proc/sgi_uv/gru` directory contains several files that have information about GRU state, as follows:

- `gru_options`
Bit-field that can be used to enable or disable options
- `cch_status`
List of tasks using GRU contexts
- `gru_status`
List of available GRU resources

- `statistics`
Detailed GRU driver statistics (if enabled)

- `mcs_status`
Timing information for kernel GRU commands

Some examples of the files in `/proc/sgi_uv/gru` are, as follows:

Example 2-1 `gru_status` - Available Resources

The file shows the free resources available in each GRU chiplet, as follows:

```
% cat gru_status
# gid nid   ctx   cbr   dsr       ctx   cbr   dsr
#           busy  busy  busy     free  free  free
    0    0     8    36 32768     8    92    0
    1    0     1     4  4096     15   124 28672
    2    1     7    56 28672     9    72  4096
    3    1     7    28 28672     9   100  4096
```

Example 2-2 `gru_options` - Enable or Disable Driver Features

Various GRU options (mostly debugging) can be enabled or disabled by writing values to `/proc/sgi_uv/gru/gru_options` file. Use `cat` command, to view the file to see the current settings or to see a description of the various options.

```
% cat debug_options
# bitmask: 1=trace, 2=statistics, 0x10=No_4k_dsr_AU_war
# bitmask: 0x20=no_iabort_war, 0x40=no_chiplet_affinity
# bitmask: 0x80=no_tlb_war, 0x100=no_mesq_war

0x0001 - enable statistics (they are not free)
0x0002 - enable VERY verbose driver trace information to /var/log/messages
```

Example 2-3 `statistics` - Very Detailed Driver Statistics

You can collect detailed driver statistics, as follows:

```
% echo 2 > /proc/sgi_uv/gru/gru_options
```

This enabled, detailed statistic collection occurs in numerous places in the driver. There is system usage overhead associated with this collection, especially on large systems.

```
% cat /proc/sgi_uv/gru/statistics
45806 vdata_alloc
45771 vdata_free
195712 gts_alloc
195668 gts_free
34351 gms_alloc
34333 gms_free
149398 gts_double_allocate
... (lots more)
```

grustats Command

You can use the `grustats` command, to view GRU statistics. You will see output similar to the following:

```
uv15-sys  TOTAL GRU STATISTICS SINCE COMMAND START
0 vdata_alloc          0 copy_gpa
0 vdata_open           0 read_gpa
0 vdata_free           0 mesq_receive
0 gts_alloc            0 mesq_receive_none
0 gts_free             0 mesq_send
0 gms_alloc            0 mesq_send_failed
0 gms_free             0 mesq_noop
0 gts_double_allocate  0 mesq_send_unexpected_error
0 assign_context       0 mesq_send_lb_overflow
0 assign_context_failed 0 mesq_send_qlimit_reached
0 free_context         0 mesq_send_amo_nacked
0 load_user_context    0 mesq_send_put_nacked
0 load_kcontext        0 mesq_qf_locked
0 load_kcontext_assign 0 mesq_qf_noop_not_full
0 load_kcontext_steal  0 mesq_qf_switch_head_failed
0 lock_kcontext        0 mesq_qf_unexpected_error
0 unlock_kcontext      0 mesq_noop_unexpected_error
0 get_kcontext_cbr     0 mesq_noop_lb_overflow
0 get_kcontext_cbr_busy 0 mesq_noop_qlimit_reached
0 lock_async_resource  0 mesq_noop_amo_nacked
```

```
0 unlock_async_resource      0 mesq_noop_put_nacked
0 steal_user_context         0 mesq_noop_page_overflow
0 steal_kernel_context       0 implicit_abort
0 steal_context_failed       0 implicit_abort_retried
```

... *and much more*

For a usage statement, once the `grustats` command is executing, enter the letter `h` for help. A usage statement appears, as follows:

```
Intstats help:
  h           - help (this screen)
  q           - quit
  r           - reset command-start statistics
  t or <TAB> - toggle between total and incremental mode
  CTL-L      - redraw screen

              CR - to return to display
```

GRU Software Functions

This chapter describes software functions that can be used on the global reference unit (GRU). For a description of the GRU, see Chapter 1, "Altix UV GRU Direct Access API" on page 1. This chapter describes a subset of the `/usr/include/uv/gru/gru_instructions.h` file.

Checking the Status of GRU Operations

This section describes software functions used for checking the status of GRU operations, as follows:

```
extern int gru_check_status_proc(gru_control_block_t *cb); extern int gru_wait_proc(gru_control_block_t *cb);
extern int gru_wait_abort_proc(gru_control_block_t *cb);
```

```
extern void gru_abort(int, gru_control_block_t *cb, char *str);
```

The `gru_check_status_proc()` and `gru_wait_proc()` functions return one of the following GRU control block status (CBS) values:

```
CBS_IDLE
CBS_EXCEPTION
CBS_ACTIVE
CBS_CALL_OS
```

Displaying GRU Error Information

This section describes software functions used for displaying GRU error information, as follows:

```
extern char *gru_get_cb_exception_detail_str(int ret, gru_control_block_t *cb,
char *buf, int size);
```

GRU Data Transfer Functions

This section describes some GRU data transfer functions.

GRU data transfer functions have some arguments in common with each other:

xtype

xtype - datatype of the transfer. Choose from the following list:

XTYPE_B	byte
XTYPE_S	short (2-byte)
XTYPE_W	word (4-byte)
XTYPE_DW	doubleword (8-byte)
XTYPE_CL	cacheline (64-byte)

exopc

exopc - extended opcode for atomic memory operations (AMO).

AMOs implicit operand opcodes

```
EOP_IR_FETCH /* Plain fetch of memory */
EOP_IR_CLR /* Fetch and clear */
EOP_IR_INC /* Fetch and increment */
EOP_IR_DEC /* Fetch and decrement */
EOP_IR_QCHK1 /* Queue check, 64 byte msg */
EOP_IR_QCHK2 /* Queue check, 128 byte msg */
```

Registered AMOs with implicit operand opcodes

```
EOP_IRR_FETCH /* Registered fetch of memory */
EOP_IRR_CLR /* Registered fetch and clear */
EOP_IRR_INC /* Registered fetch and increment */
EOP_IRR_DEC /* Registered fetch and decrement */
EOP_IRR_DECZ /* Registered fetch and decrement, update on zero*/
```

AMOs with explicit operand opcodes

```
EOP_ER_SWAP /* Exchange argument and memory */
EOP_ER_OR /* Logical OR with memory */
EOP_ER_AND /* Logical AND with memory */
EOP_ER_XOR /* Logical XOR with memory */
```



```
EOP_ER_ADD /* Add value to memory */
EOP_ER_CSWAP /* Compare with operand2, write operand1 if match*/
EOP_ER_CADD /* Queue check, operand1*64 byte msg */
```

Registered AMOs with explicit operand opcodes

```
EOP_ERR_SWAP /* Exchange argument and memory */
EOP_ERR_OR /* Logical OR with memory */
EOP_ERR_AND /* Logical AND with memory */
EOP_ERR_XOR /* Logical XOR with memory */
EOP_ERR_ADD /* Add value to memory */
EOP_ERR_CSWAP /* Compare with operand2, write operand1 if match*/
```

AMOs with extened opcodes in DSR

```
EOP_XR_CSWAP /* Masked compare exchange */
```

hints

```
IMA_CB_DELAY /* hold read responses until status changes */
```

Functions for GRU Instructions

This section contains functions for GRU instructions, as follows:

- nelem and stride are in elements
- tri0/tri1 is in bytes for the beginning of the data segment.

```
static inline void gru_vload(gru_control_block_t *cb, void *mem_addr,
    unsigned int tri0, unsigned char xtype, unsigned long nelem,
    unsigned long stride, unsigned long hints) {
    struct gru_instruction *ins = (struct gru_instruction *)cb;

    ins->baddr0 = (long)mem_addr;
    ins->nelem = nelem;
    ins->opl_stride = stride;
    gru_start_instruction(ins, __opword(OP_VLOAD, 0, xtype, IAA_RAM, 0,
        (unsigned long)tri0, hints));
}
```

```
static inline void gru_vstore(gru_control_block_t *cb, void *mem_addr,
    unsigned int tri0, unsigned char xtype, unsigned long nelem,
    unsigned long stride, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)mem_addr;
    ins->nelem = nelem;
    ins->opl_stride = stride;
    gru_start_instruction(ins, __opdword(OP_VSTORE, 0, xtype, IAA_RAM, 0,
        tri0, hints));
}
```

```
static inline void gru_ivload(gru_control_block_t *cb, void *mem_addr,
    unsigned int tri0, unsigned int tril, unsigned char xtype,
    unsigned long nelem, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)mem_addr;
    ins->nelem = nelem;
    ins->tril_bufsize_64 = tril;
    gru_start_instruction(ins, __opdword(OP_IVLOAD, 0, xtype, IAA_RAM, 0,
        tri0, hints));
}
```

```
static inline void gru_ivstore(gru_control_block_t *cb, void *mem_addr,
    unsigned int tri0, unsigned int tril,
    unsigned char xtype, unsigned long nelem, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)mem_addr;
    ins->nelem = nelem;
    ins->tril_bufsize_64 = tril;
    gru_start_instruction(ins, __opdword(OP_IVSTORE, 0, xtype, IAA_RAM, 0,
        tri0, hints));
}
```

```
static inline void gru_vset(gru_control_block_t *cb, void *mem_addr,
    unsigned long value, unsigned char xtype, unsigned long nelem,
    unsigned long stride, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;
```

```
ins->baddr0 = (long)mem_addr;
ins->op2_value_baddr1 = value;
ins->nelem = nelem;
ins->op1_stride = stride;
gru_start_instruction(ins, __opdword(OP_VSET, 0, xtype, IAA_RAM, 0,
    0, hints));
}

static inline void gru_ivset(gru_control_block_t *cb, void *mem_addr,
    unsigned int tril, unsigned long value, unsigned char xtype,
    unsigned long nelem, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)mem_addr;
    ins->op2_value_baddr1 = value;
    ins->nelem = nelem;
    ins->tril_bufsize_64 = tril;
    gru_start_instruction(ins, __opdword(OP_IVSET, 0, xtype, IAA_RAM, 0,
        0, hints));
}

static inline void gru_vflush(gru_control_block_t *cb, void *mem_addr,
    unsigned long nelem, unsigned char xtype, unsigned long stride,
    unsigned long hints)
{
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)mem_addr;
    ins->op1_stride = stride;
    ins->nelem = nelem;
    gru_start_instruction(ins, __opdword(OP_VFLUSH, 0, xtype, IAA_RAM, 0,
        0, hints));
}

static inline void gru_nop(gru_control_block_t *cb, int hints) {
    struct gru_instruction *ins = (void *)cb;

    gru_start_instruction(ins, __opdword(OP_NOP, 0, 0, 0, 0, 0, hints)); }
}
```

3: GRU Software Functions

```
static inline void gru_bcopy(gru_control_block_t *cb, const void *src,
    void *dest,
    unsigned int tri0, unsigned int xtype, unsigned long nelelem,
    unsigned int bufsize, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

#ifdef UV_REV_1_WARS
    if (tri0 + bufsize * 64 >= 8192)
        gru_abort_bcopy_war(0);
    if (((tri0 + bufsize * 64) & 8191) == 0) // GRU 1.0 WAR
        gru_abort_bcopy_war(1);
    if (bufsize > 128) // GRU 1.0 WAR
        gru_abort_bcopy_war(2);
#endif
    ins->baddr0 = (long)src;
    ins->op2_value_baddr1 = (long)dest;
    ins->nelem = nelelem;
    ins->tril_bufsize_64 = bufsize;
    gru_start_instruction(ins, __opdword(OP_BCOPY, 0, xtype, IAA_RAM,
        IAA_RAM, tri0, hints));
}

static inline void gru_bstore(gru_control_block_t *cb, const void *src,
    void *dest, unsigned int tri0, unsigned int xtype,
    unsigned long nelelem, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)src;
    ins->op2_value_baddr1 = (long)dest;
    ins->nelem = nelelem;
    gru_start_instruction(ins, __opdword(OP_BSTORE, 0, xtype, 0, IAA_RAM,
        tri0, hints));
}

static inline void gru_gamir(gru_control_block_t *cb, int exopc, void *src,
    unsigned int xtype, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)src;
#ifdef UV_REV_1_WARS
    ins->nelem = 1; // GRU 1.0 WAR
#endif
}
```

```
#endif
gru_start_instruction(ins, __opdword(OP_GAMIR, exopc, xtype, IAA_RAM, 0,
    0, hints));
}

static inline void gru_gamirr(gru_control_block_t *cb, int exopc, void *src,
    unsigned int xtype, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)src;
#ifdef UV_REV_1_WARS
    ins->nelem = 1; // GRU 1.0 WAR
#endif
gru_start_instruction(ins, __opdword(OP_GAMIRR, exopc, xtype, IAA_RAM, 0,
    0, hints));
}

static inline void gru_gamer(gru_control_block_t *cb, int exopc, void *src,
    unsigned int xtype,
    unsigned long operand1, unsigned long operand2,
    unsigned long hints)
{
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)src;
    ins->op1_stride = operand1;
    ins->op2_value_baddr1 = operand2;
#ifdef UV_REV_1_WARS
    ins->nelem = 1; // GRU 1.0 WAR
#endif
gru_start_instruction(ins, __opdword(OP_GAMER, exopc, xtype, IAA_RAM, 0,
    0, hints));
}

static inline void gru_gamerr(gru_control_block_t *cb, int exopc, void *src,
    unsigned int xtype, unsigned long operand1,
    unsigned long operand2, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)src;
    ins->op1_stride = operand1;
```

3: GRU Software Functions

```
    ins->op2_value_baddr1 = operand2;
#ifdef UV_REV_1_WARS
    ins->nelem = 1;    // GRU 1.0 WAR
#endif
    gru_start_instruction(ins, __opdword(OP_GAMERR, exopc, xtype, IAA_RAM, 0,
    0, hints));
}
```

```
static inline void gru_gamxr(gru_control_block_t *cb, void *src,
    unsigned int tri0, unsigned long hints) {
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)src;
    ins->nelem = 4;
    gru_start_instruction(ins, __opdword(OP_GAMXR, EOP_XR_CSWAP, XTYPE_DW,
    IAA_RAM, 0, 0, hints));
}
```

```
static inline void __gru_mesq(gru_control_block_t *cb, void *queue,
    unsigned long tri0, unsigned long nelem,
    unsigned long hints)
{
    struct gru_instruction *ins = (void *)cb;

    ins->baddr0 = (long)queue;
    ins->nelem = nelem;
    gru_start_instruction(ins, __opdword(OP_MESQ, 0, XTYPE_CL, IAA_RAM, 0,
    tri0, hints));
}
```

```
#if !defined(UV_REV_1_WARS)
static inline void gru_mesq(gru_control_block_t *cb, void *queue,
    unsigned long tri0, unsigned long nelem,
    unsigned long hints)
{
    __gru_mesq(cb, queue, tri0, nelem, hints); } #else extern void gru_mesq(gru_control_block_t *cb, void *
    unsigned long tri0, unsigned long nelem,
    unsigned long hints);
#endif
```

```
static inline unsigned long gru_get_amo_value(gru_control_block_t *cb) {
    struct gru_instruction *ins = (void *)cb;
```

```
    return ins->avalue;
}

static inline int gru_get_amo_value_head(gru_control_block_t *cb) {
    struct gru_instruction *ins = (void *)cb;

    return ins->avalue & 0xffffffff;
}

static inline int gru_get_amo_value_limit(gru_control_block_t *cb) {
    struct gru_instruction *ins = (void *)cb;

    return ins->avalue >> 32;
}

static inline union gru_mesqhead gru_mesq_head(int head, int limit) {
    union gru_mesqhead mqh;

    mqh.head = head;
    mqh.limit = limit;
    return mqh;
}

#define GRU_EXC_STR_SIZE 1024

/*
 * Control block definition for checking status */ struct gru_control_block_status {
    unsigned int icmd :1;
    unsigned int ima :3;
    unsigned int reserved0 :4;
    unsigned int unused1 :24;
    unsigned int unused2 :24;
    unsigned int istatus :2;
    unsigned int isubstatus :4;
    unsigned int unused3 :2;
};

/* Get CB status */
static inline int gru_get_cb_status(gru_control_block_t *cb) {
```

3: GRU Software Functions

```
struct gru_control_block_status *cbs = (void *)cb;

return cbs->istatus;
}

/* Get CB message queue substatus */
static inline int gru_get_cb_message_queue_substatus(gru_control_block_t *cb) {
    struct gru_control_block_status *cbs = (void *)cb;

    return cbs->isubstatus & CBSS_MSG_QUEUE_MASK; }

/* Get CB substatus */
static inline int gru_get_cb_substatus(gru_control_block_t *cb) {
    struct gru_control_block_status *cbs = (void *)cb;

    return cbs->isubstatus;
}

/*
 * User interface to check an instruction status. UPM and exceptions
 * are handled automatically. However, this function does NOT wait
 * for an active instruction to complete.
 */
static inline int gru_check_status(gru_control_block_t *cb) {
    struct gru_control_block_status *cbs = (void *)cb;
    int ret;

    __barrier();
    ret = cbs->istatus;
    /* Must call if IDLE to update statistics */
    if (ret != CBS_ACTIVE)
        ret = gru_check_status_proc(cb);
    return ret;
}

/*
 * User interface (via inline function) to wait for an instruction
 * to complete. Completion status (IDLE or EXCEPTION is returned
 * to the user. Exception due to hardware errors are automatically
 * retried before returning an exception.
 */
```



```
*
*/
static inline int gru_wait(gru_control_block_t *cb) {
    return gru_wait_proc(cb);
}

/*
 * Wait for CB to complete. Aborts program if error. (Note: error does NOT
 * mean TLB mis - only fatal errors such as memory parity error or user
 * bugs will cause termination.
 */
static inline void gru_wait_abort(gru_control_block_t *cb) {
    gru_wait_abort_proc(cb);
}

/*
 * Get a pointer to a control block
 * gseg - GSeg address returned from gru_get_thread_gru_segment()
 * index - index of desired CB
 */
static inline gru_control_block_t *gru_get_cb_pointer(gru_segment_t *gseg,
    int index)
{
    return (void *)gseg + GRU_CB_BASE + index * GRU_HANDLE_STRIDE; }

/*
 * Get a pointer to a cacheline in the data segment portion of a GSeg
 * gseg - GSeg address returned from gru_get_thread_gru_segment()
 * index - index of desired cache line
 */
static inline void *gru_get_data_pointer(gru_segment_t *gseg, int index) {
    return (void *)gseg + GRU_DS_BASE + index * GRU_CACHE_LINE_BYTES; }

/*
 * Convert a vaddr into the tri index within the GSEG
 * vaddr - virtual address of within gseg
 */
static inline int gru_get_tri(void *vaddr) {
    return ((unsigned long)vaddr & (GRU_MIN_GSEG_PAGESIZE - 1)) - GRU_DS_BASE; }

/*
```

3: GRU Software Functions

```
* Decode and print a GRU instruction.  
*/  
void gru_print_cb_detail(const char *id, int ret, void *cb);
```

Index

A

accessing the Altix UV GRU direct access API, 4

C

checking the status of GRU operations, 27

D

data transfer data type

 exopc, 28

 xtype, 28

direct GRU access overview, 4

displaying GRU error information, 27

E

environment variables

 GRU_CCH_REQUEST_SLICE, 19

 GRU_STATISTICS_FILE, 20

 GRU_TLB_PRELOAD, 20

 GRU_TLBMISS_MODE, 19

 GRU_TRACE_EXCEPTIONS, 21

 GRU_TRACE_FILE, 20

 GRU_TRACE_INSTRUCTION_RETRY, 21

 GRU_TRACE_INSTRUCTIONS, 21

F

functions for GRU instructions, 29

G

global reference unit, 1

GRU

 gru_instructions.h, 9

 initiate GRU functions, 9

 memory access functions, 9

 resource allocators, 5

 See "global reference unit", 1

 software functions, 27

 checking status of GRU operations, 27

 data transfer functions, 27

 displaying GRU error information, 27

GRU data transfer functions, 28

GRU files in /proc, 22

GRU library program example, 14

grustats command, 24

I

introduction, 1

M

MPT address mapping functions, 12

O

overview of direct GRU access, 4

P

/proc GRU files, 22

R

required RPMs

- gru-devel, 4
- gru_alloc-devel, 4
- libgru-devel, 4
- xpmem-devel, 4

mpi, shmem, sgiupc, 4

X

XPMEM library functions, 11

S

SGI APIs