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

007–5668–001

Record of Revision

Version	Description
001	June 2010 Original Printing.

Contents

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 and Other Sources

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.

- *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.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
variable	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

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.

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 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.

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*.

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 fuctions, 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.

You can find more detailed information in the following man pages:

- "`gru_temp_reserve`(3)" on page 4
- "`gru_pallocate`(3)" on page 6
- "`gru_resource`(3)" on page 7

Use the `man`(1) command to view these man pages online. For your convenience, copies of the GRU-related man pages are included in the following section.

GRU Man Pages

This section contains GRU-related man pages.

gru_temp_reserve(3)

NAME

gru_temp_reserve, gru_temp_release - temporary GRU resource allocator

SYNOPSIS

```
#include <gru_alloc.h>

void gru_temp_reserve(gru_alloc_thdata_t *gat);
int gru_temp_reserve_try(gru_alloc_thdata_t *gat);
void gru_temp_release(void);

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

LIBRARY

-lgru_alloc

DESCRIPTION

The gru_temp_reserve() and gru_temp_reserve_try() functions will allocate and reserve the temporary use GRU resources for a thread. The gru_alloc_thdata_t structure returne in gat describes the number and locations of the temporary use GRU resources which may be used until the next call to gru_temp_release().

The fields are defined, as follows:

gruseg	The GRU segment
cbp	A convenient pointer to the first control block (CB). Equal to gru_get_cb_pointer(gat->gruseg, 0).
dsegp	A pointer to data segment space (DSEG) space available for temporary use.

`cb_cnt`	The number of consecutive CBs in the GRU segment that are available for temporary use.
`dseg_size`	The size of the DSEG region available for temporary use (bytes).

The first call to `gru_temp_reserve()` will allocate a GRU segment for the calling thread, and this same segment will be assigned to the thread for use after any call to `gru_temp_reserve()`.

Every call to `gru_temp_reserve()` sets a thread-private "temporary resources in use" (TRU) flag. The temporary GRU resources identified by the `gat` structure are valid and may be referenced only when the TRU flag is set. Note that later calls to `gru_temp_reserve()` may return different values in the `gat` structure.

The program will abort if the TRU flag is already set when a call is made to `gru_temp_reserve()` or `gru_pallocate()`.

The GRU allocation library attempts to provide a quantity of temporary use GRU resources that is equal to the quantity on each UV hub divided by the number of processors per hub. This quantity will be reduced by any GRU resources permanently allocated via the `gru_pallocate()` function.

SUGGESTED USAGE CONVENTIONS

The above usage rules suggest two natural usage conventions that are equally valid:

A. Users surround code blocks that use temporary GRU resources with `gru_temp_reserve()` and `gru_temp_release()` calls.

or

B. Users should insert calls to `gru_temp_reserve()` at the beginning of each GRU-using function and calls to `gru_temp_release()` at each return point for that function. In addition, every function call site that might end up calling a GRU function with temporary GRU resources should have a call to `gru_temp_release()` prior to the call site and a call to `gru_temp_reserve()` upon return.

Note that use of GRU functions with temporary storage in signal handlers is dangerous. The program will abort if the TRU flag is set when a signal handler is entered that also calls `gru_temp_reserve()`.

ENVIRONMENT VARIABLES

See the `gru_resource`(3) man page for information about environment variables that can control the amount of GRU resources that are allocated.

RETURN VALUE

`gru_temp_reserve_try` returns 0 if able to reserve the temporary GRU resources,

and -1 otherwise.

Failure to reserve temporary resources results from a previous reservation on the temporary resource still being in effect. `gru_temp_reserve` aborts if unable to reserve the temporary GRU resources.

NOTES

The deprecated `gru_all_reserve()` function has the same effect as `gru_temp_reserve()`.

The deprecated `gru_all_release()` function has the same effect as `gru_temp_release()`.

SEE ALSO

`gru_pallocate`(3), `gru_all_reserve`(3), and `gru_resource`(3)

gru_pallocate(3)

NAME

`gru_pallocate` - permanently allocate GRU resources

SYNOPSIS

```
#include <gru_alloc.h>

int gru_pallocate(int num_cbs, int dseg_sz, gru_segment_t **gruseg,
      int *cbnum, void **dseg);

int gru_pallocate_dseg_granularity(void);
```

LIBRARY

`-lgru_alloc`

DESCRIPTION

The gru_pallocate() function will permanently reserve a specified number of GRU control blocks (CBs) and data segment space (DSEG).

Arguments are, as follows:

num_cbs	(input) the number of CBs desired.
dseg_sz	(input) the number of bytes of DSEG space desired. dseg_sz must be a multiple of the DSEG allocation granularity.
gruseg	(output) assigned the pointer to the GRU segment containing the returned resources.
cbnum	(output) assigned the ordinal value of the first CB in the GRU segment that is part of the allocation.
dseg	(output) assigned the pointer to the allocated DSEG space.

The gru_pallocate() function may not be called between calls to gru_temp_reserve() and gru_temp_release(). After gru_pallocate() is called, the amount of GRU resources available to the caller of gru_temp_reserve() will be decreased.

The gru_pallocate_dseg_granularity() function returns the DSEG allocation granularity, which is the smallest number of bytes of DSEG space that may be allocated.

RETURN VALUE

gru_pallocate() returns 0 on success, -1 on error with one of the following errno values set:

ENOMEM - the library GRU segment local to this thread had insufficient CB or DSEG space to satisfy the request.

EINVAL – the dseg_sz value is not a multiple of the DSEG allocation granularity.

SEE ALSO

gru_temp_reserve(3), gru_temp_release(3)

gru_resource(3)

NAME

gru_resource - tuning the GRU allocator run-time library

LIBRARY

libgru_alloc run-time library

DESCRIPTION

The GRU allocator run-time library is linked in to some programs and libraries to manage available GRU resources. The amount of GRU resource that can be allocated defaults to a logical CPU's share of the GRU resources on an Altix UV hub. However, the user can modify and tune the quantities of GRU resource by setting environment variables, as described in the following section of this man page.

ENVIRONMENT VARIABLES

GRU_RESOURCE_FACTOR	Multiplies the quantity of control blocks (CB) and data segment space (DSEG) resources assigned to each thread by the factor given. For example, when parallel jobs are run with only one user thread per core, a factor of 2 could be specified. If only one GRU-using thread or process will be run on each socket, and each socket had 16 hyperthreads, then a factor of 16 could be specified. If GRU_THREAD_CBS or GRU_THREAD_DSEG_SZ are specified, they override GRU_RESOURCE_FACTOR.
GRU_THREAD_CBS	Overrides the number of per-thread CBs assigned to the caller of gru_temp_reserve(). The default is a processor's fair portion of the available CBs, which is 8 on systems with 8 cores per socket and 10 on systems with 6 cores per socket.
GRU_THREAD_DSEG_SZ	Overrides the amount of per-thread DSEG space assigned to the caller of gru_temp_reserve(). The default is a processor's fair portion of the available DSEG space, which is 2048.

SEE ALSO

cpumap(1)

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. These functions use an ordinary virtual address or a GRU-mapped virtual address to reference the remote memory.

The interfaces to these functions are viewable in the `uv/gru/gru_instructions.h` header file installed by the `gru-devel` RPM. Use a hardware reference manual to get functional descriptions of these operations. *TBD? What hardware reference manual?*

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 sytems offers many options for doing this, as follows:

- `mmap`
- System V shared memory
- memory sharing among `pthreads`
- 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-noship` 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 the SGI internal XPMEM API document. TBD

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.

  MPI_GAM_NONE       - not referenceable by load/store or GRU
  MPI_GAM_CPU_NONCOH - Altix 3700 noncoherent
  MPI_GAM_CPU        - if referencable by load/store only
  MPI_GAM_GRU        - if referenceable by GRU only
  MPI_GAM_CPU_PREF   - if referenceable by either load/store
                         or GRU, preferred by load/store
  MPI_GAM_GRU_PREF   - 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. Contigous groups of ranks within a hostcan 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.

acc_mode	Chooses CPU or GRU addressable
MPI_GAM_CPU	Requests CPU address that can be referenced
MPI_GAM_GRU	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 specfied MPI process. See shmem_ptr(3) for more details.

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_STATISTICS_FILE

You can collect statistics of a task's usage of GRU contexts by using this option to specify a statistics file. Whenever a task exits or a GRU context is destroyed, statistics are written to this file. To specify a statistics file, perform the following:

```
setenv GRU_STATISTICS_FILE <filename>
```

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_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
 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, tri1: 0x40, nelem: 0x1
        istatus: IDLE
 Pid: 25276 - gru_wait
        opc: IVSTORE, xtype: DWORD, ima: DelResp, baddr0: 0x0, tri0: 0x0, tri1: 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
```

Index