Emu1 System Programming Guide

Document Number: 150918

Version Number: 2.26

Origination Date: September 18, 2015

Modification Date: February 14, 2019

Emu Solutions, Inc.

1400 East Angela Blvd, Unit 101

South Bend, IN 46617

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This document provides an overview for programming the Emu1 system, covering both the Emu Simulation Environment (`emusim`) and the Emu1 Chick hardware. It is expected that programmers will initially develop, validate, and optimize programs using the simulation environment and then move to execution on the Emu1 Chick hardware.

The goal of `emusim` is to model Emu's first generation system and permit emulation and characterization of programs running on this system. It provides debugging capabilities and insight into execution characteristics that are not yet available on the hardware.

The simulation environment serves as a platform for application development and analysis as well as for architectural design studies. As the system evolves, the simulation environment will continue to evolve as well and work to improve the simulator fidelty is on-going.

Note: The term "threadlets" was previously used to refer to the program threads running on the nodelets/Gossamer Cores (e.g, threadlets run on nodelets). This term has been eliminated in favor of just using "threads" everywhere. Some documentation and simulator output may not have eliminated all references to "threadlets" yet.

1.2 System Overview

This document describes Emu's first generation large memory system. This section introduces the system organization, memory layout, and program execution with threads.

1.2.1 Emu System Organization

An Emu system is a shared memory multiprocessor consisting of a number of nodes and an IO system connected by an interconnection network. Each node contains one or more Stationary Cores (SCs), one or more Gossamer Nodelets, and a Migration / DMA Engine. A Gossamer Nodelet consists of one or more Gossamer Cores (GCs) tightly coupled to a bank of memory. Figure 1.1 illustrates the high level architecture.

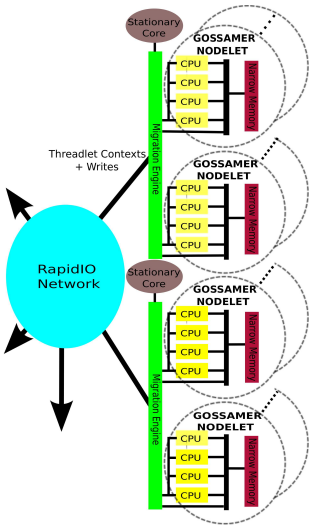


Figure 1.1: Emu1 System Architecture

Emu1 is a specific implementation of this architecture. Emu1 nodes each contain 2 Stationary Cores and 8 Gossamer Nodelets with each nodelet comprised of multiple Gossamer Cores and an 8 bit wide (plus ECC) bank of DDR4 DRAM containing up to four ranks of 4 or 8 bit wide DRAM chips. Specifically, the initial Emu1 product utilizes 12 2Gx4 DRAM modules, implementing 8 GB of memory per nodelet. In addition, the Emu1 nodes will contain a Solid State Disk that is connected to the node through a PCIExpress link. Data transfers to and from the SSD are initiated and controlled by the SC. It is anticipated that all transfers from memory to the SSD must read data from the local memory on the node, but transfers from SSD to memory may go anywhere in the Gossamer address space (The initial implementation may only support transfers to the local memory). Emu1 is planned to include 0.5, 1 or 2 TB of solid state disk on each node (size is a customer option). In addition, the SC is connected to a small FLASH NVRAM that is used to hold the boot program and some system parameters, and will also be connected to a small block of DRAM that is private to the SC and not part of the Gossamer address space. In Emu1, the interconnect network is a switched RapidIO network, and each node connect to this network using six 4-lane wide RapidIO v2.1 interfaces. Routing in the interconnect network is based on system addresses, all of which are visible from any node in the system. In Emu1, the Stationary Cores are implemented using an off-the-shelf microprocessor while the remaining node hardware blocks (the Gossamer nodelets, the Migration Engine, the DDR4 DRAM controllers, the RapidIO network controllers) are implemented on an FPGA connected to narrow-channel DRAM.

Emu1 Chick Model: This is a small system consisting of up to 8 nodes fitting into a desktop tower chassis. This model does not use any switches in the RapidIO network but instead directly connects nodes to one another. In the 8 node version, each node is connected to 6 of the other 7 nodes; messages being sent to the unconnected node must hop through an intermediate node. As the Chick is Emu's first product, most efforts regarding program optimization and simulator design/implementation have been targeting this model. Note that the remainder of this document assumes an 8 node system.

1.2.2 Memory Organization

The Emu system architecture implements a Partitioned Global Address Space (PGAS). In a PGAS, all memory in the system has unique addresses that are visible to everywhere in the system. However, the range of addresses need not be contiguous, and the memory may be distributed across various partitions (nodes and nodelets for Emu systems) with different latencies and bandwidths available depending on the proximity between a device requesting a memory access and the memory it accesses. In the case of Emu1, the primary memory connected to the nodelets is DDR4 DRAM. However, each node also contains a number of hardware registers and static RAMs that are mapped into the PGAS space and may be accessed by programs for specialized purposes. As required in a PGAS, all processing elements have mechanisms that allow them to access all memory that is part of the PGAS. In some cases, this access may be performed indirectly, using other devices as an agent to assist in the process. Specifically, SCs in Emu1 nodes can directly access the FLASH NVRAM in their own node and all DRAM in the nodelets of that node, as well as various static RAMs and control registers. In addition, the SCs can directly perform writes (but not reads) to individual memory locations on other nodes and reads and writes to memory locations in the IO system. GCs in Emu1 nodes can directly access DRAM memory on their own nodelet, as well as certain static RAMs and control registers. Accesses other than these are performed indirectly, as discussed in the following paragraphs. Physical per-nodelet memory sizes on the hardware are 1GiB for the stack and 6GiB for the system heap. Physical per-nodelet memory sizes on the simulator are 1GiB for the stack and 2GiB for the system heap.

1.2.3 Execution Model

Programs executing on the Gossamer Cores are known as *Threads*. A thread has a lightweight context that the Emu system hardware automatically transfers from nodelet to nodelet so that all of the memory references it makes occur at the nodelet where it is executing. In other words, if the thread attempts access to a memory location that is part of the memory of a nodelet other than the one where it is currently executing, hardware suspends its execution, transmits its context to the nodelet containing the referenced memory, and restarts the thread on that nodelet. This process is known as a migration. Once the migration occurs, the actual memory reference always occurs locally. Since the bandwidth required to migrate a thread context is only slightly more than the bandwidth consumed transmitting a memory reference across the network, this strategy is performance neutral when only a single reference occurs at a nodelet, and is a significant win any time multiple references are made to memory at that nodelet. Since, in practice, most algorithms usually make at least a few such references between migrations, significantly less bandwidth is consumed overall than in a conventional computer.

Programs executing on the Stationary Cores are known as *Stationary Threads*. Stationary threads are the type of programs typically run on conventional computers. They always execute entirely on the same core where they were started, although they may create threads to perform functions on their behalf. For Emu systems, stationary threads are used primarily for control and for performing system services. Generally, the main work done on the system will be performed by groups of threads executing on the Gossamer Cores. The stationary threads in the system are primarily intended to perform the following functions, although they are not limited to these:

- Run the Operating System (typically Linux)

- Perform input/output operations and run I/O Drivers
- Perform exception handling for most exceptions that occur on Gossamer cores
- Initialize and boot the system
- Perform various system services for threads running on Gossamer Cores

As noted previously, Stationary Cores are not capable of reading memory except on their own node, and have limited capability to write memory on nodes other than their own. When such memory accesses are required, they typically create threads to perform remote memory accesses on their behalf. Also, they may program the Migration Engine or the Stationary Core Proxy to perform block copies of memory between PGAS memory on the same or different nodes.

Note: At this time, user programs are launched by the Stationary Cores and run exclusively on the Gossamer Cores, with the Stationary Cores used only to handle exceptions, I/O, etc. User programs do not run on the SCs. The simulator does not explicitly model the SCs.

1.3 Overview of the Simulation Environment

The Emu Simulation Environment (`emusim`) provides a simulator to emulate application execution on the Emu1 system as well as a visualization tool to view the results of simulation. The simulator can run in functional simulation or performance simulation modes. The functional simulation runs applications with a simplified system model to allow quicker simulation for debugging and correctness testing. The performance mode provides timing and statistics for application performance starting at a desired location (future development will allow the user to stop a timing simulation as well) to predict application performance on the actual Emu1 hardware. Statistics can also be generated for the visualization tool to show the thread count and depth of various system queues at a given sampling rate to assist in identifying bottlenecks and/or load imbalances.

1.4 Overview of the Hardware Environment

The Emu1 Chick system runs a Linux OS on the SCs. Programs are launched from the SC and execute in the GCs, returning control to the SC on completion. A diagnostic tool is provided to enable inspection of various control registers and nodelet memory from the SC. At this time, there are limited debugging and analysis tools available, so initial program development and analysis should be performed with the simulator before moving to the hardware.

1.5 Organization

This document is organized as follows:

- Chapter 2 describes the thread creation and control routines.
- Chapter 3 describes the memory configuration, data distribution and data allocation routines.

- Chapter 4 describes Emu architecture-specific operations such as atomic memory operations and system queries.
- Chapter 5 provides a summary of available libraries.
- Chapter 6 provides a set of example programs to illustrate program development and analysis.
- Chapter 7 describes the simulation environment and the statistics that can be collected.
- Chapter 8 describes the visualization tools that can be used to analyze the simulator output.
- Chapter 9 describes how to start up and configure the Emu1 Chick system.
- Chapter 10 describes how to execute programs on the Emu1 Chick system.

1.6 Version History

Table 1.1 gives the version history for this document.

Version	Date	Changes
2.26	2/13/2019	Minor cleanup of CilkPlus toolchain and ongoing efforts. Add `--memory_trace` flag for the simulator. Fix bug in text of `mw_localmalloc`.
2.25	12/18/2018	Clarifications to Sec. 5.5.
2.24	11/30/2018	Add Sec. 5.5 discussing the newly create multinode file I/O functions.
2.23	11/27/2018	Updated the `.vsf` subsection in Sec. 7.6 to clarify what is presented when running an untimed simulation.
2.22	11/9/2018	Add CilkPlus toolchain info including the migrate hint and spawn_at constructs and reducers info. Update toolchain next steps. Comment out discussion of spawn depth since it is not used.
2.21	10/3/2018	Added new thread management intrinsics.
2.20	9/28/2018	Updated section on emu_c_utils library. New flags for simulator.
2.19	9/12/2018	Updates for the September release, update program delimiter and file I/O working directories.
2.18	8/13/2018	Add section on linking with x86 emu_c_utils, update examples and code for simple array and random access.
2.17	8/10/2018	Add emu_c_utils documentation, Update simulator documentation
2.16	7/3/2018	Updated system heap size for the hardware, Add mw_mallocrepl
2.15	6/8/2018	Clean up intro, add HW environment, minor clean-up and clarifications of data allocation, add discussion of C++ library
2.14	5/24/2018	Note that document assumes 8 node system. Update system configuration. Clarify mw_malloc2d structure and usage. Clarify mw_arrayindex
2.13	4/15/2018	Note use of volatile with CLOCK(), cilk_for not fully supported on HW, minor updates RE HW.
2.12	1/30/2018	Update portions of Chapter 7 and Chapter 10 related to toolchain updates. Formatting updates.

2.11	1/16/2018	Separate Emu Chick execution into configuration and execution. Update single node configuration and execution, add multinode configuration and execution, including chick-helper scripts.
2.10	1/5/2018	Minor update to configuration parameters.
2.9	9/21/17	Hardware 2.1.1, sc-driver 2.0.6, sc-driver-tests 2.0.12: Update Section 7.3 to remove stoptiming. Add clarification of how to use CLOCK for timing in Sec.4.5. Update Sec. 5 libraries and remove special print functions e.g. printHex. Update hardware instructions for loop test, diagnostic tool, logs, and exception codes.
2.8	9/18/17	Hardware v2.0.23, sc-driver v1.3.3, sc-driver-tests v1.0.2: Update Section 9 with latest usage for emu_handler_and_loader. Add section on sc-driver-tests and emu_loop_test.
2.7	8/10/17	Hardware v2.0.16, sc-driver v1.2.0: Update Section 9 with latest flags for emu_handler_and_loader
2.6	7/24/17	Add initial information for hardware execution
2.5	4/3/16	Remove mallocstripe, update toolchain path, minor fixes
2.4	12/7/16	Address comments.
2.3	11/20/16	Begin updates for latest toolchain changes and add placeholder for chapter on Emu Chick Execution
2.2	08/05/16	Added copyright line and removed MW references
2.1	06/07/16	Update for latest toolchain changes including musl library, replicated functions, and simulator updates. • Introduction: Added a short paragraph on the Chick system. • Threadlet Creation and Control: Added notes to cilk_spawn for migrate to pointer in arg list, added note to cilk-for that it is best for local spawns. • Data Allocation and Distribution: update and reorganize all the mallocs and replicated init functions • Architecture Specific Operations: added resize and reschedule. Updated status of others and added descriptive text. Added note that atomics work ONLY for 64-bit data types. • Libraries (formerly System Calls): updated to reflect availability of musl libC • Example Programs: ensure that all run with latest toolchain and update example/output as needed. • Simulation Execution: Updated to match May 2016 simulator release • Statistics Visualization and Analysis Tool: Updated to match May 2016 release.
2.0	12/01/15	Major update for initial release of the programming toolchain and simulation environment to external users.

1.0	09/18/15	Used MemWeb API document as basis for this document. Updated to become MW1 programming guide including: • New memory model and allocation functions • New code generation toolchain • New ISA version of simulator (timsim)

Table 1.1: Version History.

Chapter 2

Thread Creation and Control

2.1 Overview

Execution in Emu systems is performed by one or more *threads* that run in parallel on the same or multiple Gossamer nodelets. The Cilk programming language has been chosen to support thread spawning and synchronization using the Cilk functions shown in Table 2.1. Note that the Emu architecture supports thread spawning and synchronization directly in hardware and does not use the Cilk software runtime. This reduces the overhead involved in managing threads.

Table 2.1: Cilk Support for Spawning and Synchronization

Attribute	**Functionality**
cilk_spawn	Indicates that the function can run in parallel with the caller. Typically causes a new thread to be spawned to execute the function.
cilk_sync	Indicates that the current function cannot continue in parallel with its children and must wait for all children to complete.
cilk_for	Replaces a normal for loop to indicate that the loop iterations can execute in parallel.

Note that `spawn` was part of the original Cilk implementation. The `cilk_spawn`, `cilk_sync`, and `cilk_for` are all part of Intel's Cilk Plus which evolved from the original Cilk.[1] Cilk Plus also provides additional functionality that is not currently implemented in our system.

2.2 Cilk Functions

2.2.1 cilk_spawn

```
[var = ] cilk_spawn function_name(arguments)
```

Functionality: The `cilk_spawn` keyword modifies a function call statement to tell the runtime system that the function may (but is not required to) run in parallel with the caller (referred to

[1] https://www.cilkplus.org

as the parent). This typically results in a new thread being spawned to execute the function. If a thread cannot be spawned, the function is executed as a traditional function call.

The compiler supports the idea of migration during spawning; that is, the programmer can specify that a new thread be started on the nodelet associated with a particular memory address (as opposed to spawning on the local nodelet). This memory address, or "hint", can be provided via `cilk_migrate_hint` (Section 2.2.2) or `cilk_spawn_at` (Section 2.2.3). The child thread's stack will be allocated in the remote nodelet's memory, which can aid in reducing total migrations and distributing work across the system.

(Deprecated in CilkPlus toolchain): If no hint is provided, the compiler will look at the arguments to the spawn and migrate to the processor associated with the first pointer found in the argument list. If there is no pointer, there won't be a migration before the spawn. In addition, if the first pointer turns out to be 0, then the compiler avoids trying to migrate. Relying on the first pointer argument as a hint is deprecated, since the compiler may remove or reorder arguments during the optimization pass.

Notes:

1. A `cilk_spawn` expression must be the only expression on the right side of an assignment and cannot be part of a larger expression. For example, the following is disallowed and will cause the compiler to issue an error diagnostic:

   ```
   var = var2 + cilk_spawn func(args);
   ```

2. If a child thread is spawned, the parent continues execution in parallel with the child thread. There is an implicit `cilk_sync` at the end of every function and every `try` block that contains a `cilk_spawn`. Otherwise, it is up to the programmer to insert a `cilk_sync` before any code that requires all children to have completed. For example, a `cilk_sync` is required before the parent attempts to access a value returned from a function invoked with a `cilk_spawn`.

3. When a `cilk_spawn` is encountered, the hardware decides whether there are sufficient resources to spawn a new thread. If there are not, it will execute the function as a call. The Emu1 system can execute about 256 threads per nodelet concurrently and hold up to approximately 500 more threads. However, the number of threads that are executing and/or held may be lower or higher depending on the program's execution characteristics such as the number of migrations and/or remote memory operations issued and the workload distribution. The `.cdc` and `.vsf` statistics files will indicate the number of successfully spawned threads

4. In the CilkPlus toolchain, when an expression is passed as an argument to a spawned function, the expression is computed within the child thread; previously these expressions were computed in the parent thread. This can change the semantics of the program with respect to with code written for the old toolchain, especially if the any of the argument expressions contain side effects. The toolchain tries to detect this situation and diagnose it with a warning. If the old behavior is desired, move the expression out of the spawn and pass a local variable instead.

   Example:

   The post-increment of `k` will occur within the child thread, creating a race condition.

```
long k = 0;
for (long i = 0; i < 10; ++i)
  cilk_spawn foo(k++);
```

The following warning will be issued:

```
[Warning] Function call argument will be part of the spawn.
 Check for side-effects or move out of the spawn call.
```

Moving the post-increment of `k` out of the spawn expression fixes the race condition and silences the warning.

```
long k = 0;
for (long i = 0; i < 10; ++i) {
  cilk_spawn foo(k);
  k++;
}
```

The compiler checking is not precise; it will raise a warning whenever expressions are used as arguments to a spawn, even if there are no side-effects. This category of warnings can be safely ignored so long as the expression can be safely evaluated within the child thread.

2.2.2 cilk_migrate_hint

```
cilk_migrate_hint(hint);
```

Functionality:

This built-in provides a locality hint for the next `cilk_spawn`. The child will be spawned on the nodelet that `hint` points to. If `hint` is null, the spawn will occur locally.

Notes:

1. See `cilk_spawn_at` for a more convenient syntax for providing spawn hints.
2. Implemented in the CilkPlus toolchain only.

2.2.3 cilk_spawn_at

```
cilk_spawn_at(hint) function_name(arguments)
```

Functionality:

Combines `cilk_spawn` and `cilk_migrate_hint` into a single command to spawn a thread at a particular memory location. The function will be spawned on the nodelet that `hint` points to. If `hint` is null, the spawn will occur locally.

Notes:

1. Because `cilk_spawn_at` is implemented as a macro which emits a `cilk_migrate_hint` followed by a `cilk_spawn`, it cannot be used in contexts where a single statement is expected. For example, this `for` loop will fail to compile without braces.

   ```
   // Not supported, need to add {}
   for (long i=0; i<N; i++)
       cilk_spawn_at(&T[n]) foo(a, i);
   ```

2. Similarly, assigning the result of `cilk_spawn_at` to a variable is not supported. Use `cilk_migrate_hint` (Section 2.2.2) directly in these cases.

3. Implemented in the CilkPlus toolchain only.

2.2.4 cilk_sync

```
cilk_sync
```

Functionality: The cilk_sync statement applies to a task block, which, in most cases, is a function. It indicates that the current function cannot continue in parallel with its spawned children, and must wait. After all the children complete, the current function can continue.

Notes:

There is an implicit `cilk_sync` at the end of every function and every `try` block that contains a `cilk_spawn`. Otherwise, it is up to the programmer to insert a `cilk_sync` before any code that requires all children to have completed. For example, a `cilk_sync` is required before the parent attempts to access a value returned from a function invoked with a `cilk_spawn`.

2.2.5 cilk_for

```
cilk_for (declaration and initialization; conditional expression; increment expression)
body
```

Functionality: The cilk_for loop is a replacement for the normal C/C++ for loop in order to permit loop iterations to run in parallel.

Notes:

The cilk_for statement divides the loop into chunks with each containing one or more loop iterations. During execution of the loop a thread is spawned to execute each chunk of the loop with an implicit `cilk_sync` at the end of the loop to wait for all threads to complete.

The maximum number of iterations in each chunk is the grainsize. The current default grainsize is 16 iterations per thread and the grainsize can be set using the cilk grainsize pragma as shown in

the following example:

```
#pragma cilk grainsize = 4
```

Note that because the loop iterations may be executed in parallel, copies of the local variables are made for each thread but any non-local variables are treated as shared variables that can be accessed by any of the threads.

The `cilk_for` is best used for spawning threads locally as it has not yet been optimized for non-local spawns.

Restrictions: When used with the old toolchain, `cilk_for` can cause hardware crashes. These issues have been fixed in the CilkPlus toolchain.

2.3 Priority, Flow Control, and Control of Parallelism

Emu systems provide mechanisms that control scheduling priority for threads waiting to execute. These mechanisms largely conform to the concepts used in the Cilk programming language, which is intended to be the primary programming language used to program Emu1. The same mechanisms are involved in limiting spawning to programmer defined and system limits, and related mechanisms are used in network flow control. This section describes these mechanisms and their application.

2.3.1 Thread Priority

Each thread in the Emu1 system is assigned a priority from 0 to 255, with 0 being the highest. This priority is stored in the thread and is used in determining scheduling order and in making determinations about whether the thread is permitted to spawn additional threads. User programmers cannot alter the priority of an executing thread, but can determine the priority of the children they spawn, within the limit that a child's priority must always be less than its parent. This restriction is required to conform to the recommended operating behavior of Cilk programs, and also makes sense in general - threads higher in the spawn tree can dispatch many descendants, and therefore typically exercise greater influence over the progress of the application than their descendants.

A Spawn Tree is a collection of threads consisting of a parent, the first thread spawned, and all of its descendants. As discussed above, the parent is always the thread with the highest priority of the members of such a tree. When a new Spawn Tree is created by a thread executing on a Stationary Core, the parent may be created with an arbitrarily chosen priority. By system convention, it is assumed that such trees are normally created with the parent's priority set to 1. Priority 0 is reserved for special system functions that need to have the ability to run ahead of other user functions. By default, the children of any parent will be given priority one more than that of their parent, but an option in the Spawn instruction will be available that adds a constant greater than one to the parent's priority, thus lowering that child's priority. In this way, a parent can create sets of children that execute at different priority levels.

Note: Thread priority is not yet implemented in the simulator.

2.3.2 Nodelet Space Limitations and Control of Parallelism

Each nodelet, and thus the system as a whole, has physical limitations on the amount of storage space available for thread states. To control the total number of threads alive in the system and prevent running out of space for threads, a software based credit mechanism is utilized. If no credits are available when a thread attempts to spawn, the spawn will fail and be handled by the software. The current mechanism in place will turn failed spawns into a function call so that execution can continue.

Hardware based mechanisms are in place to ensure that all threads can be stored in system memory; this is especially necessary when a specific nodelet becomes an algorithmic hot spot. These hardware based mechanisms are frequently referred to in other documentation as hotspot mitigation.

Chapter 3

Data Allocation and Distribution

3.1 Overview

This section provides an overview of data allocation and distribution on Emu systems. Because execution moves to the data in these systems, the way in which the data is distributed in memory helps to define the execution parallelism available to the algorithm.

In Emu1, data can be defined in a single nodelet's memory or can be distributed across multiple nodelets in the system. Table 3.1 summarizes the various functions used to support dynamic memory allocation and free. Table 3.2 summarizes the `replicated` attribute and functions used to access replicated data.

Table 3.1: Support for Dynamic Memory Allocation

Function	Functionality
Data allocation on single nodelet	
`malloc()`	Allocate data structure on local nodelet
`mw_localmalloc()`	Allocate data structure local to (on the same nodelet as) some other data structure
Data allocation on multiple nodelets	
`mw_malloc1dlong()`	Allocate a 1D array of longs striped across multiple nodelets with a single element on each nodelet
`mw_malloc2d()`	Allocate data structure striped across multiple nodelets
Free previously allocated data	
`free()`	Free data allocated by `malloc`
`mw_free()`	Free data allocated by `mw_malloc1dlong` or `mw_malloc2d`
`mw_localfree()`	Free data allocated by `mw_localmalloc`
Access data allocated with `mw_malloc2d`	
`mw_arrayindex()`	Access data allocated by `mw_malloc2d` without causing a migration

Table 3.2: Support for Replicated Data

Attribute/Function	Functionality
Static Replicated Data	
`replicated`	Attribute placed in front of a variable definition to indicate that the variable should be replicated on each nodelet
Dynamic Replicated Data	
`mw_mallocrepl()`	Dynamically allocate a replicated block on each nodelet
Replicated Data Initialization	
`mw_replicated_init()`	Initialize a replicated variable to the same value on each nodelet
`mw_replicated_init_multiple()`	Initialize each instance of a replicated variable to a different value as returned by the user-defined `init_func()` as a function of the node id
`mw_replicated_init_generic()`	Initialize each instance of a replicated variable using the user-defined `init_func()`
Replicated Data Access	
`mw_get_nth()`	Get the instance of a replicated data structure on nodelet n
`mw_get_localto()`	Get the instance of a replicated data structure local to (on the same nodelet as) some other data structure

3.2 Local Dynamic Data Allocation and Free

Local dynamic data allocation utilizes the standard C library functions `malloc` and `free` as defined in `stdlib.h`.

3.2.1 malloc

```
void* malloc( size_t Size)
```

Return Value: Returns a void* to the nodlet's local heap

Arguments:

- Size: the size in bytes to be allocated

Functionality: The `malloc` function specifies the total block size (bytes) to be allocated on the local heap of the nodelet on which the thread is currently executing and returns a pointer to the requested block of memory.

Notes:

Example:

3.2.2 free

```
void free(void* EltPtr)
```

Return Value:

Arguments:

- EltPtr: pointer to a block of memory previously allocated with malloc

Functionality: The `free` function deallocates the memory on the local heap previously allocated by malloc.

Notes:

Example:

3.3 Distributed Dynamic Data Allocation and Free

These functions are used to dynamically allocate data structures on a non-local nodelet or distributed across multiple nodelets in the system. These functions require `memoryweb.h`.

3.3.1 mw_localmalloc

```
void * mw_localmalloc(size_t eltsize, void * localpointer)
```

Return Value: Returns a void * to a block of eltsize memory located on the nodelet that contains localpointer

Arguments:

- eltsize: the size in bytes to be allocated
- localpointer: pointer to a variable on the nodelet where you want the allocation.

Functionality: The `mw_localmalloc` allocates a chunk of at least eltsize on the same nodelet that localpointer points to. A localpointer of zero will cause an exception.

Notes: Defined in `distributed.h`.

Example:

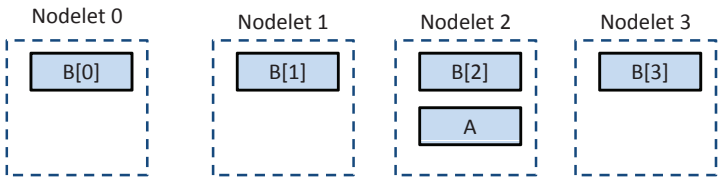


Figure 3.1: Allocating `A` co-located with `B[2]` using `mw_localmalloc()`

The following code demonstrates how to use the local malloc to allocate a variable `A` co-located (on the same nodelet as) `B[2]` as shown in Figure 3.1:

```
long * A = (long *) mw_localmalloc(sizeof(long), &B[2]);
```

3.3.2 mw_malloc1dlong

```
long * mw_malloc1dlong(size_t numelements)
```

Return Value: Returns a long * to an array of numelements longs

Arguments:

- numelements: number of 64-bit elements to be allocated

Functionality: The `mw_malloc1dlong` allocates an array of `numelements` longs striped across the nodelets round robin. This array may wrap if `numelements` is larger than the number of nodelets. Arrays allocated using this function are accessed using array notation. Since each sequential element is on a different nodelet, accessing sequential elements of this array will cause a migration on each access. This distribution is best used for cases where the array is not access sequentially.

Notes: Defined in `distributed.h`. Works ONLY for 64-bit types (e.g. long).

Example:

Figure 3.2: Round robin array distribution for array of longs.

The following code demonstrates how to create and access an array `A` of size 10 longs with one element on each nodelet as shown in Figure 3.2:

```
#define N 10
long * A = mw_malloc1dlong(N);
for (long i=0; i<N; i++)
       A[i] = i;
```

This function ONLY works for 64-bit types. For types of other sizes, you must use `mw_malloc2d`.

3.3.3 mw_malloc2d

```
void ** mw_malloc2d(size_t numblocks, size_t blocksize)
```

Return Value: Returns a void ** to an array of `numblocks` elements, each of which is a pointer to a block of memory of size `blocksize`.

Arguments:

- numblocks: the number of blocks to be allocated
- blocksize: the size in bytes to be allocated for each block

Functionality: The `mw_malloc2d` function allocates a distributed array of numblocks pointers striped across nodelets round robin. Each points to a co-located memory block of blocksize.

Notes:

- Defined in `distributed.h`.
- Returns an array of pointers, so must be cast to `<type>` **
- Because it is a distributed array, accessing elements can cause migrations
- See `mw_arrayindex` for a more complete discussion of when migrations may occur and approaches to avoid them.

Example:

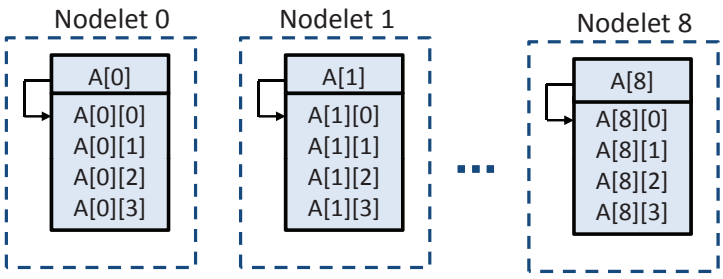


Figure 3.3: 2D array distribution on 8 nodelets using `mw_malloc2d()`

The following code demonstrates how to create and access a 2D array, `A`, of longs. This example has 32 elements with 4 consecutive elements on each of 8 nodelets as shown in Figure 3.3:

```
#define N 32
long epn = N/NODELETS();
long ** A = (long **) mw_malloc2d(NODELETS(), epn * sizeof(long));
for (long i=0; i<N; i++)
```

```
        A[i/epn][i%epn] = i;
```

Note that this loop will access the first 4 elements of the array locally, then migrate to the next nodelet to access that nodelet's elements, and so on.

3.3.4 mw_free

```
void mw_free(void *allocedpointer)
```

Return Value: None

Arguments:

- allocedpointer: pointer to allocated memory

Functionality: The `mw_free` frees data allocated by `mw_malloc1dlong`, `mw_malloc2d`, or `mw_mallocstripe`

Notes: Defined in `distributed.h`.

3.3.5 mw_localfree

```
void mw_localfree(void *allocedpointer)
```

Return Value: None

Arguments:

- allocedpointer: pointer to allocated memory

Functionality: The `mw_localfree` frees data allocated by mw_localmalloc

Notes: Defined in `distributed.h`.

3.3.6 mw_arrayindex

```
void * mw_arrayindex(void * array2d, unsigned long i,
   unsigned long numblocks, size_t blocksize)
```

Return Value: Returns a `void *` to the address of `array2d[i][0]`.

Arguments:

- array2d: array allocated with `mw_malloc2d`
- i: index for first dimension
- numblocks: numblocks parameter passed to `mw_malloc2d` when allocating `array2d`
- blocksize: blocksize parameter passed to `mw_malloc2d` when allocating `array2d`

Functionality: The `mw_arrayindex` takes as input a 2D array allocated with `mw_malloc2d` along with an index, `i`. It then calculates and returns the address of `array2d[i][0]`.

Notes: Defined in `distributed.h`.

This function is used for computing an address for `array2d[i][j]` without migrating to `array2d[i]` to load the pointer. It uses knowledge of how `mw_malloc2d` lays data out in memory to compute the address of `array2d[i][j]` directly. This function would be used to compute the address for a remote memory operation in order to avoid migration during the address computation. Note that this can be an expensive calculation and should be used at an outer level whenever possible, not as part of the inner loop computation.

Example: The following code demonstrates how to use `mw_arrayindex` to access a 2D array as shown in Figure 3.3:

```
#define N 32
long epn = N/NODELETS();
long** A = (long **) mw_malloc2d(NODELETS(), epn * sizeof(long));

// Migrating loop
for (long i=0; i<NODELETS(); i++)
    for (long j=0; j<epn; j++) {
        A[i][j] = 0;


// Loop using remote updates
for (long i=0; i<NODELETS(); i++) {
    long * Ai = (long *) mw_arrayindex(A, i, NODELETS(), epn * sizeof(long));
    for (long j=0; j<epn; j++)
        REMOTE_ADD((Ai+j), 1);
}
```

In the first loop, the thread will migrate to each element of `A[i][j]` in order to initialize it to 0. In the second loop, we want to use a remote update where the thread does not migrate. The compiler can calculate the location of `A[i]` directly based on the address in `A`. However, because `A[i]` is a pointer, it would migrate to `A[i]` and load that address in order to compute the location of `A[i][j]`. The `mw_arrayindex` function uses knowledge of the data layout to compute the start of the block pointed to by A[i], thus avoiding migrations in the computation of the location of A[i][j].

3.4 Replicated Data

These functions are used to allocate and access replicated data. When a variable is marked as `replicated`, a copy of the data element is allocated on each nodelet at the same address. The address of a replicated data element is configured such that the local instance (i.e. the instance located on the nodelet where the thread is executing) is always accessed. Therefore, specialized functions are needed to initialize and access specific instances of a replicated data structure. These functions require `memoryweb.h`.

3.4.1 replicated

```
replicated <var type> <var name> [= <intial value>];
```

Functionality: The `replicated` key word is used to indicate a global replicated variable. A copy of this variable is placed on each nodelet. A replicated variable always accesses the local copy on the nodelet where the thread is executing. Replicated variables are best used for data where it does not matter which instance of the variable is used, such as for constants or accumulating local data.

Notes: Defined in `repl.h`.

Note that when you update a replicated variable it updates only the LOCAL instance of that variable. Therefore different instances can have different values. In order to access a specific instance you must use specialized access functions.

Example:

```
struct stats {
    double max;
    double min;
};
replicated struct stats s;
replicated double PI = 3.14;
```

In this example, the stats struct, `s`, is replicated on each nodelet in the system. `PI` is also replicated on each nodelet in the system, with each copy initialized to 3.14. Whenever either `s` or `PI` is referenced in the program, the thread accesses the value on the local nodelet.

3.4.2 mw_mallocrepl

```
long * mw_mallocrepl(size_t blocksize)
```

Return Value: Returns a pointer to replicated block of size `blocksize`.

Arguments:

- blocksize: the size in bytes of the block to be allocated at each nodelet

Functionality: The `mw_mallocrepl` allocates a block of size `blocksize` on each nodelet. It returns a replicated pointer so that accesses using that pointer will always access the block on the local nodelet.

Notes: Defined in `memory.h`.
This is similar to using the `replicated` key word but is used when the size of the data structure is not known at compile time and therefore must be dynamically allocated.

Example:

The following code demonstrates how to dynamically allocate a replicated structure at each nodelet.

```
#include "memoryweb.h"
#include "stdio.h"
#include "stdlib.h"

typedef struct info {
    long nlet;
    long index;
} info;

replicated info *rinfo;

// Initialize the info struct on each nodelet
void fill_struct(void *ptr, long node)
{
    info *gto = (info *)ptr;
    gto->nlet = node;
    gto->index = NODE_ID();
}

// Allocated a replicated info struct on each nodelet
info *alloc_struct()
{
    info *mr = (info *)mw_mallocrepl(sizeof(info));
    mw_replicated_init_generic(mr, fill_struct);

    return mr;
}

int main(int argc, char *argv[])
{
    // Array with one element on each nodelet
    long *A = mw_malloc1dlong(NODELETS());

    info *tmp_info = alloc_struct();
    mw_replicated_init((long *)&rinfo, (long)tmp_info);

    // Print info from instance of info struct on each nodelet
    for (long i = 0; i < NODELETS(); i++) {
        info *local_info = mw_get_nth(rinfo, i);
        MIGRATE(&A[i]);
        printf("i=%d node=%d: LOCAL_INFO nlet %d index %d, RINFO nlet %d index %d\n",
               i, NODE_ID(), local_info->nlet, local_info->index, rinfo->nlet, rinfo->index);
    }

    return 0;
}
```

Figure 3.4: Simple Replicated Structure

3.4.3 mw_replicated_init

```
void mw_replicated_init(long * repl_addr, long value)
```

Return Value: None

Arguments:

- repl_addr: pointer to the replicated variable to be initialized
- value: initial value

Functionality: This function takes as input a pointer to a replicated variable and an initial value. It initializes each instance of the replicated variable to the initial value.

Notes: Defined in `repl.h`.

Example: The following code demonstrates how to initialize each instance of a replicated variable to the SAME value (3.14 in this case):

```
replicated long PI;
int main()
{
  mw_replicated_init(&PI, 3.14);
  ...
}
```

3.4.4 mw_replicated_init_multiple

```
void mw_replicated_init_multiple(long * repl_addr, long (*init_func)(long))
```

Return Value:

Arguments:

- repl_addr: pointer to the replicated variable to be initialized
- init_func: user-defined function that takes the nodelet id as input and returns the initial value for the replicated variable at that nodelet

Functionality: This function takes as input a pointer to a replicated variable and user-define function `init_func(long nid)`. It initializes the instance of the replicated variable on nodelet n to the value that is returned by the function when n is passed as the argument.

Notes: Defined in `repl.h`.

Example: The following code snippet demonstrates how to initialize each instance of a replicated variable to a different value. In this example, each instance of B is initialized to (nodelet ID * 5). So, for example, the instance on nodelet 0 is initialized to 0 and the instance on nodelet 5 is initialized to 25.

```
replicated long B;

long init_func(long nid) {
    return nid * 5;
}

int main()
{
    mw_replicated_init_multiple(&B, init_func);
    ...
}
```

3.4.5 mw_replicated_init_generic

```
void mw_replicated_init_generic(void * repl_addr, void (*init_func)(void *, long))
```

Return Value:

Arguments:

- repl_addr: pointer to the replicated variable to be initialized
- init_func: user-defined function that takes the absolute address of the replicated data structure on the nodelet with the ID given by the second argument.

Functionality: This function initializes a replicated data structure by calling `init_func` on each nodelet. The parameter `init_func` should take two arguments. The first is an absolute address of the replicated data structure on the nodelet with the ID given by the second argument.

Notes: Defined in `repl.h`.

Example: The following code snippet demonstrates how to initialize each instance of a replicated variable to a different value.

```
#include <memoryweb.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
replicated long foo;

replicated
struct info
{
    long x;
    long y;
} info;

void init_info(void *m, long node)
{
    struct info * i = (struct info*) m;
    i->x = node;
    i->y = 5*node + 4;
    return;
}

int main()
{
    long ** array = mw_malloc2d(NODELETS(), sizeof(long));
```

```
    mw_replicated_init_generic(&info, init_info);
    long i;
    for(i = 0; i != NODELETS(); i++)
    {
    MIGRATE(array[i]);
    printf("info.x = %ld and info.y = %ld on node %ld\n", info.x, info.y, i);
    }
    printf("info.x = %ld and info.y = %ld on node 5\n",
       ((struct info*)mw_get_localto(&info, array[5]))->x,
       ((struct info*)mw_get_localto(&info, array[5]))->y);
    return 0;
}
```

3.4.6 mw_get_nth

```
void * mw_get_nth(void * repl_addr, long n)
```

Return Value: Returns a pointer to the nth instance of the ptr data structure.

Arguments:

- repl_addr: pointer to the replicated data structure
- n: index indicating which instance of the data structure

Functionality: This function takes as input a pointer to a replicated data structure and returns a pointer to the nth instance of that data structure.

Notes: Defined in `repl.h`.

Example:

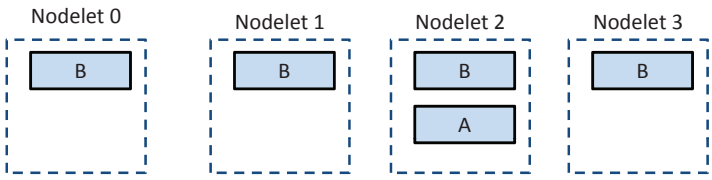


Figure 3.5: Data to illustrate use of `mw_get_nth` and `mw_get_localto`.

The following code demonstrates how to use the `mw_get_nth` function. It returns a pointer to the instance of the replicated variable `B` on nodelet 1 as shown in Figure 3.5.

```
replicated long B;
long * Bi = mw_get_nth(&B, 1);
```

3.4.7 mw_get_localto

```
void * mw_get_localto(void * repl_addr, void * localpointer)
```

Return Value: Returns a void * to the instance of the replicated variable located on the same nodelet as the address referenced by localpointer.

Arguments:

- repl_addr: a pointer to a replicated data structure.
- localpointer: pointer to a second data structure

Functionality: The function returns a pointer to the instance of the replicated data structure that is located on the same nodelet as the address referenced by localpointer.

Notes: Defined in `repl.h`.

Example: The following code demonstrates how to use the `mw_get_localto` function to access a specific instance of a replicated data structure as shown in Figure 3.5. It returns a pointer to the instance of B that is located on the same nodelet as A.

```
replicated long B;
...
long * localB = (long *) mw_get_localto(&B, &A);
```

Chapter 4

Architecture Specific Operations

Intrinsic functions are constructs that allow the programmer to directly access particular instructions provided by the machine. The following tables list the available intrinsic functions. The `memoryweb.h` file must be included to use them.

4.1 Atomic Operations

A full set of atomic operations on 64-bit data are provided as shown in Table 4.1. The atomics ensure that an operation completes without the possibility of another thread accessing the data element during the operation. These operations take as input a pointer to the memory location to be atomically updated and an additional argument. There are four "flavors" of atomic operations that specify what result is to be written to memory and what result is to be returned by the function, specified as follows:

- **No suffix**: the memory location is left unchanged and the result of the operation is returned by the function.
- **M**: the result of the operation is written to the memory location and returned.
- **S**: the second argument is written to the memory location and the result of the operation is returned.
- **MS**: the original value at the memory location is returned, and the result is written to the memory location.

For example, `ATOMIC_ADD(&A, 2)` loads the value of A from memory, adds 2, and returns the result without modifying memory. In contrast, `ATOMIC_ADDM(&A, 2)` loads the value of A from memory, adds 2, writes the result to A in memory, and returns the result. Executing `ATOMIC_ADDS(&A, 2)` loads the value of A from memory, adds 2, returns the result, and writes 2 to A in memory. Finally, `ATOMIC_ADDMS(&A, 2)` loads the value of A in from memory, adds 2, writes the result to A in memory and returns the original value of A loaded from memory.

IMPORTANT NOTE: the first argument MUST be a pointer to a 64-bit data type. The atomic instructions load and store 64-bit words, so calling the functions on a data-type smaller than 64-bits is not supported.

Table 4.1: Atomic Arithmetic Operation Intrinsics

long ATOMIC_ADD(long *, long)
long ATOMIC_AND(long *, long)
long ATOMIC_OR(long *, long)
long ATOMIC_XOR(long *, long)
long ATOMIC_MAX(long *, long)
long ATOMIC_MIN(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is returned. The location specified by the first argument is unchanged. The entire function is performed atomically.

long ATOMIC_ADDM(long *, long)
long ATOMIC_ANDM(long *, long)
long ATOMIC_ORM(long *, long)
long ATOMIC_XORM(long *, long)
long ATOMIC_MAXM(long *, long)
long ATOMIC_MINM(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is returned and also written to the location specified by the first argument. The entire function is performed atomically.

long ATOMIC_ADDS(long *, long)
long ATOMIC_ANDS(long *, long)
long ATOMIC_ORS(long *, long)
long ATOMIC_XORS(long *, long)
long ATOMIC_MAXS(long *, long)
long ATOMIC_MINS(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is returned and the value of the second argument is written to memory at the location specified by the first argument. The entire function is performed atomically.

long ATOMIC_ADDMS(long *, long)
long ATOMIC_ANDMS(long *, long)
long ATOMIC_ORMS(long *, long)
long ATOMIC_XORMS(long *, long)
long ATOMIC_MAXMS(long *, long)
long ATOMIC_MINMS(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is written to memory and the original value at the memory location addressed by the first argument is returned. The entire function is performed atomically.

4.2 Remote Updates

Remote updates are atomic updates to 64-bit data in memory that do NOT cause a thread migration. Instead, they send to the remote memory location a packet that contains data and the operation to be performed. Remote updates to the same address from the same thread are guaranteed to be performed in order. Remote updates to the same address from different threads or to different addresses from the same thread may occur in any order. Remote updates do NOT return a result; however they send an acknowledgement (ACK) that indicates that the update has been accepted by the memory front-end for processing (ACKs can be turned off). A thread is not allowed to migrate until all ACKs have returned. This ensures that (a) the ACK can "find" the thread and (b) that the thread cannot access the data before the remote update has completed. An explicit FENCE operation can also be used to block the thread until all ACKs have returned. The full set of remote update intrinsics are shown in Table 4.2.

IMPORTANT NOTE: the first argument MUST be a pointer to a 64-bit data type. The remote instructions load and store 64-bit words, so calling the functions on a data-type smaller than 64-bits is not supported.

Table 4.2: Remote Update Intrinsics

void REMOTE_ADD(long *, long)
void REMOTE_AND(long *, long)
void REMOTE_OR(long *, long)
void REMOTE_XOR(long *, long)
void REMOTE_MAX(long *, long)
void REMOTE_MIN(long *, long)
These intrinsic functions perform the selected operation between the operand addressed by the first argument and the second argument. The result is written to memory at the location specified by the first argument. A remote packet is generated to travel to the specified memory location and perform the function atomically.

void FENCE(void)
The thread performing the FENCE stalls until there are no outstanding acknowledges from previously executed remote memory operations. This instruction will otherwise act as a NOOP (No Operation) and makes no changes to the state of the thread.

4.3 Swaps

The architecture provides two types of atomic swap as detailed in Table 4.3.

IMPORTANT NOTE: the first argument MUST be a pointer to a 64-bit data type. The swap instructions load and store 64-bit words, so calling the functions with a data-type smaller than 64-bits is not supported.

Table 4.3: Atomic Swap Intrinsics

long ATOMIC_SWAP(long *, long)
This function writes the second argument to the location addressed by the first argument. The original value in memory is returned. The entire operation is performed atomically.

long ATOMIC_CAS(long *, long, long)
This function compares the third argument to the value addressed by the first argument. If equal, the second argument is written to memory. The original memory value is returned regardless of the result of comparison. The entire operation is performed atomically.

4.4 Specialized Operations

Two specialized operations are provided as shown in Table 4.4.

IMPORTANT NOTE: the second argument to `POPCNT` MUST be a pointer to a 64-bit data type. The swap instructions load and store 64-bit words so calling the functions with a data-type smaller than 64-bits is not supported.

Table 4.4: Specialized Intrinsic Operations

long POPCNT(long, long *)
Computes the popcount: The 64 bit word in the second argument is read and the number of 1 bits in the word counted. This value is added to the first argument and the result is returned.

unsigned long PRIORITY(unsigned long)
Computes the 6 bit priority encode on the argument, returning the result. The priority encode is the bit position of the highest number bit that is nonzero.

4.5 Thread Management

A set of intrinsics provide thread management capabilities as detailed in Table 4.5.

Table 4.5: Thread Management

void RESIZE()
This intrinsic resizes the thread to carry only the currently live registers. It is used by the compiler and programmer before a possible migration to reduce the thread size and thus the bandwidth requirements.

void RESCHEDULE()
This intrinsic places the thread at the end of the Run Queue to allow a new thread to be scheduled in the core. It can be used as a part of a busy wait loop (e.g. while waiting for a lock to become available). **This should not be used if you hold a lock.**

void ENABLE_ACKS()
This intrinsic will require all remotes issued by the thread to return an ACK.
void DISABLE_ACKS()
This intrinsic will turn off the ACK messages from remotes issued by the thread.

void DISABLE_MIGRATIONS()
This intrinsic will set the thread to not migrate; if the thread attempts to migrate in this state, an exception (except_code=6, except_cause=6 in the hardware) will occur. This may be a useful debugging aid.
void ENABLE_MIGRATIONS()
This intrinsic will allow the thread to resume migrations.

void DISABLE_INTERRUPTS()
This intrinsic will set the thread to not be interrupted by the timeslice interrupt (if enabled). It should generally not be needed in regular user code.
void ENABLE_INTERRUPTS()
This intrinsic will allow the thread to be interrupted by the timeslice interrupt. As with the corresponding `DISABLE_INTERRUPTS()`, this should not typically be needed in regular user code.

void ENTER_CRITICAL_SECTION()
This intrinsic combines `DISABLE_MIGRATIONS()` and `DISABLE_INTERRUPTS()` into a single call. This can be used upon acquiring a shared lock to ensure that the thread will release the lock before vacating its Gossamer core; this intrinsic may help prevent starvation.
void EXIT_CRITICAL_SECTION()
This intrinsic combines `ENABLE_MIGRATIONS()` and `ENABLE_INTERRUPTS()` into a single call. This can be used when preparing to release a shared lock.

When an application is launched, the default behavior is to have ACKs, migrations, and interrupts all enabled. Both the simulator and hardware have options for disabling ACKs when launching an application. As of v2.21 of this guide, timeslice interrupts are not enabled in the hardware.

4.6 System Information

Table 4.6 lists the set of intrinsics that are used to provide system specific information.

Table 4.6: System Query Intrinsics

unsigned long CLOCK(void)
Returns a 64-bit real time counter for the clock on the current node. See below for additional information on using this intrinsic for timing comparisons.

unsigned long MAXDEPTH(void)
Not yet implemented. Returns the current system setting for maximum spawn depth. Default is 255.

unsigned long THREAD_ID(void)
Returns a special value called a thread ID that can be used to uniquely identify a thread for debugging.

unsigned long NODE_ID(void)
Returns the *nodelet* ID of the nodelet on which it is executed.

unsigned long NODELETS(void)
Returns the total number of nodelets in the system.

unsigned long BYTES_PER_NODELET(void)
Returns the total number of bytes of memory for each nodelet.

The `CLOCK()` intrinsic returns a 64-bit real time counter that can be used for timing comparisons in the simulator and on the hardware. When using `CLOCK()` for timing comparisons, it is important to read the start and stop values from the same node since counters on different nodes may vary. Given start and stop counter values, the clock rate can be used to calculate elapsed time.

Section 7.3 has information on using the `starttiming()` function on the simulator, which is treated as a NOOP on the hardware. Alternatively, the `time.h` functions, such as `clock_gettime()`, are supported and use the SC clock. However, the time involved in doing a system call to the SC to get the time may be significant for small programs, so `CLOCK()` is more accurate. `CLOCK()` is the preferred mechanism for timing on the hardware.

It is recommended that the variables used to store the `CLOCK()` value are declared as `volatile`. Otherwise, the compiler may perform optimizations on those variable, and the results of such program may be invalid.

Chapter 5

Toolchain and Libraries

The Emu toolchain and libraries currently support program development in C/C++/Cilk. A beta release of the CilkPlus toolchain is also now available.

5.1 C Support

Emu supports the C language and has ported much of the musl C library. The toolchain currently supports the most common functionality from the following categories:

- stdio.h
- math.h
- string.h
- stdlib.h
- rand.h
- libgen.h
- stdarg.h
- search.h
- assert.h

The following are not supported and/or known to fail. In particular, we do not support much of the pthread functionality at this time:

- pthread.h
- wctype.h
- wchar.h
- stdio.h wide character conversions e.g. btowc
- sys.h: most of these are not yet tested
- spawn.h: not supported

- semphores.h: not supported
- unistd.h

Testing of the C library is an ongoing process and new features are added as needed. The most current status can be found in the "Libraries Status" spreadsheet on Box.com.

5.2 C++ Support

Emu supports C++ and has ported key functionality of the libcxx C++ library. The toolchain currently supports basic C++, such as classes and objects, inheritance, polymorphism, and templates, along with the most common functionality from the C++ standard library including:

- Containers: array, deque, forward_list, list, map, queue set, stack, unordered_map, unordered_set, vector
- General: algorithm, chrono, iterator, tuple
- Language support: limits, new, typeinfo
- Numerics: valarray, numeric, ratio, random
- Strings
- Streams: IO and file streams

Additional work is still needed in the following areas:

- C++ exception handling (try/catch/throw) is not yet supported. It is turned off via a compiler flag in the current toolchain.
- C++ containers (e.g. vector, list, etc.) are single threaded and allocated on a single nodelet.

5.3 GNU Multiple-precision Library

The GNU Multiple Precision Arithmetic (GMP) library is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. Emu has ported the signed integer arithmetic portion of the GMP library for the Emu architecture.

5.4 Emu C utilities library

5.4.1 Overview

Parallelism on Emu is essential to achieving good performance. Since cilk creates threads at the level of a function, implementing patterns such as recursive spawn trees or a two-level remote spawn require a lot of boilerplate code and intermediary functions. The `emu_c_utils` library implements most of these common patterns efficiently as library calls.

The programmer provides a pointer to a function that handles a single array slice, and the implementation spawns threads across the entire system to process each slice in parallel. Local variables can be made available to each worker function using the varargs calling convention.

The parallel apply functions accept a grain size argument. In order to avoid overwhelming the system with too many threads, it is usually best to spawn just enough threads to saturate the GC's on each nodelet. Assuming `GCS_PER_NODELET` is set correctly in your environment, `LOCAL_GRAIN()` will calculate a grain size from the array length to spawn exactly enough threads to saturate a single nodelet. `GLOBAL_GRAIN()` will do the same for the entire system. `GLOBAL_GRAIN_MIN()` and `LOCAL_GRAIN_MIN()` accept an additional grain size argument to avoid spawning too many threads for a small array.

Table 5.1: Overview of utility functions

Function	Functionality
Working with local arrays	
`emu_local_for()`	Applies a function to a range in parallel.
`emu_local_for_set_long()`	Initializes an array of `long`s to a single value in parallel.
`emu_local_for_copy_long()`	Copies an array of `long`s in parallel.
`emu_sort_local()`	Sorts a range in parallel with a custom comparator (like qsort).
`emu_memcpy()`	Copies a range of bytes (like memcpy).
`LOCAL_GRAIN_MIN()`	Chooses a grain size based on the array size.
Working with distributed (striped) arrays	
`emu_1d_array_apply()`	Implements a distributed parallel for over a `malloc1dlong()` array
`emu_1d_array_reduce_sum()`	Implements a distributed parallel reduction over a `malloc1dlong()` array
`GLOBAL_GRAIN_MIN()`	Chooses a grain size based on the array size.
Working with distributed (chunked) arrays	
`emu_chunked_array_replicated_new()`	Allocates and initializes an `emu_chunked_array`, returning a replicated pointer to the data structure.
`emu_chunked_array_replicated_free()`	Frees a pointer allocated with `emu_chunked_array_replicated_new`.
`emu_chunked_array_replicated_init()`	Initializes an `emu_chunked_array` struct.
`emu_chunked_array_replicated_deinit()`	Deallocates the array associated with a `emu_chunked_array` struct.
`emu_chunked_array_index()`	Returns a pointer to the `i`th element within the array.
`emu_chunked_array_size()`	Returns the number of elements in the array.
`emu_chunked_array_apply()`	Implements a distributed parallel for over an `emu_chunked_array`.
`emu_chunked_array_set_long()`	Initializes an array of longs stored within an `emu_chunked_array` to a single value in parallel.
`emu_chunked_array_reduce_sum()`	Implements a distributed parallel reduction over an `emu_chunked_array`.
`emu_chunked_array_from_local()`	Implements a scatter operation from a local array to an `emu_chunked_array`.
`emu_chunked_array_to_local()`	Implements a gather operation from an `emu_chunked_array` to a local array.
Timing hooks	
`hooks_region_begin()`	Marks the beginning of a region of interest.
`hooks_region_end()`	Marks the end of a region of interest.
`hooks_set_attr_u64()`	Records an unsigned integer value for the current region.
`hooks_set_attr_i64()`	Records a signed integer value for the current region.
`hooks_set_attr_f64()`	Records a double-precision floating point value for the current region.
`hooks_set_attr_str()`	Records a string value for the current region.
`hooks_set_active_region()`	Sets which region will be active during the current execution

5.4.2 Working with local arrays

These functions constitute an alternative to `cilk_for`, implemented without compiler support.

5.4.2.1 emu_local_for

```
void
emu_local_for(
    long begin,
    long end,
    long grain,
    void (*worker)(long begin, long end, va_list args),
    ...
);
```

Arguments:

- begin: beginning of the iteration space (usually 0)
- end: end of the iteration space (usually array length)
- grain: Minimum number of elements to assign to each thread
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function should go from `begin` to `end` with a stride of 1.
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Applies a function to a range in parallel. These loops can be replaced with `cilk_for` once it is working.

Notes:

- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
long * x = malloc(n * sizeof(long));

void worker(long begin, long end, va_list args)
{
    long * x = va_arg(args, long*);
    long b = va_arg(args, long);
```

```
    for (long i = begin; i < end; ++i) {
        x[i] += b;
    }
}
emu_local_for(0, n, LOCAL_GRAIN_MIN(n, 64), worker, x, b);
```

5.4.2.2 emu_local_for_set_long

```
void
emu_local_for_set_long(long * array, long n, long value);
```

Arguments:

- dst: Pointer to array
- n: Array length
- value: Value to set

Functionality: Sets each value of `array` to `value` in parallel.

Notes:

- Initializing striped arrays with this function will be inefficient, use `emu_1d_array_apply` for that instead.

Example:

```
long n = 1024;
long * a = malloc(n * sizeof(long));
// Initialize elements of a to zero
emu_local_for_set_long(a, n, 0);
```

5.4.2.3 emu_local_for_copy_long

```
void
emu_local_for_copy_long(long * dst, long * src, long n);
```

Arguments:

- dst: Pointer to destination array
- src: Pointer to source array
- n: Array length

Functionality: Copies `src` to `dst` in parallel.

Notes:

- If `src` and `dst` are on different nodelets, then it is most efficient to call this function on the source nodelet, so that remote writes can be used to send the data.
- Copying striped arrays with this function will be inefficient, use `emu_1d_array_apply` for that instead.

Example:

```
long n = 1024;
long * a = malloc(n * sizeof(long));
long * b = malloc(n * sizeof(long));
// Copy a to b
emu_local_for_copy_long(b, a, n);
```

5.4.2.4 emu_memcpy

```
void *
emu_memcpy(void * dst, void * src, size_t n)
```

Arguments:

- dst: Pointer to destination array
- src: Pointer to source array
- n: Number of bytes to copy

Functionality: The parallel equivalent of `memcpy`.

Notes:

Example:

```
size_t n = 10000;

// Allocate two buffers
long * array1 = malloc(sizeof(long) * n); assert(array1);
// Give the second array a weird offset
long * array2 = malloc(sizeof(long) * n); assert(array2);

// Initialize to known values
for (long i = 0; i < n; ++i) {
    array1[i] = i;
    array2[i] = 0;
}

// Do the copy in parallel
emu_memcpy(array2, array1, n * sizeof(long));

// Check
for (long i = 0; i < n; ++i) {
    TEST_ASSERT_EQUAL(i, array2[i]);
}

// Clean up
free(array1);
free(array2);
```

5.4.2.5 LOCAL_GRAIN_MIN

```
static inline long
LOCAL_GRAIN_MIN(long n, long min_grain);
```

Return Value: Returns a grain size suitable for a loop processing `n` items on a single nodelet.

Arguments:

- n: Number of elements in the array.
- min_grain: Minimum number of elements per thread.

Functionality: Emu can't handle a very large number of threads, as might occur when using a small grain size and a large array size. This function caps the total number of threads by increasing the grain size.

Notes: Software cannot detect the number of GC's per nodelet. Set the environment variable `GCS_PER_NODELET` if running on the simulator with more than one GC per nodelet.

Example:

5.4.3 Working with distributed (striped) arrays

As discussed in Section 3.3.2, an array allocated with `mw_malloc1dlong` will be striped across nodelets in a round-robin fashion. The most efficient way to access this type of array is with a stride of `NODELETS()`, such that each thread remains on a single nodelet. These functions implement a two-level spawn tree and split the array into striped slices for each worker function.

5.4.3.1 emu_1d_array_apply

```
void
emu_1d_array_apply(
    long * array,
    long size,
    long grain,
    void (*worker)(long * array, long begin, long end, va_list args),
    ...
);
```

Return Value:

Arguments:

- array: Pointer to striped array allocated with `malloc1dlong()`.
- size: Length of the array (number of elements)
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function should go from `begin` to `end` and have a stride of `NODELETS()`. Each worker function will be assigned elements on a single nodelet.
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel for over a `malloc1dlong()` array.

Notes:

- Each worker function will assigned an iteration space on a single nodelet. Don't forget to use striped indexing (see example)!
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
long * x = malloc1dlong(n);

void worker(long * array, long begin, long end, va_list args)
{
    long * x = array;
    long b = va_arg(args, long);
    for (long i = begin; i < end; i += NODELETS()) {
        x[i] += b;
    }
}
emu_1d_array_apply(x, n, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.4.3.2 emu_1d_array_reduce_sum

```
long
emu_1d_array_reduce_sum(
    long * array,
    long size,
    long grain,
    void (*worker)(long * array, long begin, long end, long * sum, va_list args),
    ...
);
```

Return Value: Returns the sum of all the elements in the array.

Arguments:

- array: Pointer to striped array allocated with `malloc1dlong()`.
- size: Length of the array (number of elements)
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function should go from `begin` to `end` and have a stride of `NODELETS()`. Each worker function will be assigned elements on a single nodelet.
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel reduce over a `malloc1dlong()` array.

Notes:

- Each worker function will assigned an iteration space on a single nodelet. Don't forget to use striped indexing (see example)!
- The `sum` argument to the worker function points to a local accumulation variable that is shared by all the threads on a given nodelet. An atomic function such as `REMOTE_ADD` should be used to update this sum. Additionally, it may be more efficient for each thread to calculate a local sum before adding to the nodelet-local sum. See the example for more details.
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
```

```
long * x = malloc1dlong(n);

void worker(long * array, long begin, long end, long * sum, va_list args)
{
    long * x = array;
    long b = va_arg(args, long);
    long local_sum = 0;
    for (long i = begin; i < end; i += NODELETS()) {
        local_sum += x[i] * b;
    }
    REMOTE_ADD(sum, local_sum);
}
long sum = emu_1d_array_reduce_sum(x, n, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.4.3.3 GLOBAL_GRAIN_MIN

```
static inline long
GLOBAL_GRAIN_MIN(long n, long min_grain);
```

Return Value: Returns a grain size suitable for a loop processing `n` items on the entire system.

Arguments:

- n: Number of elements in the array.
- min_grain: Minimum number of elements per thread.

Functionality: Emu can't handle a very large number of threads, as might occur when using a small grain size and a large array size. This function caps the total number of threads by increasing the grain size.

Notes: Software cannot detect the number of GC's per nodelet. Set the environment variable `GCS_PER_NODELET` if running on the simulator with more than one GC per nodelet.

Example:

5.4.4 Working with distributed (chunked) arrays

The term "chunked" (or "blocked") refers to a particular strategy for using `mw_malloc2d()`. The first dimension of the 2D array now holds pointers to a chunk of memory on each nodelet instead of a pointers to each element. Unlike a striped array, elements with consecutive indices have spatial locality.

```
typedef struct point { long x; long y;} point;
long n = 1024;
point * last_point;
point ** striped_array = mw_malloc2d(n, sizeof(point));
last_point = striped_array[n - 1];
point ** chunked_array = mw_malloc2d(NODELETS(), n/NODELETS() * sizeof(point));
last_point = &chunked_array[NODELETS() - 1][n/NODELETS() - 1];
```

Working with chunked arrays introduces several complications:

- Converting logical array indexes into 2D index operations (i.e. `array[i / N][i % N]`)
- Keeping track of the block size `N` and per-element size to use in each indexing operation.
- Using `mw_arrayindex` for indexing operations to avoid migrating where possible.
- Initializing and replicating the array metadata to all nodelets.

The struct `emu_chunked_array` encapsulates this logic within an abstract data type. It holds the array metadata in replicated storage, and functions are provided to calculate indices and apply functions to the array in parallel.

```
typedef struct point { long x; long y; } point;
long n = 1024;
emu_chunked_array *point_array = emu_chunked_array_replicated_new(n, sizeof(point));
point * last_point = emu_chunked_array_index(point_array, n-1);
```

5.4.4.1 emu_chunked_array_replicated_new

```
emu_chunked_array *
emu_chunked_array_replicated_new(
    long num_elements,
    long element_size
);
```

Return Value: A relative pointer to a replicated `emu_chunked_array`.

Arguments:

- num_elements: Number of elements in the array
- element_size `sizeof()` each array element

Functionality: Allocates replicated storage for an `emu_chunked_array`, initializes the storage on each nodelet, and returns a relative pointer to the data structure. There will be `num_elements` elements with `element_size` bytes each.

Notes:

- Use `emu_chunked_array_replicated_free()` to free the array.

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
emu_chunked_array_replicated_free(x);
```

5.4.4.2 emu_chunked_array_replicated_free

```
void emu_chunked_array_replicated_free(emu_chunked_array * self);
```

Return Value:

Arguments:

- self: Pointer initialized with `emu_chunked_array_replicated_new`

Functionality: Frees a pointer allocated with `emu_chunked_array_replicated_new`.

Notes:

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
emu_chunked_array_replicated_free(x);
```

5.4.4.3 emu_chunked_array_replicated_init

```
void
emu_chunked_array_replicated_init(
    emu_chunked_array * self,
    long num_elements,
    long element_size
);
```

Return Value:

Arguments:

- self: Pointer to uninitialized struct, which MUST be located in replicated storage.
- num_elements: Number of elements in the array
- element_size: `sizeof()` each array element

Functionality: Initializes an `emu_chunked_array`. There will be `num_elements` elements with `element_size` bytes each. Array metadata will be replicated onto each nodelet.

Notes:

- The first argument MUST point to replicated storage, i.e. a `replicated` global variable.
- Use `emu_chunked_array_replicated_deinit()` to free the array.
- `emu_chunked_array_replicated_new` is the dynamic-allocation equivalent of this function, and is generally more convenient since it handles the replicated allocation for you.

Example:

```
long n = 1024;
replicated emu_chunked_array x;
emu_chunked_array_replicated_init(&x, n, sizeof(long));
emu_chunked_array_replicated_deinit(&x);
```

5.4.4.4 emu_chunked_array_replicated_deinit

```
void emu_chunked_array_replicated_deinit(emu_chunked_array * self);
```

Return Value:

Arguments:

- self: Pointer to struct initialized with `emu_chunked_array_replicated_init`

Functionality: Deallocates the array associated with a `emu_chunked_array` struct.

Notes:

- Use only in conjunction with `emu_chunked_array_replicated_init`. If you created the array with `emu_chunked_array_replicated_new`, use `emu_chunked_array_replicated_free` instead.

Example:

```
long n = 1024;
replicated emu_chunked_array x;
emu_chunked_array_replicated_init(&x, n, sizeof(long));
emu_chunked_array_replicated_deinit(&x);
```

5.4.4.5 emu_chunked_array_index

```
static inline void *
emu_chunked_array_index(emu_chunked_array * self, long i);
```

Return Value:

Arguments:

- self: Pointer to `emu_chunked_array`
- i: Index of requested element

Functionality: Returns a pointer to an element within the array.

Notes:

- Remember to cast the returned value to the appropriate type before dereferencing. It may be helpful to define a helper macro to make the syntax less verbose (see example).

Example:

```
long n = 1024;
emu_chunked_array * array = emu_chunked_array_replicated_new(n, sizeof(long));
#define X(I) *(long*)emu_chunked_array_index(array, I)
X(10) = X(20) + X(30);
```

5.4.4.6 emu_chunked_array_size

```
void
emu_chunked_array_size(emu_chunked_array * array);
```

Return Value: Returns the number of elements in the array (same as `num_elements` argument that was passed during initialization).

Arguments:

- array: Pointer to initialized `emu_chunked_array`

Functionality: Returns the number of elements in the array (same as `num_elements` argument that was passed during initialization).

Notes:

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
assert(emu_chunked_array_size(x) == n);
```

5.4.4.7 emu_chunked_array_apply

```
void
emu_chunked_array_apply(
    emu_chunked_array * array,
    long grain,
    void (*worker)(emu_chunked_array * array, long begin, long end, va_list args),
    ...
);
```

Return Value:

Arguments:

- array: Pointer to `emu_chunked_array`
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function is responsible for array elements from `begin` to `end` with a stride of 1. Because each worker function will be assigned elements on a single nodelet, it is more efficient to call `emu_chunked_array_index` once before the loop, and do linear indexing from that pointer (see example).
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel for over an `emu_chunked_array`.

Notes:

- Each worker function will assigned an iteration space on a single nodelet. After indexing, you may treat the local slice like a contiguous array (see example).
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));

void
worker(emu_chunked_array * array, long begin, long end, va_list args)
{
    long b = va_arg(args, long);
```

```
    long * x = emu_chunked_array_index(array, begin);

    for (long i = 0; i < end-begin; ++i) {
        x[i] += b;
    }
}
emu_chunked_array_apply(x, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.4.4.8 emu_chunked_array_set_long

```
void
emu_chunked_array_set_long(emu_chunked_array * array, long value);
```

Return Value:

Arguments:

- array: `emu_chunked_array` to initialize
- value: Set each element to this value

Functionality: Initializes each element of the array to a single value, in parallel.

Notes:

- The array must have been initialized to store a long datatype.

Example:

```
long n = 1024;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));
emu_chunked_array_set_long(x, 0);
emu_chunked_array_replicated_free(x);
```

5.4.4.9 emu_chunked_array_reduce_sum

```
long
emu_chunked_array_reduce_sum(
    emu_chunked_array * array,
    long grain,
    void (*worker)(emu_chunked_array * array, long begin, long end,
          long * partial_sum, va_list args),
    ...
);
```

Return Value: Returns the sum of all the elements in the array.

Arguments:

- array: Pointer to `emu_chunked_array`
- grain: Minimum number of elements to assign to each thread.
- worker: worker function that will be called on each array slice in parallel. The loop within the worker function is responsible for array elements from `begin` to `end` with a stride of 1. Because each worker function will be assigned elements on a single nodelet, it is more efficient to call `emu_chunked_array_index` once before the loop, and do linear indexing from that pointer (see example).
- ...: Additional arguments to pass to each invocation of the worker function. Arguments will be passed via the varargs interface, and you will need to cast back to the appropriate type within the worker function using the `va_arg` macro.

Functionality: Implements a distributed parallel reduction over an `emu_chunked_array` of `long` type.

Notes:

- This function only makes sense if the `emu_chunked_array` was initialized to store elements of type `long`.
- Each worker function will assigned an iteration space on a single nodelet. After indexing, you may treat the local slice like a contiguous array (see example).
- The `partial_sum` argument to the worker function points to a local accumulation variable that is shared by all the threads on a given nodelet. An atomic function such as `REMOTE_ADD` should be used to update this sum. Additionally, it may be more efficient for each thread to calculate a local sum before adding to the nodelet-local sum. See the example for more details.
- There is an alternate version of this function, suffixed with `_var`, that accepts a `va_list` instead of varargs.

Example:

```
long n = 1024;
long b = 5;
emu_chunked_array * x = emu_chunked_array_replicated_new(n, sizeof(long));

void
worker(emu_chunked_array * array, long begin, long end,
       long * partial_sum, va_list args)
{
    long b = va_arg(args, long);
    long * x = emu_chunked_array_index(array, begin);
    long local_sum;
    for (long i = 0; i < end-begin; ++i) {
        local_sum += x[i] * b;
    }
    REMOTE_ADD(partial_sum, local_sum);
}
long sum = emu_chunked_array_reduce_sum(x, GLOBAL_GRAIN_MIN(n, 64), worker, b);
```

5.4.4.10 emu_chunked_array_from_local

```
void
emu_chunked_array_from_local(emu_chunked_array * self, void * local_array);
```

Return Value:

Arguments:

- self: Pointer to `emu_chunked_array`
- local_array: Pointer to local array

Functionality: Copies elements from a local array to an emu_chunked_array in parallel.

Notes:

- The chunked array must have already been initialized with the same number of elements as the local array, and the same element size.

Example:

```
long n = 1024;

// Allocate a local and a chunked array
long * x = malloc(n * sizeof(long));
emu_chunked_array * y = emu_chunked_array_replicated_new(n, sizeof(long));

// Assign index to each value of local array
for (long i = 0; i < n; ++i) {
    x[i] = i;
}
// Set chunked array to invalid values
emu_chunked_array_set_long(y, -1);

// Do the scatter
emu_chunked_array_from_local(y, x);

// Check
for (long i = 0; i < n; ++i) {
    long val = *(long*)emu_chunked_array_index(y, i);
    TEST_ASSERT_EQUAL(i, val);
}

// Clean up
free(x);
emu_chunked_array_replicated_free(y);
```

5.4.4.11 emu_chunked_array_to_local

```
void
emu_chunked_array_to_local(emu_chunked_array * self, void * local_array);
```

Return Value:

Arguments:

- self: Pointer to `emu_chunked_array`
- local_array: Pointer to local array

Functionality: Copies elements from an emu_chunked_array to a local array in parallel.

Notes:

- The local array must have already been allocated with the same number of elements as the chunked array, and the same element size.

Example:

```
long n = 1024;

// Allocate a local and a chunked array
long * x = malloc(n * sizeof(long));
emu_chunked_array * y = emu_chunked_array_replicated_new(n, sizeof(long));

// Assign index to each value of chunked array
emu_chunked_array_apply(y, GLOBAL_GRAIN_MIN(n, 64), assign_index_worker);

// Set local array to invalid
for (long i = 0; i < n; ++i) {
    x[i] = -1;
}

// Do the gather
emu_chunked_array_to_local(y, x);

// Check
for (long i = 0; i < n; ++i) {
    TEST_ASSERT_EQUAL(i, x[i]);
}

// Clean up
free(x);
emu_chunked_array_replicated_free(y);
```

5.4.5 Timing Hooks

These functions implement a timer subsystem suitable for performance analysis, along with tools to annotate the output and control the level of detail in the simulator.

5.4.5.1 hooks_region_begin

```
void
hooks_region_begin(const char* name);
```

Return Value:

Arguments:

- name: Name of the region

Functionality: Marks the beginning of a region of interest, and takes the following actions:

- Calls `starttiming()` to trigger detailed timing mode when running within the simulator.
- Starts a timer by calling the `CLOCK()` macro on nodelet 0.

Notes:

- Each call to `hooks_region_begin()` should be matched with a call to `hooks_region_end()`. Regions may not be nested, however there may be many regions within the same program, and the same region name can be reused multiple times.
- Since software cannot currently detect the clock frequency, timing accuracy depends on the environment variable `CORE_CLK_MHZ` being set correctly. The number of ticks elapsed, however, will be correct regardless.
- The timing output can be directed to an arbitrary file by setting the environment variable `HOOKS_FILENAME`.

Example:

```
// Initialization section, will not be timed
long n = 1024;
long * x = malloc1dlong(n);
for (long i = 0; i < n; ++i) { x[i] = i; }
// Region of interest, will be timed
hooks_region_begin("sum");
long sum = 0;
for (long i = 0; i < n; ++i) { sum += x[i]; }
double time_ms = hooks_region_end();
// Output: {"region_name": "sum", "time_ms": 3.14, "ticks": 1234567}
```

5.4.5.2 hooks_region_end

```
double
hooks_region_end(const char* name);
```

Return Value: The time elapsed since the call to `hooks_region_begin`, in milliseconds.

Arguments:

- name: Name of the region

Functionality: Marks the beginning of a region of interest, and takes the following actions:

- Stops the timer by calling the `CLOCK()` macro on the same nodelet it was called on initially.
- Outputs information about this region in JSON format, including the region name, the time elapsed in milliseconds, the number of ticks elapsed, and any attributes that were set within the region with `hooks_set_attr` functions.

Notes:

- Each call to `hooks_region_begin()` should be matched with a call to `hooks_region_end()`. Regions may not be nested, however there may be many regions within the same program, and the same region name can be reused multiple times.
- Since software cannot currently detect the clock frequency, timing accuracy depends on the environment variable `CORE_CLK_MHZ` being set correctly. The number of ticks elapsed, however, is independent of clock rate.
- The timing output can be directed to an arbitrary file by setting the environment variable `HOOKS_FILENAME`.

Example:

```
// Initialization section, will not be timed
long n = 1024;
long * x = malloc1dlong(n);
for (long i = 0; i < n; ++i) { x[i] = i; }
// Region of interest, will be timed
hooks_region_begin("sum");
long sum = 0;
for (long i = 0; i < n; ++i) { sum += x[i]; }
double time_ms = hooks_region_end();
// Output: {"region_name": "sum", "time_ms": 3.14, "ticks": 1234567}
```

5.4.5.3 hooks_set_active_region

```
void
hooks_set_active_region(const char* name);
```

Return Value:

Arguments:

- name: Name of the region

Functionality: Sets which region name will trigger `starttiming()`. All "inactive" regions will not call starttiming. If this function is never called, all regions will be active. Only the simulator pays attention to which regions are active/inactive, the hardware executes all regions normally.

Notes:

- Until `stoptiming()` is implemented in the simulator, it is impossible to switch back to functional simulation for regions after the active region.
- You may wish to set the active region directly from an environment variable at the start of your program like this:

  ```
  // Set active region from environment variable
  hooks_set_active_region(getenv("HOOKS_ACTIVE_REGION"));
  ```

Example:

```
// Set active region
hooks_set_active_region("sum");

// Initialization section, will be simulated in functional mode
hooks_region_begin("init");
long n = 1024;
long * x = malloc1dlong(n);
for (long i = 0; i < n; ++i) { x[i] = i; }
hooks_region_end();

// Region of interest, will be simulated in timing mode
hooks_region_begin("sum");
long sum = 0;
for (long i = 0; i < n; ++i) { sum += x[i]; }
hooks_region_end();
```

5.4.5.4 hooks_set_attr

```
void hooks_set_attr_u64(const char * key, uint64_t value);
void hooks_set_attr_i64(const char * key, int64_t value);
void hooks_set_attr_f64(const char * key, double value);
void hooks_set_attr_str(const char * key, const char* value);
```

Return Value:

Arguments:

- key: Name to be associated with the value in the output
- value: Value to print after the end of this region

Functionality: Adds a custom key/value pair to the JSON output printed after a region ends. This can be used to annotate each piece of timing data with additional information, such as the array size, version number, or other configuration parameters.

Notes: Attributes must be set before a region or within a region; `hooks_region_end()` resets all attributes.

Example:

```
long n = 1024;
long * x = malloc1dlong(n);
for (long i = 0; i < n; ++i) { x[i] = i; }
hooks_set_attr_i64("N", n);
hooks_set_attr_str("version", "2.0rc3");
hooks_region_begin("sum");
long sum = 0;
for (long i = 0; i < n; ++i) { sum += x[i]; }
hooks_region_end();
// Output: {"region_name": "sum", "time_ms": 3.14, "ticks": 1234567, "N": 1024,
// "version": "2.0rc3"}
```

5.4.6 Building

The following examples assume that the toolchain is installed in the default location, `/usr/local/emu`.

5.4.6.1 Emu Toolchain

The emu_c_utils library was incorporated into the Emu toolchain as of version 18.08. The following items will need to be passed to `emu-cc` to be able to utilize the library on the Emu architecture:

```
-lemu_c_utils
```

If using CMake, the following line can be added to CMakeLists.txt:

```
link_libraries(emu_c_utils)
```

5.4.6.2 x86 toolchain

The 18.08 release additionally provides x86 libraries for memoryweb and emu_c_utils. Emu applications can be cross-compiled for x86, allowing developers to rapidly build and test code using familiar IDE's and debuggers before running on `emusim` or on Emu hardware. When running an Emu application targeting the x86 architecture, the application treats the system as having a single nodelet.

In order to cross-compile for Emu a compiler with Cilk support, such as GCC 5.5, is required.

The include and library paths for the x86 implementation are as follows:

- include: `/usr/local/emu/x86/include/`
- library: `/usr/local/emu/x86/lib/`

Add the following lines to your Makefile to link with the x86 libraries for memoryweb and emu_c_utils:

```
CFLAGS+=-I/usr/local/emu/x86/include
LDFLAGS+=-lemu_c_utils -L/usr/local/emu/x86/lib
```

Or, with CMake:

```
include_directories(/usr/local/emu/x86/include)
link_libraries(/usr/local/emu/x86/lib/libemu_c_utils.a)
```

GCC will additionally require `-lcilkrts` and `-fcilkplus`.

5.5 Emu multinode I/O library functions

5.5.1 Overview

Due to the current state of the toolchain and user-space driver, multinode I/O on Emu requires fine tuning. The goal of this library is to abstract I/O on Emu such that no matter what nodelet

the caller resides on, the I/O will succeed. This means that unless the directory(s) the application is opening files in is cross-mounted across all node-boards, the file, if created, will reside on the node-board where `mw_fopen` was successfully invoked and if the file is not created, it must already reside on the given node-board. Once `mw_fopen` succeeds on node-board X, all calls to `mw_fread, mw_fwrite and mw_fclose` will also occur on node-board X.

IMPORTANT: The recommended and most efficient use of `mw_fread and mw_fwrite` is to ensure that the first argument `ptr`, points to a non-replicated memory location that's **local** to node-board X. A great deal of care should be taken prior to invoking `mw_fread and mw_fwrite` when `ptr` points to replicated memory.

5.5.1.1 mw_fopen

```
/**
 * Calls libc's fopen and ensures that the returned pointer is resides on the
 * same nodelet as local_ptr.
 *
 * @param path The path to the file.
 * @param mode The mode used for opening the file.
 * @param local_ptr A pointer to a local nodelet address where I/O will be
 * performed.
 * @return A local pointer to the file handle upon success, NULL with errno
 * set appropriately otherwise.
 */
FILE *mw_fopen(const char *path, const char *mode, void *local_ptr);
```

Functionality: Ensures that the file pointer returned is local to the nodelet that the caller specifies via the `local_ptr` argument.

Example:

```
FILE *fp = mw_fopen("testfile", "w+");
if (!fp)
    exit(EXIT_FAILURE);
```

5.5.1.2 mw_fclose

```
/**
 * Calls libc's fclose and ensures I/O happens on the nodelet where fp resides.
 *
 * @param fp A local pointer to the file handle.
 *
 * @return 0 upon success, otherwise EOF with errno set appropriately.
 */
int mw_fclose(FILE *fp);
```

Functionality: Ensures that the resources associated with the given file pointer, `fp`, are released.

Example:

```
int ret = mw_fclose(fp);
if (!ret)
    exit(EXIT_FAILURE);
```

5.5.1.3 mw_fread

```
/**
 * Calls libc's fread and ensures I/O happens on the nodelet where fp resides.
 *
 * @param ptr The pointer to the buffer that should be read into.
 * @param size The size of a given member in the buffer pointed to by ptr.
 * @param nmemb The number of members in the buffer pointed to by ptr.
 * @param fp The local file handle to read from.
 *
 * @return The number of items read upon success, otherwise a short item
 * count (or zero).
 *
 * @note Ensure ptr is a valid view1 pointer local to "this" node for best
 * performance.
 */
int mw_fread(void *ptr, size_t size, size_t nmemb, FILE *fp);
```

Functionality: Identifies whether `ptr` is not replicated and local to "this" node; if it is, `fread(3)` is invoked; otherwise, temporary space is allocated on "this" node and `fread(3)` is invoked, then the `memcpy(3)` is used to copy the contents of the temporary space to `ptr`.

Example:

```
char *buf = mw_localmalloc(1024, local_ptr);
int nread = mw_fread(buf, sizeof(char), 1024, fp);
if (nread != expected_nread)
    exit(EXIT_FAILURE)
```

5.5.1.4 mw_fwrite

```
/**
 * Calls libc's fwrite and ensures I/O happens on the same nodelet where fp
 * resides.
 *
 * @param ptr The pointer to the buffer that should be read from.
 * @param size The size of a given member in the buffer pointed to by ptr.
 * @param nmemb The number of members in the buffer pointed to by ptr.
 * @param fp The local file handle to written to.
 *
 * @return The number of items written upon success, otherwise a short item
 * count (or zero).
 *
 * @note Ensure ptr is a valid view1 pointer local to "this" node for best
 * performance.
 */
```

Functionality: Identifies whether `ptr` is not replicated and local to "this" node; if it is, `fwrite(3)` is invoked; otherwise, temporary space is allocated on "this" node, the data at `ptr` is copied to the temporary space, and `fwrite(3)` is invoked.

Example:

```
int nwrote = mw_fwrite(buf, sizeof(char), 1024, fp);
if (nwrote != expected_nwrote)
    exit(EXIT_FAILURE)
```

5.6 CilkPlus

A beta release of the CilkPlus toolchain was made available in conjunction with v2.22 of this docuement (the emu-18.11-cplus toolchain). As of v2.26 of this document, the CilkPlus toolchain is now the mainline toolchain and support for the non-CilkPlus toolchains has been ended. The CilkPlus toolchain provides improved C++ support (especially C++11 and C++14), fixes a bug in cilk_for that prevented its use on the hardware, and includes an initial implementation of a subset of CilkPlus reducers.

5.7 CilkPlus Reducers

A CilkPlus Reducer is an object that allow several threads to efficiently combine partial results into a single value while avoiding data races and preserving serial semantics. For a basic overview of the concepts behind CilkPlus Reducers, see Intel's documentation at https://www.cilkplus.org/tutorial-cilk-plus-reducers.

The Emu CilkPlus toolchain currently only provides a subset of all possible reducers, implemented using remote atomics for 64-bit integers. One view is allocated on each nodelet in replicated memory, so that threads never need to migrate in order to reduce with a view.

A `PerNodeletReducer` is instantiated with a single template argument: the Monoid which encapsulates the the view type with corresponding identity and reduce operators. The following Monoid implementations are provided:

```
cilk::op_add<long> // Signed addition
cilk::op_and<long> // Bitwise AND
cilk::op_or<long>  // Bitwise OR
cilk::op_xor<long> // Bitwise XOR
cilk::op_min<long> // Minimum
cilk::op_max<long> // Maximum
```

Example:

```
#include <cilk/cilk.h>
#include <cilk/reducer.h>
#include <vector>

int main(int argc, char* argv[])
{
    // Allocate local array, initializing each element to 1
    long n = 64;
    std::vector<long> data(n, 1);
    // Create a reducer
    // Using dynamic allocation here to force replicated storage
    auto result = new cilk::PerNodeletReducer<cilk::op_add<long>>(0);
    // Convert pointer to reference for convenience
    auto&r = *result;
    cilk_for (long i = 0; i < n; ++i) {
        (*r) += data[i];
    }
    // Expected result is n
    return r.get_value();
}
```

Notes:

1. Reducers are only available when compiling a C++ file with the CilkPlus toolchain.
2. A PerNodeletReducer MUST be allocated on the heap using `new`, stack allocations will lead to incorrect behavior. Example:

```
// GOOD
auto good_reducer = new cilk::PerNodeletReducer<cilk::op_add<long>>(0);
// BAD
cilk::PerNodeletReducer<cilk::op_add<long>> bad_reducer(0);
```

3. Reducers are compatible with any parallel construct, not only `cilk_for`. This includes custom uses of `cilk_spawn` and the parallel functions provided by `emu_c_utils`.

5.8 Ongoing Efforts

Work continues to extend and optimize the compilation toolchain and libraries. Key efforts include:

- Optimize performance of thread spawning: The CilkPlus runtime implementation of Emu threads is not yet fully optimized and may reduce performance for some applications. Efforts are underway to optimize the implementations of cilk_spawn and cilk_for.
- Extend CilkPlus reducers: Only a subset of the CilkPlus reducers are implemented at this time. Work continues to extend the available types of reducers.
- Extend C++ support: Efforts will continue to extend the C++ support as well as to provide a set of distributed C++ containers to improve programmer productivity.

Chapter 6

Example Programs

This chapter provides a set of simple example programs to illustrate the creation and control of threads, the distribution of data structures, and architecture specific operations. These examples are also used to demonstrate some of the debugging features and statistics. Most of these example codes can also be found in the Examples folder distributed with the simulator and toolchain.

6.1 Fibonacci

The `cilk_fib.c` program shown in Figure 6.1 is the canonical Cilk example used to demonstrate how to create and control threads using cilk_spawn and cilk_sync in a program. The first action in `main` on line 24 is to call the sequential function `fib_seq()`, a simple recursive function that calulates Fibonacci numbers. Line 25 calls the cilk function `fib_cilk()`.

In `fib_cilk()`, cilk_spawn is used to spawn a thread for each recursive call, allowing the child threads to execute in parallel. The cilk_sync statement on line 11 indicates that the parent thread cannot continue until all child threads have completed and guarantees that the values of `a` and `b` are valid before returning the final value.

The values from the sequential and cilk versions are compared for correctness beginning at line 27. If they match, a P (pass) is printed along with the result. If they do not match, an F (fail) is printed along with both results. The print functions are described in Section 5.

Note that all variables in the program are of type `long` since this corresponds to the architecture's default 64-bit word size.

```
#include "memoryweb.h"
#include "cilk.h"

#define N 3

long fib_cilk(long n) {
    if (n < 2)
        return n;
    long a = cilk_spawn fib_cilk(n-1);
    long b = cilk_spawn fib_cilk(n-2);
    cilk_sync;
    return a + b;
}

long fib_seq(long n) {
    if (n < 2)
        return n;
    long a = fib_seq(n-1);
    long b = fib_seq(n-2);
    return a + b;
}

long main() {
    long result_seq = fib_seq(N);
    long result_cilk = fib_cilk(N);

    if (result_seq == result_cilk) {
         printf("Pass: %ld \n", result_cilk); // Pass
    } else {
        printf("Fail: %ld, expected %ld\n", result_cilk, result_seq); // Fail
    }
    return 0;
}
```

Figure 6.1: Simple Fibonacci program

```
1  #include "memoryweb.h"
2  #include "timing.h"
3  #include "stdio.h"
4
5  #define N 32 // 32 elements on 8 nodelets -> 4 elements per nodelet
6  #define VERIFY 0
7
8
9  int main()
10 {
11     long* A = mw_malloc1dlong(N);
12
13     starttiming();
14
15     // Generates a remote store (WRD) to initialize each element
16     for (long i=0; i < N; i++)
17         A[i] = 0;
18     printf(" A[0] = %ld\n", A[0]);
19
20     // Thread will migrate to each element and update it
21     for (long i=0; i < N; i++) {
22         A[i] += 1; // Use ATOMIC_ADDM(&A[i], 1) for multithreaded code
23         //REMOTE_ADD(&A[i], 1); // Replace prev line for remote atomic add
24     }
25     printf(" A[0] = %ld\n", A[0]);
26
27 #if VERIFY
28     // Thread will migrate to read element of list and then print
29     for (long i=1; i < N; i++)
30       printf(" A[%ld] = %ld\n", i, A[i]);
31 #endif
32
33     return 0;
34 }
```

Figure 6.2: Simple array update program with thread migration (`simpleArrayUpdate_v1.c`)

6.2 Simple Array Update

This section uses a series of simple programs that update the elements of a 1D array to illustrate program structure, demonstrate the various ways in which threads may access a data structure, and model how to interpret and use statistics generated by the simulator.

6.2.1 Migrating Array Update

The `simpleArrayUpdate_v1.c` program, shown in Figure 6.2, allocates a one dimensional array `A` of size 32 elements at line 11 using the function `mw_malloc1dlong()` found in `distributed.h`. The array is distributed round-robin across the nodelets (assume 8 nodelets for this example) with each consecutive element on a different nodelet. The resulting data distribution is shown in Figure 6.3. Note that there are 4 elements on each nodelet.

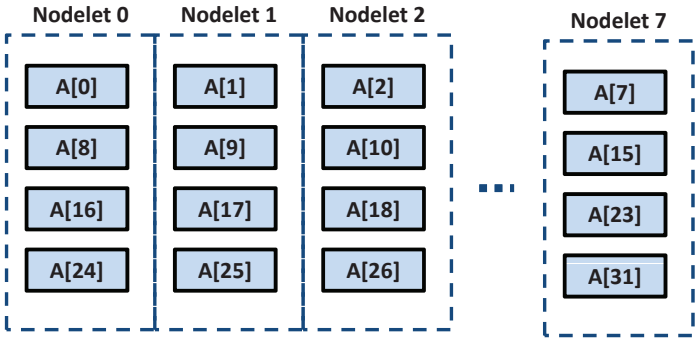


Figure 6.3: 1D array distribution on 8 nodelets using `mw_malloc1dlong()`

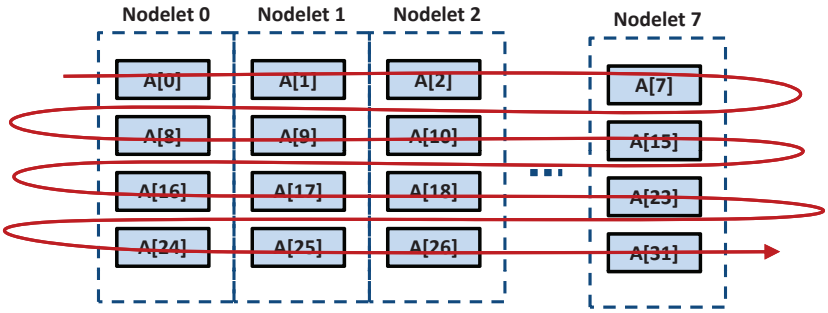


Figure 6.4: Thread migrates to each element of the 1D array and increments it

The `starttiming()` call at line 13 indicates the start of timing and statistics in the simulator. Timing in the current simulator continues until the end of the program's execution as there is not yet support for a `stoptiming()` function.

The loop at line 16 is used to initialize the array elements to 0. When the thread attempts to store A[i] and finds that it is not local, a remote write (WRD) is used to update the element. The compiler automatically generates `WRD` instructions for all stores. The `WRD` instruction is a remote write to memory that returns an acknowledgement (`ACK`) on completion. Therefore, in this loop the thread stays in the nodelet and issues remote writes to all the array elements. Note that a thread cannot leave a Gossamer core until all `ACKs` have returned.

In contrast, the loop at line 21 will cause the thread to migrate to each array element as shown in Figure 6.4. When the thread attempts to load `A[i]` and finds that it is not local, the hardware will automatically migrate the thread to that array element and restart the instruction so that the operation is local. If this were multithreaded code, the `ATOMIC_ADDM` intrinsic would be used to atomically add 1 to the array elements.

The final block of code can be used to read and print each element of the array for verification. The verification code is not executed in the current example so that the statistics files reflect only the initialization and update code.

The `.cdc` statistics file generated for every simulation run can be used to verify the expected behavior of the program. Note that Section 7.6 provides an overview of all the statistics files

generated by the simulator.

Figure 6.5 shows the beginning of the `simpleArrayUpdate_v1.cdc` file generated by this program. The file starts with the name of the program executed, simulator version, and system configuration. Line 21 gives the total execution time from the `starttiming()` call to the end of the program. The next lines show the total number of threads that were active at the end of the program as well as the total number created and ended during execution.

Line 29 begins the memory map. Each row in the memory map represents the source nodelet on which a memory instruction was encountered and the column represents the destination nodelet on which the memory instruction was ultimately executed. Values on the diagonal represent local memory accesses and non-diagonal values show the number of memory accesses that required a migration to another nodelet to complete. For example, line 30 is the total number of memory instructions encountered on nodelet 0 after the `starttiming()` call in the program. Of the total memory instructions encountered on nodelet 0, 1503 were for addresses on nodelet 0 and 7 were for addresses on nodelet 1 (requiring a migration to nodelet 1). The next row, at line 31, represents the memory accesses at nodelet 1 which indicates that 0 accesses were local and 8 required a migration from nodelet 1 to nodelet 2. This pattern of 8 migrations from nodelet i to i+1 represents the migration of the thread accessing the array elements in consecutive order, i.e. it will migrate from nodelet 0 to 1 to 2 to $\cdots$ 7 then wrap back to zero 4 times in the first loop and 4 times for the second loop. Note that the additional local memory operations on nodelet 0 are part of the printf functions and program completion.

Line 40 begins the remotes map. Similar to the memory map, each row in the table is a source nodelet and each column is a destination nodelet for a remote memory operation. Note that the diagonal values are always zero because a remote update to a local value will be counted as a local memory operation. In this example, all the values are zero showing that no remote memory operations occurred.

Part of writing an efficient program is understanding the underlying data distribution mechanisms and thread execution in order to structure the program in a way that provides sufficient parallelism while minimizing unnecessary migrations. The distribution of memory operations and remotes found in the statistics file is one resource that can be used to verify that the code is executing as intended.

```
Post Startup: CurrentRSS (MB)=25.7461
Post Startup: PeakRSS (MB)=25.8516
**********************************************
Program Name/Arguments:
simpleArrayUpdate_v1.mwx
**********************************************
Simulator Version: ES-2.0
**********************************************
Configuration Details:
Log2 Num Nodelets=3
Log2 Memory Size/Nodelet=33
Capture queue depths=false
Send ACKs from remotes=true
Core Clock=333 MHz, Pd=3.003
Memory bandwidth=1.892 GiB/s (2.032 GB/s)
SRIO SystemIC bandwidth=2.32 GiB/s (2.5GB/s)
**********************************************
Post SystemC Startup: CurrentRSS (MB)=25.8516
Post SystemC Startup: PeakRSS (MB)=25.9258
PROGRAM ENDED.
Emu system run time 0.000468 sec==468468000 ps
System thread counts:
       active=0, created=1, died=1,
       max live=1 first occurred 0 s with prog 0and last occurred
           0 s with prog 0% complete
Num_Core_Cycles=156000
Num_SRIO_Cycles=292792
Num_Mem_Cycles=118991
**********************************************
MEMORY MAP
1503,8,0,0,0,0,0,0
0,0,8,0,0,0,0,0
0,0,0,8,0,0,0,0
0,0,0,0,8,0,0,0
0,0,0,0,0,8,0,0
0,0,0,0,0,0,8,0
0,0,0,0,0,0,0,8
8,0,0,0,0,0,0,0

**********************************************
REMOTES MAP
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
```

Figure 6.5: Subset of statistics from `simpleArrayUpdate_v1.cdc`

6.2.2 Array Update Using Remote Memory Operations

The update loop (line 21) in Figure 6.2 will cause the thread to migrate to each element in order to add 1. It is also possible to use a `REMOTE_ADD` to update each element of the array without migrating, by replacing it with the loop shown in Figure 6.6. Note that the remote updates are atomic so there are no conflicts if multiple remotes are sent to the same memory location, either from a single thread or from multiple threads. Remote updates from a single thread to the SAME memory location are guaranteed to complete in order. However, remote updates to different addresses from the same thread or from multiple threads to the same address are not guaranteed to be in any particular order.

```
1    // Thread will send a remote add of 1 to each element
2    for (long i=0; i < N; i++)
3      REMOTE_ADD(&A[i], 1);
```

Figure 6.6: Simple array update loop using `REMOTE_ADD`

Figure 6.7 shows the memory map and remotes map when using the remote updates. In this example, only the first loop will migrate so the memory map now shows only 4 memory operations from nodelet i to i+1 instead of 8. The other 4 memory operations now appear in the remotes map at line 13, showing 4 remote writes sent from nodelet 0 to each of the nother nodelets.

```
1  MEMORY MAP
2  1508,4,0,0,0,0,0,0
3  0,0,4,0,0,0,0,0
4  0,0,0,4,0,0,0,0
5  0,0,0,0,4,0,0,0
6  0,0,0,0,0,4,0,0
7  0,0,0,0,0,0,4,0
8  0,0,0,0,0,0,0,4
9  4,0,0,0,0,0,0,0
10
11 ***********************************************
12 REMOTES MAP
13 0,4,4,4,4,4,4,4
14 0,0,0,0,0,0,0,0
15 0,0,0,0,0,0,0,0
16 0,0,0,0,0,0,0,0
17 0,0,0,0,0,0,0,0
18 0,0,0,0,0,0,0,0
19 0,0,0,0,0,0,0,0
20 0,0,0,0,0,0,0,0
```

Figure 6.7: Subset of statistics from simpleArrayUpdate_v1.cdc with `REMOTE_ADD`

6.2.3 Spawning Threads for Array Updates

Let's assume you wish to have different threads update different elements of the array. In this case you could spawn a new thread to update one or more elements of the array as shown in Figure 6.8.

NOTE: It is important to ensure that there is enough work for each thread to offset the overhead involved in spawning. Spawning a thread to execute a single instruction would NOT be efficient and is shown here only as a simple example.

```
1  #include "timing.h"
2  #include "stdio.h"
3
4  #define N 32  // 32 elements on 8 nodelets -> 4 elements per nodelet
5  #define VERIFY 0
6
7  void addOne(long *A) {
8      (*A) += 1;
9  }
10
11 int main()
12 {
13     long* A = mw_malloc1dlong(N);
14
15     starttiming();
16
17     for (long i=0; i < N; i++)
18       A[i] = 0;
19     printf(" A[0] = %ld\n", A[0]);
20
21     // Thread will migrate to the locale of each element
22     // and spawn a child thread to do work there
23     for (long i=0; i < N; i++) {
24         cilk_spawn addOne(&A[i]);
25     }
26     // Wait for spawned threads to complete
27     cilk_sync;
28     printf(" A[0] = %ld\n", A[0]);
29
30 #if VERIFY
31     // Thread will migrate to each element of list and then print
32     for (long i=0; i < N; i++)
33       printf(" A[%ld] = %ld\n", i, A[i]);
34 #endif
35
36     return 0;
37 }
```

Figure 6.8: Spawning threads to update array elements

The loop at line 23 uses `cilk_spawn` to indicate that this function can be done in parallel and that a child thread should be spawned to execute `addOne` for each iteration of the loop. The address of

the assigned array element is passed as a parameter. This allows the child thread to update the memory location but it is also used by the `cilk_spawn` to determine where the child thread should be spawned. If there are no addresses in the parameter list, the child thread will be spawned locally. Otherwise, the parent thread will migrate to the location of the first address in the parameter list and spawn the child thread there. This feature helps to ensure that any stack frames that might need to be created are created where the thread will ultimately execute, eliminating unnecessary migrations. For very small "leaf" functions a stack is typically not required. However, if there are additional function calls or additional thread spawning then stacks and synchronizations structures may be required.

For the current example, the parent will migrate through the elements of the array as shown in Figure 6.4 but instead of updating the element itself, it will spawn a child thread to update the element while the parent moves on to the next element. When the parent thread finishes spawning all the children, it will encounter the `cilk_sync`. This indicates that it cannot continue execution until all children have completed. The thread will check its synchronization structure to see if it is the last thread to reach the sync point. If so, it will continue execution. If not, it will store a "continuation" in the synchronization structure and quit. Upon completion, each of the child threads will migrate back to update the parent's synchronization structure. If a child thread is not the last one to return, it will quit. If it finds that it is the last one, it will pick up the continuation and continue executing from there.

Figure 6.9 shows a portion of the `.cdc` file for this program. At line 4, the total number of threads created and died are 33. This corresponds to the original thread plus the 32 child threads (one for each array element). Additional per nodelet information about the spawns can be found starting at line 29. The statistics given for each nodelet are (in order left to right):

- #_created: number of threads created on that nodelet
- #_spawns: number of child threads spawned on that nodelet
- #_failed_spawns: number of spawns that failed and became function calls
- #_quits: number of threads that quit on that nodelet
- #_migrates: number of threads that migrated from that nodelet
- #_rmos_in: number of remote memory operations whose destination is that nodelet
- #_rmos_out: number of remote memory operations originating from that nodelet
- #_mem_bw: memory bandwidth utilization on that nodelet (1.0 max)
- #_IPC: instructions per cycle for this nodelet (4.0 max)

From this per nodelet information, we can see that the spawns were uniformly distributed, with 4 spawns on each nodelet and none of the spawns failed. Spawn failures occur when the hardware decides that the spawn should not occur. If a thread tries to spawn and there are resource limitations on the current nodelet, the spawn will fail and be turned into a function call. Each core has enough hardware frames to support 64 threads with additional threads placed in a queue.

We can see that all 33 threads returned to nodelet 0 to quit because this is where the parent thread's synchronization structure was located.

The number of migrations per thread is also shown. Nodelet 0 had 4 thread migrations out and we can see from the memory map that they were to nodelet 1. Each of the other nodelets had 8

```
1  **********************************************
2  PROGRAM ENDED.
3  Emu system run time 6.15615e-05 sec
4  Threadlets: active=0, created=33, died=33
5  Num_Core_Cycles=20500
6  Num_SRIO_Cycles=38475
7  Num_Mem_Cycles=17545
8  **********************************************
9  MEMORY MAP
10 21,4,0,0,0,0,0,0
11 4,4,4,0,0,0,0,0
12 4,0,4,4,0,0,0,0
13 4,0,0,4,4,0,0,0
14 4,0,0,0,4,4,0,0
15 4,0,0,0,0,4,4,0
16 4,0,0,0,0,0,4,4
17 8,0,0,0,0,0,0,4
18 **********************************************
19 REMOTES MAP
20 0,4,4,4,4,4,4,4
21 0,0,0,0,0,0,0,0
22 0,0,0,0,0,0,0,0
23 0,0,0,0,0,0,0,0
24 0,0,0,0,0,0,0,0
25 0,0,0,0,0,0,0,0
26 0,0,0,0,0,0,0,0
27 0,0,0,0,0,0,0,0
28 **********************************************
29 Module: #_created, #_spawns, #_failed_spawns, #_quits, #_migrates,
30 #_rmos_in, #_rmos_out, mem_bw, IPC
31 NLET[0]: 1, 4, 0, 33, 4, 0, 28, 0.00706754, 0.0300976
32 NLET[1]: 0, 4, 0, 0, 8, 4, 0, 0.000911941, 0.0062439
33 NLET[2]: 0, 4, 0, 0, 8, 4, 0, 0.000911941, 0.0062439
34 NLET[3]: 0, 4, 0, 0, 8, 4, 0, 0.000911941, 0.0062439
35 NLET[4]: 0, 4, 0, 0, 8, 4, 0, 0.000911941, 0.0062439
36 NLET[5]: 0, 4, 0, 0, 8, 4, 0, 0.000911941, 0.0062439
37 NLET[6]: 0, 4, 0, 0, 8, 4, 0, 0.000911941, 0.0062439
38 NLET[7]: 0, 4, 0, 0, 8, 4, 0, 0.000911941, 0.006
```

Figure 6.9: Subset of statistics from simpleArray_v3.cdc

thread migrations: 4 to the next nodelet and 4 back to nodelet 0. The migrations from nodelet i to i+1 are the parent thread migrating to each array element to spawn a child thread. The migrations back to nodelet 0 are the child threads returning to synchronize and quit.

The remotes map at line 19 shows that Nodelet 0 has 28 outgoing remote memory operations (RMOs) and each of the other 7 nodelets has 4 incoming RMOs from the initialization loop.

Finally, the memory bandwidth and IPC are shown as the last two columns in the per nodelet statistics.

6.3 Simple Array Sum

The simple program, `SimpleArraySum.c`, shown in Figure 6.10 spawns a child thread to sum the elements of array `A` while it (parent thread) sums the elements of array `B`. The program uses `mw_malloc2d` to define two arrays distributed across the nodelets with a block of `epn` consecutive elements on each nodelet. For 8 nodelets, this is 4 consecutive elements on each nodelet as shown in Figure 6.11.

The child thread and parent thread sum the values of their respective arrays concurrently. Once the parent has completed its sum, it waits for the child thread to complete using `cilk_sync`. Then it prints the sums for both `A` and `B`.

The memory map for this example is shown in Figure 6.12. You might expect the program to generate 2 migrates from nodelet i to i+1 (one for each thread). However, there is only 1 migration from i to i+1 but there are additional migrations from i to 0 and 0 to i for each i. In order to understand this behavior, the `--short_trace` flag can be used with the simulator to generate a trace of spawns, migrations, and quits for each thread in the system.

The first lines of the resulting trace is shown in Figure 6.13. Each line of the trace shows at least the following information:

- time step: execution time
- event: MIGRATE, SPAWN, or QUIT(DONE)
- nodelet: nodelet and core where the event occurred and destination nodelet for migrations
- TID: thread ID (for a spawn shows both parent and child TIDs)
- TPC: thread program counter (Note that the program counter can be matched to a line of code in the object dump to assist is debugging. This is described in more detail in Section 7.2.)
- instruction: name of the instruction executed

From this trace you can see that the main thread is TID[0] and it spawns TID[1] at timestep 1348347. TID[1] migrates as expected `(0->1->2->..)` through the nodelets from NLET[0] @4924920 to NLET[1] @7957950 to NLET[2] @ 10981971, etc. However, TID[0] migrates back to nodelet 0 each time `(0->1->0->2->...)`: from NLET[0] to NLET[1] @5237232, back to NLET[0] @8363355 before migrating to NLET[2] @8882874.

This pattern usually indicates that the thread is returning to load a value from nodelet 0, most frequently because a register was spilled to the stack. Adding `noinline` to the `sum` function prevents it from being inlined and eliminates the extra migrations.

In any routine with a spawn (like this main routine), there is likely to be register spilling. It is often preferable to isolate heavy computation like the doubly nested loop by separating it into a separate routine. There are also cases where using `noinline` to enforce the separation will improve performance. (Note: this is not necessary if the routine is only spawned and never called.) The compiler optimizations continue to evolve, but for optimum performance the programmer may need to closely analyze the program's execution characteristics.

```
#include "memoryweb.h"
#include "cilk.h"
#include "timing.h"
#include "stdio.h"

#define SIZE 32


long sum(long **array, long epn) {
    long sum = 0;
    for (long i=0; i<NODELETS(); i++)
      for (long j=0; j<epn; j++)
        sum += array[i][j];
    return sum;
 }

int main()
{
    // Allocate arrays
    long epn = SIZE/NODELETS(); // elements per nodelet
    long** A = (long **) mw_malloc2d(NODELETS(), epn * sizeof(long));
    long** B = (long **) mw_malloc2d(NODELETS(), epn * sizeof(long));
    long sumA;
    long sumB;

    // Initialize array values
    for (long i=0; i<NODELETS(); i++)
      for (long j=0; j<epn; j++) {
        A[i][j] = 1;
        B[i][j] = 2;
      }

    starttiming();

    // Spawn threadlet to sum values in A
    sumA = cilk_spawn sum(A, epn);

    // Call function to sum values of B (in current threadlet)
    sumB = sum(B, epn);

    // Wait for spawned threadlet to complete
    cilk_sync;

    printf("A = %ld, B = %ld\n", sumA, sumB);

    return 0;
}
```

Figure 6.10: Simple Array Sum program

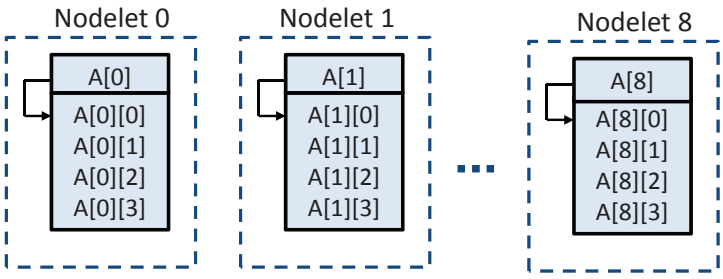


Figure 6.11: 2D array distribution on 8 nodelets using `mw_malloc2d()`

```
1 MEMORY MAP
2 1108,2,1,1,1,1,1,1
3 1,8,1,0,0,0,0,0
4 1,0,8,1,0,0,0,0
5 1,0,0,8,1,0,0,0
6 1,0,0,0,8,1,0,0
7 1,0,0,0,0,8,1,0
8 1,0,0,0,0,0,8,1
9 3,0,0,0,0,0,0,8
```

Figure 6.12: Memory Map for Simple Array Sum

```
Start untimed simulation with local date and time= Thu Jun  2 16:34:16 2016
End untimed simulation with local date and time= Thu Jun  2 16:34:16 2016
SysC Enumeration done. Program launching...
Simulation @0 s with local date and time= Thu Jun  2 16:34:16 2016

@330330 ps: on NLET[7].GC[1] TID[0] MIGRATE out to NQM TID=0,
TPC=8871, instName=LDE and going to NLET[0]

@1348347 ps: SPAWN out to NQM NLET[0].GC[0] with parent TID[0]
and child TID[1] with TID=1, TPC=8962

@4924920 ps: on NLET[0].GC[2] TID[1] MIGRATE out to NQM TID=1,
TPC=8253, instName=LDE and going to NLET[1]

@5237232 ps: on NLET[0].GC[0] TID[0] MIGRATE out to NQM TID=0,
TPC=9087, instName=LDE and going to NLET[1]

@7957950 ps: on NLET[1].GC[2] TID[1] MIGRATE out to NQM TID=1,
TPC=8253, instName=LDE and going to NLET[2]

@8363355 ps: on NLET[1].GC[0] TID[0] MIGRATE out to NQM TID=0,
TPC=9083, instName=ADD and going to NLET[0]

@8882874 ps: on NLET[0].GC[3] TID[0] MIGRATE out to NQM TID=0,
TPC=9087, instName=LDE and going to NLET[2]

@10981971 ps: on NLET[2].GC[0] TID[1] MIGRATE out to NQM TID=1,
TPC=8253, instName=LDE and going to NLET[3]

@12005994 ps: on NLET[2].GC[0] TID[0] MIGRATE out to NQM TID=0,
TPC=9083, instName=ADD and going to NLET[0]

@12525513 ps: on NLET[0].GC[1] TID[0] MIGRATE out to NQM TID=0,
TPC=9087, instName=LDE and going to NLET[3]

@14018004 ps: on NLET[3].GC[0] TID[1] MIGRATE out to NQM TID=1,
TPC=8253, instName=LDE and going to NLET[4]

@15663648 ps: on NLET[3].GC[1] TID[0] MIGRATE out to NQM TID=0,
TPC=9083, instName=ADD and going to NLET[0]

@16189173 ps: on NLET[0].GC[2] TID[0] MIGRATE out to NQM TID=0,
TPC=9087, instName=LDE and going to NLET[4]

@17048031 ps: on NLET[4].GC[0] TID[1] MIGRATE out to NQM TID=1,
TPC=8253, instName=LDE and going to NLET[5]

@19321302 ps: on NLET[4].GC[0] TID[0] MIGRATE out to NQM TID=0,
TPC=9083, instName=ADD and going to NLET[0]

@19846827 ps: on NLET[0].GC[0] TID[0] MIGRATE out to NQM TID=0,
TPC=9087, instName=LDE and going to NLET[5]
```

Figure 6.13: Short Trace for Simple Array Sum

6.4 Random Access Benchmark (GUPS)

This section describes an implemenation of the Random Access benchmark also known as GUPS. This benchmark uniformly distributes a table across the nodelets in the system. It then generates a sequence of random numbers that are used as indices into the Table and executes an `XOR` with that table element. This generates random accesses distributed throughout the system.

Figures 6.14, 6.15, and 6.16 show the implementation for the Emu system. This implementation distributes the table round robin across with nodelets in the system with each element on a different nodelet. It then spawns a set of worker threads at each nodelet that each generate a subset of the updates. Each thread generates a random number then uses a `REMOTE_XOR` to update the table element.

```
1  #include "memoryweb.h"
2  #include "cilk.h"
3  #include "timing.h"
4
5  #define VERIFY 0
6
7  // constants for RNG
8  #define K (1L << 43)
9  #define A_R ((1L << 62) + (1L << 53) + (1L << 41) + (1L << 36) + (1L << 23) + (1L <<
       17) + 1) // congruent to 1 mod 4
10 #define C_R 3898255708540604102L // congruent to 2 mod 4
11 #define A_L (6*K + 1)
12 #define C_L 0
13 #define X_0 1 // must be odd
14
15 // Total array elements MUST BE POWER OF 2 and must be > N*T
16 #define E (1<<15) // 2^15 elements
17 #define T 256 // Threadlets per nodelet
18 #define U 4 // Updates factor = number updates per element
19
20 replicated unsigned long * Table;
21 replicated long nUpdates = E*U; // Needs to be computed so initialize each copy
22 replicated long nThreadlets = T;
```

Figure 6.14: Random Access Benchmark (GUPS)(part 1)

First note that `Table` is defined at line 20 as a replicated variable. This indicates that a copy of `Table` is found on each nodelet at the same offset and accessing it will always access the local copy. Using replicated variables may help to eliminate register spills and/or migrations. The table array is allocated at line 74 using `mw_malloc1dlong`. The address of the beginning of the array is then copied into the replicated variable `Table` on each nodelet using the `mw_replicated_init` function provided in the `data_helper.h` file. Finally the table elements are initialized by the loop at line 75.

The `cilk_spawn` at line 81 begins the recursive spawning of threads at each nodelet. This `recursive_spawn` function generates a spawn tree to create one "master" thread for each nodelet, as shown in Figure 6.17. This master thread at each nodelet then generates `T` local `RandomAccess` worker threads that each generate a subset of the table updates.

```
24 static
25 void RandomAccessUpdate(unsigned long * dest, unsigned long tUpdates,
26                       unsigned long *table, unsigned long X,
27                       unsigned long A, unsigned long C) {
28
29     // Expected Per Update: 1 remote add, no migrations
30     for (long i = 0; i < tUpdates; i++) {
31         X = A*X + C;
32         unsigned long r = X >> 8; // since lowest bits are notoriously crummy in an
               LCG
33 #if VERIFY
34         REMOTE_ADD((long *) &(table[r & (E - 1)]), 1);
35 #else
36         REMOTE_XOR((long *) &(table[r & (E - 1)]), 1);
37 #endif
38     }
39 }
40
41 // Recursive tree to spawn T worker threads at each nodelet
42 static
43 void recursive_spawn(unsigned long *dest, unsigned long low,
44                      unsigned long high, unsigned long X) {
45     // Updates per threadlet
46     long tUpdates = nUpdates / (NODELETS() * nThreadlets);
47     unsigned long *table = Table;
48
49     // Want a grainsize of 1 to spawn a threadlet at each nodelet
50     for (;;) {
51         unsigned long count = high - low;
52
53         // spawn a threadlet at each nodelet
54         if (count <= 1) break;
55
56         // Invariant: count >= 2
57         unsigned long mid = low + count / 2;
58         X = A_R*X + C_R;
59         cilk_spawn recursive_spawn(&(table[mid]), mid, high, A_L*X + C_L);
60         high = mid;
61     }
62
63     for (long i=0; i < T-1; i++) {
64         X = A_R*X + C_R;
65         cilk_spawn RandomAccessUpdate(&(table[low]), tUpdates, table,
66                                       A_L*X + C_L, A_R, C_R);
67     }
68     RandomAccessUpdate(&(table[low]), tUpdates, table, X, A_R, C_R);
69 }
```

Figure 6.15: Random Access Benchmark (GUPS)(part 2)

```
71  long main(int argc, char **argv) {
72
73      // Table is an array distributed round robin
74      unsigned long * t = (unsigned long *) mw_malloc1dlong(E);
75      mw_replicated_init((long *) &Table, (long) t);
76
77      for (long i=0; i < E; i++)
78          Table[i] = 0;
79
80      starttiming();
81      cilk_spawn recursive_spawn(Table, 0, NODELETS(), X_0);
82      cilk_sync;
83
84  #if (VERIFY)
85      // Verify results
86      long sum = 0;
87      for (long i=0; i < E; i++)
88          sum += Table[i];
89
90      if (sum == (nUpdates)) {
91              printf("Pass: %ld\n", sum);
92              return 0; // SUCCESS
93      }
94      else {
95              printf("Fail: %ld expected %ld\n", sum, nUpdates);
96              return 1; // FAILURE
97      }
98  #endif
99
100     return 0;
101
102 }
```

Figure 6.16: Random Access Benchmark (GUPS)(part 3)

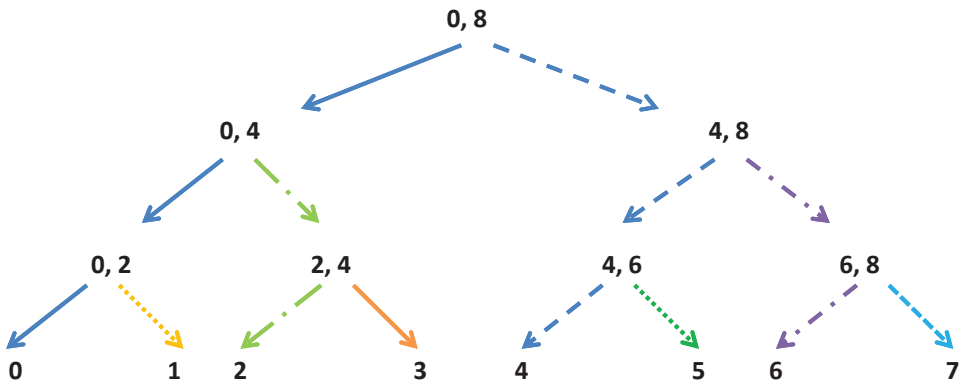


Figure 6.17: Recursive Spawn Tree on 8 nodelets

Chapter 7

Simulation Execution

7.1 Simulation Overview

The Emu Simulation Environment (`emusim`)[1] executes Emu1 machine instructions in a blend of an untimed functional model and a SystemC timed architectural performance model. Various command line arguments can be passed to `emusim` to configure the simulation and collect various statistics. It is generally expected that `emusim` will be running simulations in the SystemC based architectural model. Section 7.3 describes how to control which model is used.

The functional model models the system as a set of nodelets that are executing packets[2] without any concept of timing. Threads in this model do maintain their state information (registers, program counter, etc) and reside on a nodelet at all times. An execution "cycle" consists of all nodelets and all packets performing a single operation and moving to a different nodelet as required. This model is practical for determining program correctness and generating initial statistics related to where spawns and memory references occur.

The SystemC architectural performance model is designed to provide a near cycle-accurate model of how an application will run on the actual hardware. Within this model, the packets are treated as tokens and move amongst the various modules within the system to perform their operations. `emusim` is frequently being updated to improve the fidelity between the simulator and the actual hardware implementation. The large majority of `emusim` options target this model.

Figure 7.1 shows the organization of a single node within the architectural model; each node consists of a migration engine (ME) and 8 nodelets. Each nodelet contains the following:

- Gossamer Cores (GCs, x4): These units are heavily multithreaded cores that execute program threads. Up to 64 threads can be active in each GC.
- Nodelet Queue Manager (NQM): This unit moves threads in and out of the GCs. It also sends outgoing packets to the Migration Engine and sends all incoming remote operations and threads (only as necessary) to the Memory Front End.
- Memory Front End (MFE)/Memory: The MFE is an interface and management layer for the GCs, NQM, and stationary cores to interact with memory. All memory transactions are

[1] The simulator executable is `emusim.x`.

[2] a packet is a thread, remote operation, or acknowledgment

managed through this block.

The Migration Engine (ME) is responsible for routing packets between nodelets and the system interconnect as dictated by the packet source and destination. It contains a set of incoming queues and a set of outgoing queues where there is a queue for each incoming and outgoing link. Within the ME, packets arrive on incoming queues and their destination is determined. The incoming queue then requests to send to the appropriate outgoing and once access is granted, the packet moves from an incoming queue to an outgoing queue to be sent to either a specific nodelet or the system interconnect.

For reference, there are 28 queues in the ME: 14 incoming and 14 outgoing. There is an incoming and outgoing queue to each nodelet NQM (8 of each queue type). The remaining 6 incoming and outgoing queues are connected to the system interconnect.

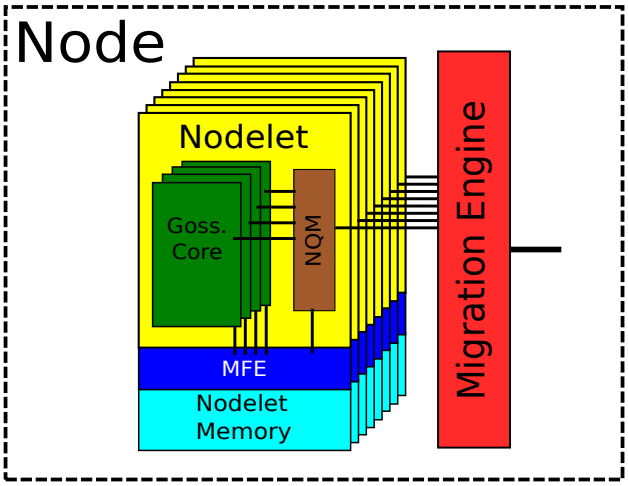


Figure 7.1: SystemC Architectural Model of a Node

For the Chick system model, the system interconnect is modeled directly and expected to be close to actual hardware execution. For the regular system model, the nodes are wired together through a simple system interconnect to route all packets to their appropriate destinations. Packets do experience a delay when being routed through the system interconnect, but effects such as collisions at the switches are not modeled.

While nodes in the full Emu1will have stationary cores attached to them, these are not explicitly modeled in `emusim` at this time.

7.2 Application Development

A customized compilation toolchain has been developed to compile applications for the Emu system architecture. A script, `emu-cc`, is provided with the Emu Software Tool Set to manage the compilation process; this script can be used much like gcc/clang. For reference, the default extension for executables intended to run on the Emu architecture is `.mwx`.

Application failures, such as seg faults, are typically observed as exceptions in the simulator and on the hardware. Frequently, the first step in debugging these failures is to identify the offending

instruction (TPC = Thread Program Counter) in both the exception message and the disassembled executable. The `gossamer64-objdump` tool has been ported to the Emu architecture and this tool can be used to generate a program's symbol table using the `-t` flag or disassembled using the `-d` flag. Documentation on using this tool can be found in the Linux manual (`man objdump`). The disassembly format is the only divergence from the standard tool. The format is as follows:

```
Instruction Byte-address: Instruction Nibble-address: Disassembled Instruction
```

Other files can be generated with `emu-cc` to further aid in debugging. In particular, users can output `.ll` (LLVM IR) and `.s` (assembly) files which can aid in correlating a line of C code to assembly code for debugging.

NOTE: The applications built by the emu-17.Y-hw series toolchains and the emu-{18,19}.Y series of the combined toolchain have a "shared" bit set. This is bit 31 in program counters, e.g. 0x8000_2000, and bit 55 in addresses, e.g. 0x0180_0000_0000_8128. When running on the simulator, this bit can be safely ignored. When running on the hardware, this bit is only an issue if it is no longer set to 1. This bit will be set when viewing a disassembled executable program; thus, `Instruction Byte-address` values will start with a 4 and `Instruction Nibble-address` values will start with an 8.

7.3 Simulation Control Functions

Table 7.1 shows the functions used to control simulation execution and statistics in `emusim`. These functions are defined in the `timing.h` file which is included when using the `memoryweb.h` file.

See Section 4.5 for information on using the `CLOCK()` intrinsic for timing comparisons in the simulator and on the hardware. Note also that the `time.h` functions, such as `clock_gettime()`, use the SC clock. The time involved in doing a system call to the SC to get the time may be significant for small programs, so `CLOCK()` is the preferred mechanism for timing.

The following subsections define each function and describe their operation and usage.

Table 7.1: Statistics Control Functions

Function	Description
void starttiming()	• Ends the functional portion of the simulation and moves all threads to the architectural model. Simulation restarts in the architectural performance model. • No-op in the hardware.
void stoptiming()	**Not yet implemented**

7.3.1 starttiming

```
void starttiming()
```

Return Value: None

Arguments:

- None

Functionality: Within `emusim` programs are initially started in the untimed model and after a call to `starttiming()` in the source code, the untimed model will complete all migrations, remote operations, and acknowledgments. All threads that exist will be moved from the untimed nodelet that models them to the equivalent architectural nodelet. Once all threads have been moved to their proper architectural nodelets, the simulation begins executing in the architectural model. If no call to `starttiming()` in the source code is made, the simulation runs completely in the functional model.

If the `--ignore_starttiming` flag is given to `emusim`, the statistics being gathered by the untimed functional model are reset when the call to `starttiming()` is made. This allows the programmer to view statistics for the same portion of the program.

Notes: While it is safe to make multiple calls to this function, it is recommended to call it only once.

Example:

7.4 Using the Simulator

The simulator is executed as follows:

```
emusim.x [<emusim options>] cilk_example.mwx <mwx options>
```

It is recommended that applications be verified with the functional model before using the timing model!

Table 7.2 lists the command line arguments that are currently recognized.

Table 7.2: Command Line Arguments

Command Line Argument	Action
Short/Long Options	
-h, --help	Prints the command line argument options and short descriptions.
-n, --log2_num_nodelets	Set log2 number of nodelets (default=1 node, range=single_node-12).
-m, --log2_memory_size	Set log2 memory size per nodelet (default 33=8GB, range 20-40).
-o, --base_ofile	Set the base file name for all output files (default is program name without the `.mwx` extension).
Long Options	
--verbose_isa	Print each ISA instruction as executed.

--verbose_tid	Print each ISA instruction as executed for a specific thread number. This is typically best done only after a simulation has failed and you know which thread you would like to make verbose.
--short_trace	Print spawn and quit instructions; also print instructions causing migrations.
--untimed_short_trace	For the untimed (functional) simulation, print spawn and quit instructions; also print instructions causing migrations.
--memory_trace	Perform a memory trace that is written out to a `.mt` file.
--output_instruction_count	Collect instruction count data for each function in the application. Data is written to a `.uis` file.
--verbose_resize	Print out details of all RESIZE functions.
--capture_timing_queues	When doing a timed simulation, collect statistics for various queues/resources in the system to use with the visualization tool. Data is written to a `.tqd` file.
--timing_sample_interval	Set timing model queue depth sampling frequency in ns (default 10000; equivalent to a 100 KHz clock frequency). Requires --capture_timing_queues also.
--max_sim_time	Set the max simulation time in milliseconds (floating point input is accepted).
--core_clk_mhz	Set the core clock rate in MHz (default is 300).
--turn_acks_off	Turn off acknowledgments for remote operations.
--ignore_starttiming	Run simulation purely in untimed mode; stats will be reset if a call to `starttiming()` is made in the source program.
--use_thread_ids	Turns on a bit in the thread state that adds a thread ID word to each thread (generally not necessary for simulation).
--chick_box	Simulate 8 node Chick model (-n flag not required).
--return_value	Output the return value from your program as an integer.
--return_value_hex	Output the return value from your program as a hex value.
--forward_return_value	If the simulation finishes correctly, the exit code of the simulator will be that of the simulated program rather than that of the simulator.
--gcs_per_nodelet	Gossamer cores per nodelet (default=4, range=1-4)
--ddr_speed	Set DDR4 speed grade (1=1333, 2=1600, 3=1866, 4=2133 (default), 5=2400, 6=2666)
--model_hw	Set various parameters to match the current 8 nodelet hardware build (clock of 175MHz, ddr_speed=2, 1 Gossamer core per nodelet)
--model_4nodelet_hw	Set various parameters to match the current 4 nodelet hardware build (clock of 175MHz, ddr_speed=2, 3 Gossamer cores per nodelet)
--log2_nodelets_per_node	Vary the number of nodelets per node (default=3, range=1-4)

--initialize_memory	This will initialize all memory to garbage values which may be useful for catching programs with bad pointers. However, this is resource intensive and the simulator will try to prevent you from using this if the simulated memory total is larger than the physical memory on the system running the simulator.

7.4.1 Simulator Configuration and Parameters

The list below summarizes many of the system configuration details in the timed model:

- Default of 1 node (configurable at command line)
- 8 nodelets per node
- 4 Gossamer cores per nodelet
- Default of 16 nodes per motherboard (128 nodelets per motherboard)
- "Perfect" network in that there are no collisions, dropped packets, broken links, etc.

The list below summarizes many of the clock rates and latencies assumed by the timed model:

- Clock frequency of 300 MHz
- Memory bandwidth of 1.892 GiB/s (2.027 GB/s)
- Memory latency of 82.7ns
- Incoming and outgoing links to system interconnect operate at 2.5 GB/s (2.32 GiB/s).
- All hops on the system interconnect require a latency related to the packet length and a bandwidth of each link equal to 2.5 GB/s.
- All switches that must be traversed by a packet on the system interconnect add a latency of 100ns (per IDT RapidIO switch data).

The baseline defaults above are where the design may be in the future. Use the `--model_hw` or `--model_4nodelet_hw` flag to configure the system parameters as they currently exist.

7.5 `emusim` Screen Output

`emusim` does generate a modest amount of output to the screen with each simulation in default mode. Additionally, the application program itself may write data to the screen.

The main output generated by `emusim` is a progress marker that is printed every 1ms of simulated Emu1 machine time with the time and date of the system. This allows a user to ensure that the simulation is progressing. Should the simulator detect that no work has been done during the prior 1ms period, the simulator will halt with the message that "No work done this output tick...stopping simulation". This should prevent most cases of an infinite program.

Other screen output may be observed if there is some form of error in the simulation. The most common cause of this is when an architectural queue has been overflowed; future updates to `emusim` will prevent these overflows from occurring.

The `--short_trace` and all of the `--verbose` options print data to the screen. It is strongly recommended that screen output be redirected to a file if using any of these options due to the amount of data they generate.

7.6 `emusim` File Output

Each run of `emusim` uses either the executable file name (without the `.mwx` extension) or the name specified with the `-o`, `--base_ofile` command line argument to generate two output files. The first output file has a `.cdc` extension and contains the configuration details and some basic performance statistics. The second output files has a `.vsf` extension and contains more verbose statistics information. The contents of these files, especially the `.vsf` file, changes considerably when running in untimed functional rather than timed mode. Other output files can be generated by using the proper command line arguments as defined in Table 7.1.

The `.gcd` output file is not discussed here as it is used only for simulator debugging and development and may be removed in future versions of the simulator.

7.6.1 Configuration Data and Summary Performance Output (.cdc)

The following list describes the data written to the `.cdc` file when doing a timed architectural simulation:

- System Configuration details including clock rates and bandwidths.
- Emu system run time and number of cycles for the core, interconnect, and memory.
- Number of threads that are active, created, and died. Created and died should be equal if the simulation ran to the end of the program.
- Memory Reference Map
- Remotes Reference Map
- Simulation wall clock time

The interconnect and memory clock rates are separate from the core clock frequency and are individually set. These rates are set to match that data bandwidths through these units with the interconnect clock transferring 4-bytes/cycle and the memory clock transferring 8-bytes/cycle.

The Memory Reference Map is computed during the architectural simulation; entry *(i,j)* in the map is the number of memory references made by threads on nodelet i to nodelet j. When $i == j$, it is the count of references to local memory; additionally, the sum of column j is the number of memory references on nodelet j (excluding remote operations). The sum of row i, excluding where $i == j$, is the number of migrations away from nodelet i.

The Remotes Reference Map is similar to the Memory Reference Map except that it counts the number of remote operations from threads on nodelet i to nodelet j. If $i == j$, the Remotes Reference Map is not updated, but instead, this operation is included in the Memory Reference Map. Thus, the diagonal of the Remotes Reference Map is 0.

The `.cdc` also collects data for each node in the system (nodes are numbered in multiples of 8) as follows:

- Nodelet performance data which includes the following:
  - Number of threads created on this nodelet when `starttiming()` was executed (`#_created`)

  - Number of threads spawned
  - Number of threads quitting (dying)
  - Number of migrations away from this nodelet
  - Number of incoming remote operations
  - Number of outgoing remote operations
  - Memory bandwidth utilized
  - IPC achieved by the nodelet (max is number of GCs per nodelet)
- Migration Engine component performance data which is collected for each queue:
  - For the ME queues incoming from the NQMs (ME[ ].FromNQM[ ]):
    * Number of incoming transactions
    * Incoming bandwidth utilization
    * Number of packets being sent to other nodelets on the same node (`#_to_nqms`)
    * Number of packets being sent to the system interconnect
    * Additional data for packet routing through the ME (non-zero only for Chick system model).
  - For the ME queues incoming from the system interconnect (ME[ ].FromSysIC[ ]):
    * Number of incoming transactions
    * Incoming bandwidth utilization
    * Outgoing bandwidth utilization
    * Additional data for packet routing through the ME (non-zero only for Chick system model).
  - For the ME queues outgoing to the NQMs (ME[ ].ToNQM[ ]):
    * Number of outgoing transactions
    * Outgoing bandwidth utilization
    * Additional data for packet routing through the ME (number of requesters to this output queue)
  - For the ME queues outgoing to the system interconnect (ME[ ].ToSysIC[ ]):
    * Number of outgoing transactions
    * Outgoing bandwidth utilization
    * Additional data for packet routing through the ME (number of requesters to this output queue)

The incoming and outgoing bandwidth utilization numbers for the ME queues connected to the system interconnect do vary as the node side bandwidth differs from the RapidIO bandwidth.

Additionally, these queues modify the packet overheads to account for the differences between the RapidIO network and the on-node network.

7.6.2 Verbose Statistics Information (.vsf)

When running an untimed simulation, this file provides a condensed version of the `.cdc` file output. For each nodelet, there are three counters displayed: outbound migrations, threads created, and threads died. If the simulated program calls `starttiming()` and the `--ignore_starttiming` simulator flag is used to run in untimed mode only, these counters are reset when `starttiming()` is executed.

When running a timed simulation, the `.vsf` file contains additional statistics that are not included in the `.cdc` file as detailed below.

Per nodelet data gathered includes

- Maximum number and time(s) of packets in each queue in the NQM
- Additional instruction based statistics:
  - Instructions retired
  - Number of threads removed from a GC due to a timeout interrupt
  - Number of threads rescheduled (user requested)
  - Number of service requests made
  - Instructions flushed: instructions that issued but could not be completed due to a resource being unavailable
  - Fences retired: these occur when an instruction is ready to execute, but cannot as the thread is still waiting for acknowledgments to return.
- Additional memory statistics:
  - Number of memory operations: number of incoming memory requests to the memory front end
  - Number of memory transactions: number of read/write operations in memory. This number is larger than the number of memory operations as atomics and remote operations count as two transactions (one for the read, one for the write)
  - Number of releases: should match the number of threads that quit on the nodelet
- GC-NQM statistics: for these statistics, the values in each GC are summed to get the total
  - Count of the number of cycles (stall_length[ ]) a thread waited in GC before the NQM could service it
  - Number of transactions for each packet length (gc_to_nqm_lengths[ ]) leaving the GCs and going to the NQM
  - outgoing_xactions, outgoing_cycles, outgoing_bw_utilization: totals and utilization information for transferring data from the GCs to the NQM

  - Similar data for the last two above items is collected for NQM to GC transactions

- NQM Total Stats:
  - Number of incoming and outgoing transactions for each packet length
  - Summary counters and bandwidth utilization for incoming and outgoing packets

For the ME incoming queues connected to the NQMs (ME[ ].FromNQM[ ]), the data collected is all for packets coming into the ME. Specific data gathered here is the number of transactions for each packet length and summary counters for the number of transactions, in busy cycles (sum of packet lengths multiplied by number of incoming transactions), and incoming bandwidth utilized.

The ME incoming queues connected to the system interconnect (ME[ ].FromSysIC[ ]) collect the number of packets for each length going in to and out of these queues. **Note:** The incoming packet lengths are for 32-bit words and the outgoing packet lengths are for 64-bit words. Incoming and outgoing bandwidth utilization is also presented.

The data collected for the ME outgoing queues connected to the NQMs (ME[ ].ToNQM[ ]) is the same as that for the ME[ ].FromNQM[ ] blocks except that it is for outgoing rather than incoming packets.

The data collected for the ME outgoing queues connected to the system interconnect (ME[ ].ToSysIC[ ]) is the same as that for the ME[ ].FromSysIC[ ] blocks except that outgoing packets are now made up of 32-bit words and incoming packets are made up of 64-bit words.

A large number of additional statistics have also been gathered for each queue type in the Migration Engine. Many of these have been collected in an effort to determine where performance was being limited in the Chick system model. These statistics are generally not required by the average user, but may be useful when trying to gain a deeper understanding of application performance. Some of these statistics may not exist in future iterations of the simulator.

For non-Chick model simulations, there is another module, labeled ME[ ].DisAggr, that is a simulator construct (does not occur in real hardware) that all packets coming into the node from the system interconnect pass through. For this module, the number of transactions for packet lengths is collected (was used for debugging the simulator).

7.6.3 Instruction Execution Statistics (.uis)

The `.uis` file consists of a set of instruction counts for each function in the program. Within each function, the number of migrations and the number of registers in the thread for each migration (migrations_with_reg_count[ ]) are also tracked. Additionally, the average number of registers removed at each resize (noted by TPC, Thread Program Counter) instruction is also collected.

7.6.4 Timed Queue Depths (.tqd)

The depth of a variety of queues in the timed model can be captured by using the `--capture_timing_queues` option. Using this option will generate a `.tqd` file which consists of a header, various queue and counter headers, and a list of queue depths at each sampling time step for the time period of interest. This file is intended for use with the visualization tools rather than being viewed as a text file.

The header contains the number of nodes (FPGAs), number of queues being measured, number of time steps measured, and the time step between measurements (ns). The queue headers contain the queue being measured, the maximum depth of the queue, and the node containing the queue. A queue depth larger than the maximum can occur; the maximum is used by the visualization tools to identify when there are too many threads in the queue. The counter headers consist of three thread counters: LIVE, BORN, DEAD. These measure the number of threads that are alive at a given time step and the cumulative number that have been created and died from the start of the simulation to the current time. These queue and counter headers identify the order of the data in each of the remaining lines of the `.tqd` file.

By default the sampling interval between timesteps is set to 10000 ns, but this can be changed using the command line argument `--timing_sample_interval`.

7.6.5 Memory Tracing (.mt)

The simulator can perform a memory trace of the full simulation by using the `--memory_trace` option. This option will generate a `.mt` file which is a CSV file with a header row. It is expected that this file will be processed with other tools, rather than being viewed as a text file. When generating this file, each data memory access will be printed into the file with the following information (as listed by the header row):

- TID: thread ID
- SRC: source nodelet
- Type: All memory accesses are grouped into one of three types:
  - L: local memory access. Remotes to the local nodelet are given this type. Also, threads that are doing their first memory access post-migration are NOT listed here (i.e., the access that caused the migration is only given an M type).
  - R: remote memory operation (remote packet generated)
  - M: memory access causing a migration
- DEST: destination nodelet. For Type L accesses, this will match SRC.
- @cycle: the cycle number where the access occurred. This value is not reset when switching from functional to timed mode; the timed mode starts its cycle count by adding 1 from the last cycle in the functional mode.
- TPC(0x): Program counter, in hex, of instruction causing the memory access.
- A(0x): Hex value in the A register when the access occurred
- LocalA(0x): Hex value that is the physical memory location being accessed on the local nodelet. This column may eventually be changed in a future revision to be a global physical address (View 1).

The TPCs displayed in the memory trace file can be correlated to the output of the object dump of the `.mwx` executable to determine the exact instruction that generated the memory access. The memory trace does not distinguish between single and multi-word memory accesses.

Chapter 8

Simulation Profiling

There are two tools available for profiling applications run by the simulator; a visualization tool that can make a "movie" and a second profiler that will combine and integrate statistics from mutliple output files generated by the simulator to create an HMTL based profile of the application.

8.1 `emuvistool`

The `emuvistool`visualization tool can be very beneficial in identifying bottlenecks and hotspots as it produces an animation of the utilization of major system resources in the system.

Assuming that a `.tqd` file generated by `emusim` is available, this tool can be run by calling `emuvistool <file>.tqd` at the command line. A screenshot of an animation that has been paused is shown in Fig. 8.1.

KNOWN ISSUE: `emuvistool`has not yet been updated to work with structural models that use something other than 8 nodelets per node.

Navigation and manipulation within `emuvistool` consists of the scroll bars, progress bar (horizontal orange bar), and the row of 5 control boxes at the bottom of the window underneath the progress bar. The scroll bars allow the user to see all nodes and nodelets in the system; once the window is wide enough to display a full node, the remaining nodes are underneath it. The progress bar measures how far into the simulation results the tool is. The 5 control boxes consist of <<, <, `Play` or `Pause`, >, and >>. Use of << and >> may occasionally cause the tool to quit working. < and > will step backward or forward from the current simulation time step. `Play` and `Pause` work as one would expect.

Across the top of the `emuvistool` window, there are four text boxes which display the current simulation time (in ns), the number of threads that have been born and died up to the current time, and the number of threads that are currently alive (active in the system).

Each node in the system is divided into three sets of gray-shaded boxes. The top box, labeled the Migration Engine and shown as "ME Total Threads" shows the total number of packets in the Migration Engine queues. We do not show the number of packets in each individual queue as it would make the animation far too crowded. One queue not shown in the current animation is

the number of packets waiting to be processed by the node that have come in from the system interconnect. This queue only occurs in non-Chick system models.

The next set of gray-shaded boxes contains the nodelets. Within the nodelets, the following measurements are taken:

- Sum of Queues: Sum of entries in the following three queues which reside in the Nodelet Queue Manager. This sum must always be less than 512.
- Migr. Queue: Number of packets in the NQM waiting to leave the nodelet.
- Remote Queue: Number of packets in the NQM waiting to be sent to memory.
- Run Queue: Number of threads awaiting a register set (aka context) within a GC.
- Register Sets: Number of register sets in use within the nodelet's GCs. Currently, there are a total of 256 register sets available.

The final set of gray-shaded boxes contains the memory system (NCDram) queues associated with each nodelet; each nodelet is directly connected to only its associated memory system. Within the NCDram systems, `emuvistool` displays the following queues:

- Mem Requests and Mem Returns: the number of transactions moving between the GCs and the Memory Front End
- NQM to MFE: the number of memory requests in the Memory Front End that have come from the NQM
- MFE to NQM: deprecated (always 0), will be removed in a future version
- MFE to DDR: number of transactions accepted by the Memory Front End awaiting processing by the DDR
- DDR: number of transactions currently being "processed" by memory

Each queue is shaded from blue to red as its utilization goes from empty to full. Additionally, each queue maintains a red box that marks the maximum depth seen to the current time of the queue.

8.2 `emusim_profile`

The `emusim_profile` script runs a program within the Emu simulator and generates a variety of useful plots after the execution has completed. This script can be used as a profiler to identify performance issues.

Before using `emusim_profile`, it is recommended to test the program with the simulator to ensure that it runs to completion in a reasonable amount of time, reducing the input size as necessary. The program will need to call `starttiming()` to get the full set of results.

The profiler is invoked in the following manner, passing the profile output directory followed by the benchmark command line:

```
emusim_profile <profile directory> <simulator options> -- mybenchmark.mwx
  --param 1 --param 2
```

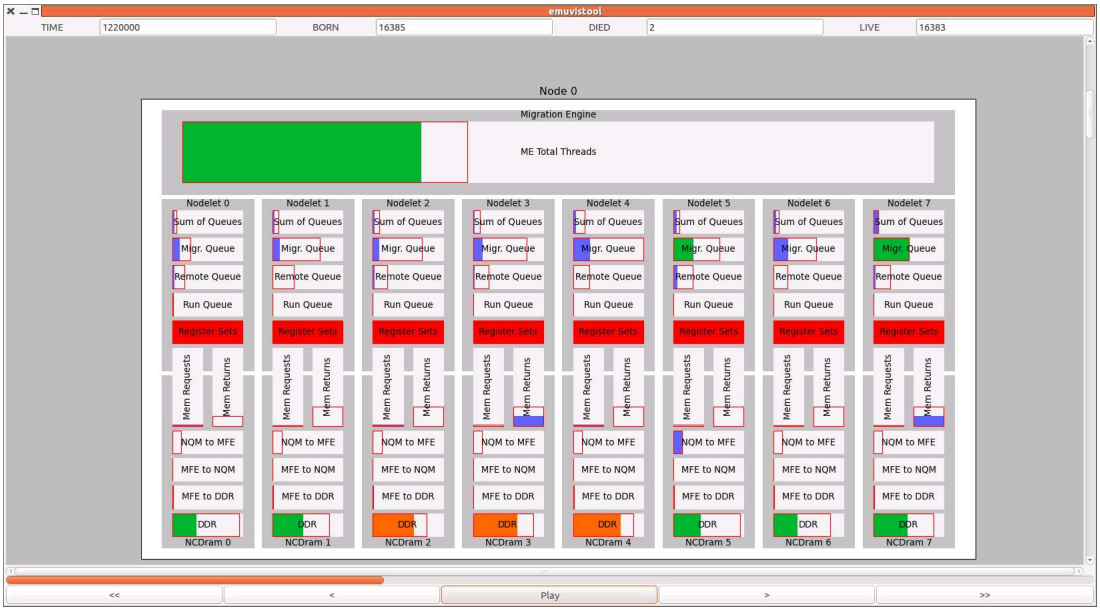
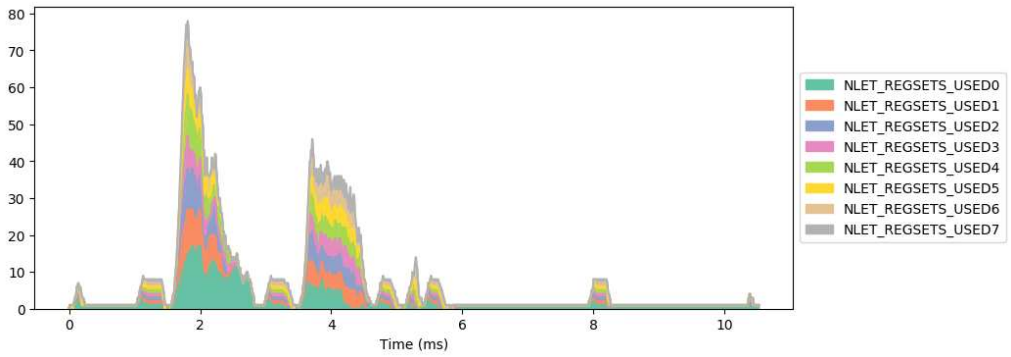


Figure 8.1: Screenshot of `emuvistool`

Several output files will be placed in the profile directory. After the execution completes, you can open the HTML document generated in the profile directory to view the report. Alternatively, you can browse through the profile directory and open each plot image individually.

8.2.1 Active threads per nodelet over time



Description: A stacked area plot with the number of active threads on the Y axis, and simulation time on the X axis. Each colored stripe represents a different nodelet. The width of each stripe represents the number of threads actively running on that nodelet, and the combined height represents the total number of actively running threads on the entire system. This plot is generated using the NLET_REGSETS_USED columns in the TQD output file.

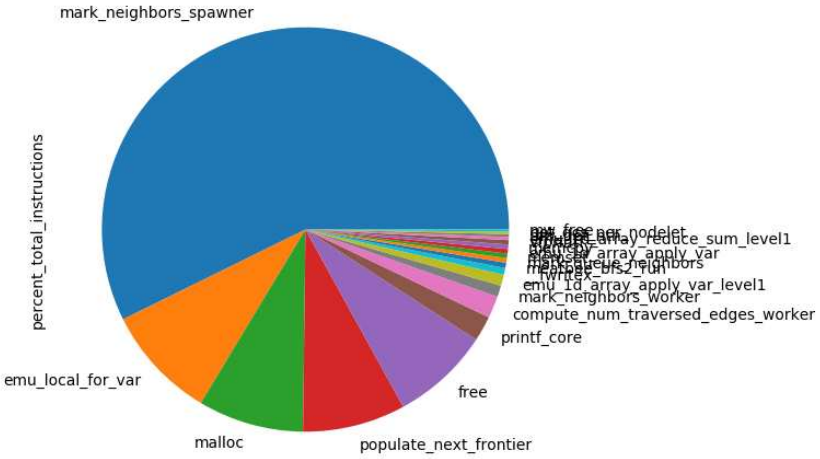
Performance Hints:

- Long flat sections indicate that serial code is dominating the execution. Look for functions

that can be parallelized.

- Spans of time with only one color indicate that one nodelet is doing all the work while other nodelets sit idle. Consider changing the data layout to distribute more evenly across the system.
- A large filled area in which all color stripes are equally sized indicates excellent parallelism and thread balance.

8.2.2 Total instructions executed by function

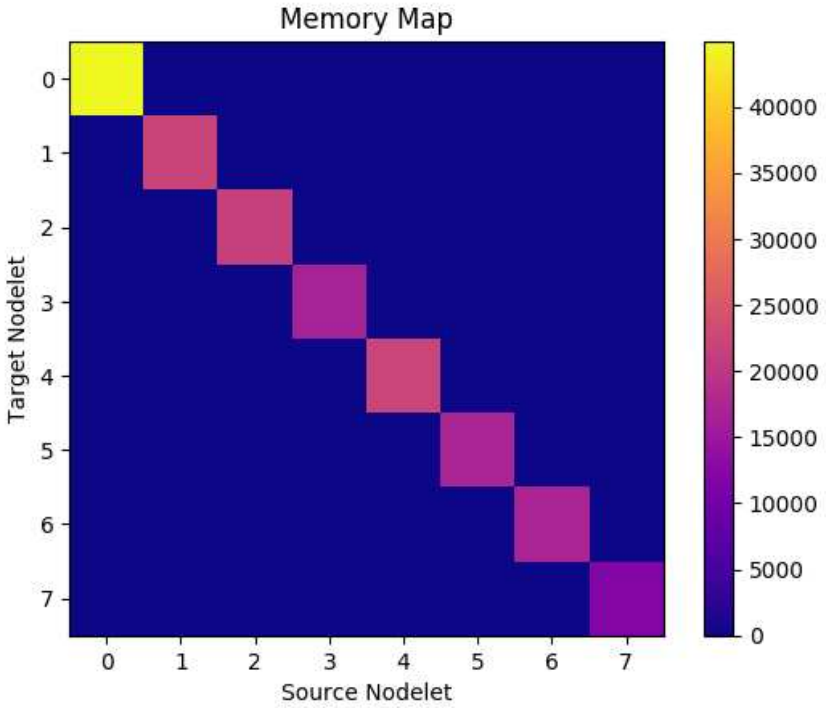


Description: The area of each slice of this pie chart is proportional to the number of instructions executed by each function, summed across all threads throughout the entire program. This plot is generated by aggregating the instruction counts from the UIS file.

Performance Hints:

- The functions with the largest area are the best candidates for optimization.
- Note that this plot counts total instructions executed by all threads, and not execution time per function. Functions that consume a lot of time in serial code may be underrepresented in this plot.

8.2.3 Memory map

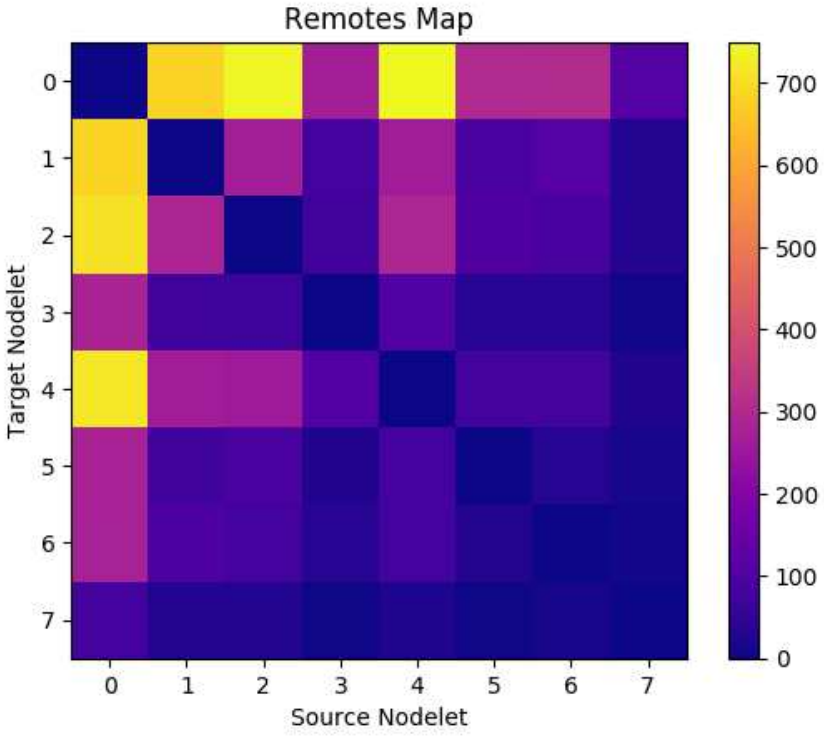


Description: A 2D heat map plotting the total number of local memory reads/writes and migrations between each pair of nodelets. Data for this plot is pulled directly from the MEMORY MAP in the CDC file.

Performance Hints:

- The diagonal cells indicate local memory traffic, which usually dominates. Comparing the brightness of each diagonal cell gives a general idea of work distribution across the entire execution.

8.2.4 Remote map

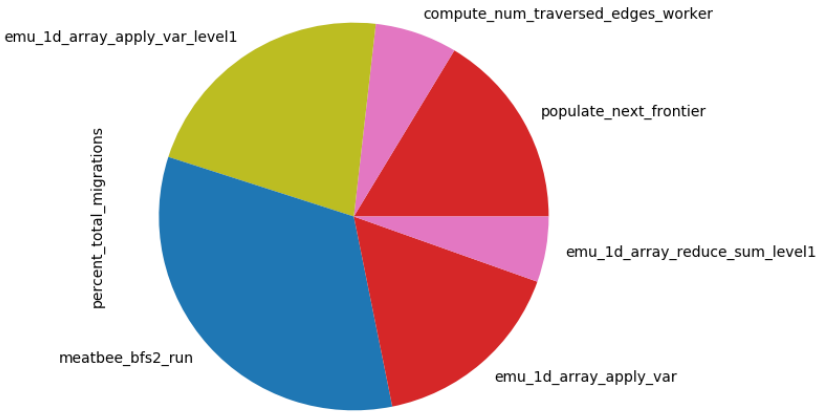


Description: A 2D heat map plotting the total number of remote writes and remote atomic instructions that traveled between each pair of nodelets. Data for this plot is pulled directly from the REMOTES MAP in the CDC file.

Performance Hints:

- Note the scale of the plot to the right. If the overall number of remotes was low, they likely did not affect performance much.

8.2.5 Percent of total migrations grouped by function



Description: The area of each slice of this pie chart is proportional to the number of migrations triggered by each function, summed across all threads throughout the entire program. This plot is generated by aggregating the migration counts from the UIS file.

Performance Hints:

- The functions with the largest area are the best candidates for optimization, especially if a function was not expected to migrate.
- See the migration report below to drill down into which instructions are causing the migrations.

Chapter 9

Configuring the Emu Chick

9.1 Overview

This chapter describes how to start up and configure the Emu1 Chick for either single-node or multi-node execution.

The Emu1 Chick consists of a system controller and up to 8 node boards (also referred to as slots or NCDIMMs), where each node contains 2 Stationary Cores and 8 nodelets. The system can be configured in two ways: single-node or multi-node.

In a single-node configuration, each node board is configured as an independent node with access to its 8 nodelets but not to other nodes in the system. This allows up to number-of-nodes independent users (one on each node) to run programs on the system concurrently.

In a multi-node configuration, a single user runs a program with access to two, four, or eight nodes in the system.

9.2 Node Board Power On/Off

The node boards can be powered on/off using the power button or via the System Controller. Whenever AC power is applied to the system, the System Controller operates from an always-on power supply. The system controller monitors the POWER button and controls power to the node boards based on that input.

To turn on power using the POWER button, simply press and hold the POWER button for one second. This will cause the system controller to begin the power sequence. Alternatively, log into the System Controller as an Administrator and issue the `emu_power_on` command.

To power off the system, execute `emu_power_off` from the system controller.

The helper script `emu_power_cycle` can also be used to properly power cycle any subset of node-boards for the given Chick. This script can be run from the system control board. See `Emu System Command Reference Manual` for more details.

9.3 System Configuration

Once the node boards are powered on, the system can be configured for either single-node or multi-node execution. To configure the system, `ssh` into the system controller and execute `emu_all_start_nodeboard`. See `Emu System Command Reference Manual` for more details. Using `emu_all_start_nodeboard`, the system can also be switched from single-node to multi-node configuration or vice versa **without** power cycling.

Note that, boards configured as single-node can be power cycled individually. See `emu_power_cycle` and `emu_all_start_nodebard` in `Emu System Command Reference Manual`. Boards configured as multi-node must all be power cycled. If you're unable to initialize the system via `emu_all_start_nodeboard`, please see the Troubleshooting Guide on Box in `Customer Support` for troubleshooting any issues with the node boards.

Chapter 10

Emu Chick Execution

10.1 Overview

This chapter describes how to run programs on the Emu1 Chick in either a single-node or multi-node configuration.

10.2 Compiling an Application

Starting with the `emu-18.X` series of toolchain releases, there is not a separate compilation toolchain when targeting hardware instead of the simulator. Section 7.2 and the Emu Software Tool Set Installation Guide contain descriptions of the process and where the appropriate tools should reside.

It is highly recommended to run all programs through the simulator first to verify correctness before running on the hardware.

10.3 Single-node Program Execution

Once the programs have been compiled on the host system, they must be copied via `scp` to the node on which they will be executed.

10.3.1 Launching a Program

The programs are executed via `emu_handler_and_loader` which loads the program, launches the initial thread into the system, and polls the system exception queue to handle system services until a thread exits or an exception occurs. It then checks all the cores to make sure there are no remaining threads and, if there are, it issues a checkpoint to dump the remaining threads. It will print information to log files for each thread that quits, exits, generates an exception, or is checkpointed. It returns the return value from the program, or 0 upon exception.

`emu_handler_and_loader` notes:

- Arguments to the handler are before the executable and command-line arguments are after the `--` delimiter.
- The current directory for any file I/O is the directory where `emu_handler_and_loader` was invoked.
- During execution, log files will be generated in the `/tmp/$(whoami)/emu-scd.log` and the `/tmp/$(whoami)/emu-scd-system.log`. The latter log file is more verbose.
- See `Emu System Command Reference Manual` for more details.

10.3.2 Interrupting Program Execution

If a program appears to be "hanging", the user can force a checkpoint by using `ctrl-C` or by executing `emu_kill_program` on the node. See `Emu System Command Reference Manual` for more details. This should cause any executing threads to be written out to the System Exception Queue (SEQ). The handler will then print the thread state and other information to the log files so that it can be examined.

10.4 Multi-node Program Execution

Once the programs have been compiled on the host system, they must be copied via `scp` to node 0 on the Chick.

10.4.1 Launching a Program

In a multi-node configuration, the programs are executed via `emu_multinode_exec` which copies the program to each node, loads the program into the memory of each node, starts a handler on each node, and launches the initial thread into node 0. The handler on each node handles any system service requests and polls that node's system exception queue until a thread exits or an exception occurs.

Once a thread has exited or an exception has been identified, a completion process is started. During this process, all the cores are checked to make sure there are no remaining threads and, if there are, the remaining threads are checkpointed. Information is printed to `/tmp/$(whoami)/emu-scd-system.log` and `/tmp/$(whoami)/emu-scd.log` on each node for each thread that quits, exits, generates an exception, or is checkpointed on that node. Once all nodes have completed the process, each node's log files are copied to node 0 in the working directory as `mn_exec_{sys, usr}.<PID>.<$HOSTNAME>.log` where `<PID>` is the process ID and `$HOSTNAME` is the hostname of the given node-board. For example, for a process ID of 2376, the log files would be named `mn_exec_sys.2376.n0.log` to `mn_exec_sys.2376.n7.log` and `mn_exec_usr.2376.n0.log` to `mn_exec_usr.2376.n7.log`.

`emu_multinode_exec` notes:

- Arguments to the script are before the executable and command-line arguments are after the `--` delimiter.
- The current directory for any file I/O on node 0 is the directory where `emu_multinode_exec` was invoked.
- The current directory for any file I/O on nodes, other than node 0, is the user's home directory.
- See `Emu System Command Reference Manual` for more details.

10.4.2 Interrupting Program Execution

`emu_kill_program` can be run on any node during multi-node execution in order to kill the running application. See above for `emu_kill_program` usage.

10.5 Debugging

When a thread executes a system call, encounters an exception, or exits, its NMB and Thread Status Word (TSR) are written to the SEQ to be processed by the handler running on that node's SC. On exit or when an exception has been identified, the handler will issue a checkpoint to halt all other threads in the system and write out their TSRs to the SEQ. This information should be available in the log files and can be used to help debug the program. In particular, the program counter (TPC) where an exception occurs and the values of the thread registers at that point in execution can be compared to the verbose output of the simulator to help identify how and why the exception was triggered.

10.5.1 Log File Environment Variables

There is one environment variable that can be set to generate additional debugging information printed to `/tmp/$(whoami)/emu-scd-system.log`.

In order to generate additional messages you may export `EMU_DEBUG_LOG` on the node-board you're launching the program from. To do so, you must run one of the following prior to program execution:

- `export EMU_DEBUG_LOG=debug`: generates debug messages.
- `export EMU_DEBUG_LOG=thread_launcher`: generates additional messages in the thread launcher.
- `export EMU_DEBUG_LOG=handler`: generates additional messages in the handler.
- `export EMU_DEBUG_LOG=all`: generates debug, thread launcher, and handler messages.

10.5.2 NMB and Thread Status Word (TSR)

The format of the information stored to the SEQ (and printed as a part of the debug log) is as follows:

- NMB: Packet header
- A register: Holds the current address
- TCB0 and TCB 1: Thread Control Block (includes the following)
  - TPC: thread program counter
  - Exception code: see exception codes
  - TS: thread state
  - M: migration flag
  - Z: zero flag
  - N: negative flag
  - CB: carry/borrow flag
  - V: overflow flag
  - NaN: not a number
  - Exception cause: see exception codes
  - Internal exception flags

The remaining values are the live registers as specified by the Register Usage Fields (RUF). The many include one or more of the following:

- D register
- D2 register
- A2 register
- E[0] to E[15] registers
- Thread ID

10.5.3 Exception Codes

The exception codes are as follows.

- Quit: 0x0
- SC Runtime Service: 0x1
- GC Runtime Service: 0x2
- SC OS Service: 0x3 (e.g. system calls)
  - E[2] specifies the system call code

  - Up to 7 arguments are placed in E[3..9]
- Arithmetic Exception: 0x4
  - cause = 0: divide-by-0
  - cause = 1: square root on a negative number
- Address Exception: 0x5
  - cause = 0: attempting to do memory reads across nodelets
  - cause = 1: attempting to access memory on a remote node in view 0
- View Exception: 0x5
  - cause = 4: attempting to access memory with an illegal view ID
  - cause = 5: attempting to perform a remote operation with an illegal view ID
  - cause = 6: view ID is not supported
- Alignment Exception: 0x6
  - cause = 0: address is not aligned properly
- Range Exception: 0x6
  - cause = 4: TPC is not within the shared or AID text_base/text_limit
  - cause = 5: memory referred by A register is not within the shared or AID data_base/data_limit
  - cause = 6: migration detected with it has been disabled in the TSR
  - cause = 7: address mapping to a non-existent node is detected
- Illegal Exception: 0x7
  - cause = 0: illegal instruction opcode
  - cause = 1: attempting to use a register not enabled in the RUF
  - cause = 2: illegal MTS operand
  - cause = 3: attempting to execute SWAPA when A2 is not used
  - cause = 4: illegal fence counter: either an attempt to increment beyond 511 or to decrement past 0
- Privilege exception: 0x8
  - cause = 0: attempting to restore a thread with RTS instruction but with a mismatched AID
  - cause = 1: attempting to restart a thread with RST instruction but with a mismatched AID or the executing thread's AID is not 0
- Memory Exception: 0x9
  - cause = 0: memory error detected when an instruction causes a memory read

  - cause = 1: memory error detected with an instruction cache fill and when the instruction word is used
  - cause = 2: memory read data has a mismatched tag
  - cause = 3: cache fill data has a mismatched tag
  - cause = 4: memory read data return was not expected
  - cause = 5: cache fill data was not expected
- Checkpoint/Kill: 0xB
- Return to SC: 0xC
- Single step: 0xD
- Migration Queue: 0xE
- Run Queue: 0xF

10.5.4 Diagnostic Tool

The `emu_all_diagnostic` script on the system controller is very helpful for collecting verbose diagnostic information for both single-node and multi-node execution. Running the `emu_all_diagnostic` script will append a time stamp and verbose diagnostic information to `emu_all_diagnostic.log` in the user's current working directory. See `Emu System Command Reference Manual` for more details. Once `emu_all_diagnostic.log` has been collected, please send the following to emu: emu_all_diagnostic.log from the system controller, the binary and source code of the program you're debugging and which toolchain version was used to generate the binary, the commands used to initialize the system and when the system was initialized, the commands used to run the program and when the program was run and emu-scd.log and emu-scd-system.log from the nodeboards.

For more targeted debugging, the `emu_diagnostic_tool` allows the user to inspect various hardware registers and memory locations. See `Emu System Command Reference Manual` for more details.

For example, assume the program writes a value to a variable at address `0x0180_0000_0000_2ac0`. The `emu_diagnostic_tool` could be used to examine the contents at that location using option 6 as in the following example:

```
Enter the nodelet number (0-7): 0
Enter the offset in hex: 2ac0
Add shared data/text base to memory location (y/n): y

Memory word at 0x0000_0000_0080_2ac0=0x0000_0000_0000_0000
```

The shared data/text base should always be added unless you are examining the values in the SEQ or RQD. Many of the other values are used primarily for debugging hardware issues.

10.6 Running the sc-driver-tests

A set of tests have been included that can be used to verify functionality after system updates, power failures, etc. These tests are installed in `/usr/sc-driver-tests/tests/{singlenode,multinode}`. This test suite can be executed using the program `/usr/sc-driver-tests/bin/emu_loop_test`, which runs the tests in a specified directory and any subdirectories
(e.g. /usr/sc-driver-tests/tests/singlenode). Note that the system should be in single-node configuration for the singlenode tests and multi-node configuration for the multinode tests. See `Emu System Command Reference Manual` for more details.

Long running tests are often executed with `nohup` so that the loop test will continue even if the ssh session is closed. For example, to run 10 iterations of the tests with a timeout of 16 seconds for each test:
`nohup emu_loop_test 16 10 /usr/sc-driver-tests/tests/singlenode &`

For each test in the specified directory, the loop test will:

- run `file.mwx` using either `emu_handler_and_loader`, if the system was initialized for single-node execution, or, `emu_multinode_exec`, if the system was initialized for multi-node execution.
- redirect the handler output to `file.mwx.log`
- compare the return value to `file.status`
- generate a message indicating whether the test passed or failed, based on the return value comparison.

.

This results in a log file for each test along with the file `nohup.out` with the results for all tests and the pass/fail messages.

To evaluate the results:

- `grep PASS nohup.out`: tests that passed
- `grep FAIL nohup.out`: tests that failed

To inspect logs after the script has run to completion or been killed do:
`scp /tmp/$(whoami)/emu_loop_test_nX.tar.gz user@local_machine:/path/`
where `X` is the node id (i.e. X=[0,7], depending on the slot that `emu_loop_test` was run on). NOTE: this tarball is overwritten each time `emu_loop_test` is run and exits normally.

Note that if you use 0 for the number of iterations, you will need to kill the loop test with `kill $(cat /tmp/ilp.pid)`.

Alternatively, use the `emu_batch_test` script from the system controller to start and monitor long running test as well as check whether the tests passed. This script assumes the target nodes have been initialized and will start tests if nothing is already running, attach to and monitor already running tests, and finally copy relevant log files to the user's working directory when the tests stop running or the user enters `ctrl-C`. See `Emu System Command Reference Manual` for more details.

Bibliography