Cray XMT™ Programming Environment User's Guide

S–2479–20

RECORD OF REVISION

S–2479–20 Published May 2011 Supports release 2.0 GA running on Cray XMT compute nodes and on Cray XT 3.1UP02 service nodes. This release uses the System Management Workstation (SMW) version 5.1UP03.

1.5 Published December 2010 Supports release 1.5 running on Cray XMT compute nodes and Cray Linux Environment (CLE) release 2.241A on Cray XT service nodes. This release requires the System Management Workstation (SMW) version 4.0.UP02, which is based on the SLES10 SP3 base operating system.

1.4 Published December 2009 Supports release 1.4 running on Cray XMT compute nodes and Cray Linux Environment (CLE) release 2.241A on Cray XT service nodes. This release requires the System Management Workstation (SMW) version 4.0.UP02, which is based on the SLES10 SP3 base operating system.

1.3 Published March 2009 Supports release 1.3 running on Cray XMT compute nodes and on Cray XT 2.1.50HD service nodes. This release requires the System Management Workstation (SMW) version 3.1.09 that is based on the SLES10 SP1 base operating system.

1.2 Published August 2008 Supports general availability (GA) release 1.2 running on Cray XMT compute nodes and on Cray XT 2.0.49 service nodes. This release uses the System Management Workstation (SMW) version 3.1.04 that is based on the SLES9 SP2 base operating system.

1.1 LA Published March 2008 Supports limited availability (LA) release 1.1.01 running on Cray XMT compute nodes and on Cray XT 2.0 service nodes.

1.0 LA Published August 2007 Draft documentation to support Cray XMT limited-availability (LA) systems.

Changes to this Document

Cray XMT™ Programming Environment User's Guide S–2479–20

This rewrite of *Cray XMT Programming Environment User's Guide* supports the 2.0 release of the Cray XMT operating system and programming environment. For more information see the release announcement that accompanies this release.

Added information

- Two new pragmas: `#pragma mta max` *n* `processors` and `#pragma mta max concurrency` *c*. See Compilation Directives on page 109.
- Additional programming examples.

Revised information

- The snapshot documentation has been revised extensively. See Chapter 6, Managing Lustre I/O with the Snapshot Library on page 67.
- Technical and editorial corrections.

The conceptual content that made up the first chapters of previous versions of this guide have been moved to a new document, *Cray XMT Programming Model*.

Contents

Page

Figures

Introduction [1]

This guide describes the Cray XMT Programming Environment. It includes procedures and examples that show you how to set up your user environment and build and run optimized applications. The intended audience is application programmers and users of the Cray XMT system. For information about debugging your application, see *Cray XMT Debugger Reference Guide*. For information about performance analysis tools that you can use to tune your application, see *Cray XMT Performance Tools User's Guide*.

This chapter presents a general overview of the Cray XMT. Subsequent chapters of this manual cover the details for how to write programs for the Cray XMT.

1.1 The Cray XMT Programming Environment

The Cray XMT Programming Environment (XMT-PE) includes the following:

- Cray XMT compilers for C and C++
- Cray mdb debugger, which is an adaptation of the Free Software Foundation's gdb debugger
- Apprentice2 performance analysis tool

The XMT-PE runs on a Linux operating system on a service node. You write and compile your program on the service partition and launch it from the service partition onto the compute partition.

Setting Up the User Environment [2]

Configuring your user environment on a Cray XMT system is similar to configuring a typical Linux workstation.

2.1 Setting Up a Secure Shell

Cray XMT systems use `ssh` and `ssh-`enabled applications such as `scp` for secure, password-free remote access to the login nodes.

Before you can use the `ssh` commands, you must generate an RSA authentication key. The process for generating the key depends on the authentication method you use. There are two methods of passwordless authentication: with or without a passphrase. Although both methods are described here, you must use the latter method to access the compute nodes through a script or when using a single-system view (SSV) command.

2.1.1 RSA Authentication

You can set up RSA authentication with or without a passphrase.

Procedure 1. Setting up RSA authentication with a passphrase

To enable `ssh` with a passphrase, complete the following steps.

1. Generate the RSA keys by typing the following command and follow the prompts. The program requests you to supply a passphrase.

   ```
   % ssh-keygen -t rsa
   ```

2. Create a `$HOME/.ssh` directory and set permissions so that only the file's owner can access them by typing the following commands:

   ```
   % mkdir $HOME/.ssh
   % chmod 700 $HOME/.ssh
   ```

3. The public key is stored in your `$HOME/.ssh` directory. Copy the key to your home directory on the remote host (or hosts) by typing the following command:

   ```
   % scp $HOME/.ssh/key_filename.pub \
   username@system_name:.ssh/authorized_keys
   ```

4. Connect to the remote host by typing the following commands.

   If you are using a C shell, type:

   ```
   % eval `ssh-agent`
   
   % ssh-add
   ```

   If you are using a `bash` shell, type:

   ```
   $ eval `ssh-agent -s`
   
   $ ssh-add
   ```

5. Enter your passphrase when prompted, followed by:

   ```
   % ssh remote_host_name
   ```

Procedure 2. Using RSA authentication without a passphrase

To enable `ssh` without a passphrase, complete the following steps.

1. Generate the RSA keys by typing the following command:

   ```
   % ssh-keygen -t rsa -N ""
   ```

2. Create a `$HOME/.ssh` directory and set permissions so that only the file's owner can access them by typing the following command:

   ```
   % mkdir $HOME/.ssh
   % chmod 700 $HOME/.ssh
   ```

3. The public key is stored in your `$HOME/.ssh` directory. Copy the key to your home directory on the remote host (or hosts) by typing the following command:

   ```
   % scp $HOME/.ssh/key_filename.pub \
   username@system_name:.ssh/authorized_keys
   ```

   Note: This step is not required if your home directory is shared.

4. Connect to the remote host by typing the following command:

   ```
   % ssh remote_host_name
   ```

2.1.2 Additional Information

For more information about setting up and using a secure shell, see the `ssh`(1), `ssh-keygen`(1), `ssh-agent`(1), `ssh-add`(1), and `scp`(1) man pages.

2.2 Using Modules

The Cray XMT system uses modules in the user environment to support multiple versions of software, such as compilers, and to create integrated software packages. As new versions of the supported software and associated man pages become available, they are added automatically to the Programming Environment, while earlier versions are retained to support legacy applications. By specifying the module to load, you can choose the default version of an application or another version.

The modules for the compilers and associated products are:

- `mta-pe` for the C and C++ compilers. This is the default environment.

Modules also provide a simple mechanism for updating certain environment variables, such as `PATH`, `MANPATH`, and `LD_LIBRARY_PATH`. In general, you should make use of the modules system rather than embedding specific directory paths into your startup files, makefiles, and scripts.

The following subsections describe the information you need to manage your user environment.

2.2.1 Modifying the PATH Variable

Do not reinitialize the system-defined `PATH`. The following example shows how to modify it for a specific purpose (in this case to add `$HOME/bin` to the path).

If you are using a C shell, type:

```
% set path = ($path $HOME/bin)
```

If you are using `bash`, type:

```
$ export $PATH=$PATH:$HOME/bin
```

2.2.2 Software Locations

On a typical Linux system, compilers and other software packages are located in the `/bin` or `/usr/bin` directories. However, on Cray XMT systems these files are in versioned locations under the `/opt` directory.

Cray software is self-contained and is installed as follows:

- Base prefix: `/opt/`*pkgname*`/`*pkgversion*`/`, such as `/opt/mta-pe/default`
- Package environment variables: `/opt/`*pkgname*`/`*pkgversion*`/var`
- Package configurations: `/opt/`*pkgname*`/`*pkgversion*`/etc`

Note: To run a Programming Environment product, specify the command name (and arguments) only; do not enter an explicit path to the Programming Environment product. Likewise, job files and makefiles should not have explicit paths to Programming Environment products embedded in them.

2.2.3 Module Commands

The `mta-pe` modules are loaded by default.

To find out what modules have been loaded, type:

```
% module list
```

To switch from one Programming Environment to another, type:

```
% module swap switch_from_module switch_to_module
```

For example, to switch from the Cray XMT Programming Environment to the GNU Programming Environment, type:

```
% module swap mta-pe PrgEnv-gnu
```

For further information about the module utility, see the `module`(1) and `modulefile`(4) man pages.

Developing an Application [3]

This chapter provides an overview of some Cray XMT functions and describes how to perform some common programming tasks, such as floating-point operations, sorting, dataflow, searching, and I/O.

Before you begin developing your program, you must log in to the login node using `ssh`. You develop, compile, debug, and launch your program from the login node.

Before developing your application, review the data types and keywords that are supported by the Cray XMT compilers. For a list of data types, see Appendix E, Data Types on page 137. For a list of keywords, see Appendix F, Keywords on page 139.

3.1 The Cray XMT Programming Environment

The Cray XMT Programming Environment (XMT-PE) contains the following modules:

- `mta-pe`
- `xmt-tools`
- `mta-man`

The `mta-pe` module contains the C/C++ compilers and some utilities that are useful during the development process. The following table lists the commands for `mta-pe` utilities and provides a brief description.

Table 1. `mta-pe` Utilities

Utility Name	Description
`dis`	Disassembles object code.
`header`	Displays a Cray XMT Executable and Linking File (ELF) header for a specified object, exec, or library file.
`mdb`	Starts debugger for Cray XMT programs.
`nm`	Lists symbols from object files.

The `mta-pe` module also contains support for functions that are specific to the Cray XMT environment. For more information, see Overview of Cray XMT Generic and Intrinsic Functions on page 20.

The `xmt-tools` module contains the tools that you use to run and monitor a program. To run a program, use the `mtarun` command. For more information, see Launching the Application on page 91 or the `mtarun`(1) man page. To monitor the program, use the `mtatop` or `dash` command. For more information, see *Cray XMT System Management* or the `mtatop`(1) man page.

The `mta-man` module contains the man pages for all the utilities, tools, and functions that you find in the XMT-PE.

3.2 Overview of Cray XMT Generic and Intrinsic Functions

The Cray XMT Programming Environment (XMT-PE) supports a number of Cray XMT functions. For a list of these functions, see the `generics`(1) and `mta_intrinsics`(3) man pages. You can refer to the man page for each function for details about how to use that function. Man pages for functions list the names of the header files you must include in your program when using that function.

3.2.1 Generic Functions

The Cray XMT compiler provides a number of generic functions that operate atomically on scalar variables (variables that hold a single value). The generic functions perform `read` and `write`, `purge`, `touch`, and `int_fetch_add` operations on variables. The most common use of the generic functions is to manipulate sync and future variables, but you can also use all of the generic functions, except for the `touch` function, on other types of variables.

Generic functions frequently affect, or have behavior that is dependent upon, the full-empty state of the variable. Because of this, you must know the initial full-empty state of the variable before you allocate it. For sync variables, this state is full if you initialize the variable in the declaration, and empty if you do not initialize the variable. For future variables, the initial state is full. For all other variables, the initial state is full if you initialize the variable in the declaration and undefined if you do not initialize the variable.

You should avoid using generic functions on a variable (other than a sync or future variable) that is less than a word in length. Each 8-byte word of memory is associated with only one full-empty bit. If two or more variables share the same word, they share a single full-empty bit; using a generic function to modify the full-empty state of one of the variables also changes the state of the other variable(s).

You must be careful when using multiword scalars. When you use ordinary language constructs, a `read` or `write` operation of a sync or future multiword variable occurs as if the multiple words are fused and have a single full-empty bit, even when there are other `read` or `write` operations that use the same variable.

When a set of generic functions access a multiword variable simultaneously, the resulting behavior depends on the generic functions that constitute the set. If all the generic functions in the set require the variable to be in either a full or empty state, the functions access the variable in a serialized manner and the user-visible state is consistent. However, if any generic function in the set does not depend on the full-empty state (such as the `purge`, `readxx`, and `writexf` functions), the ability to serialize the set is not guaranteed. If the set is not serialized, generic functions may access words in the variable in a different order, resulting in inconsistencies in one or more of the following: the state of the value returned by one or more of the generics; the memory holding the variable; the data value; or the full-empty bits.

Accessing an individual memory word that is part of a multiword variable (for example, using a cast or a union) could result in inconsistent full-empty states and a data value partially composed of both current and obsolete memory contents. It may also cause a deadlock to occur.

3.2.1.1 Generic Write Functions

The generic write functions write new values to variables, depending upon the full-empty state of the variable. If the type for a value does not match the type for the variable that stores the value, the value is cast to the correct type before being written. For example, in the following:

```
int i;
writeff(&i, 2.0);
```

The value `2.0` (`float`) is cast to `2` (`int`) before being written to `i`.

The Cray XMT compiler recognizes the following generic write functions.

`writeef(&v, value)`

Writes `value` in variable `v` when `v` is in an empty state and sets `v` to a full state. This allows one or more threads waiting for `v` to change to a full state to resume execution. If `v` is in a full state, the write operation is blocked until `v` changes to an empty state. This generic function behaves like a write access to a sync variable.

`writeff(&v, value)`

Writes `value` in variable `v` when `v` is in a full state and leaves `v` in a full state. If `v` is in an empty state, the write is blocked until `v` changes to a full state. This generic function behaves like a write access to a future variable that occurs outside the body of a future statement.

`writexf(&v, value)`

Writes `value` in variable `v` and sets `v` to a full state. This allows one or more threads waiting for `v` to change to a full state to resume execution. This generic function behaves like the write of a return value that occurs at the end of the body of a future statement but is not like a write access to a variable declared with the future qualifier.

`int_fetch_add(&v, i)`

Atomically adds integer `i` to the value at address `v`, stores the sum at v, and returns the original value from `v` (setting `v` to a full state).

Regardless of its type, `i` is cast as an 8-byte integer. Neither parameter can be a multiword object. If `v` is less than the size of a word, the compiler generates a warning diagnostic. If `v` is an empty sync or future variable, the operation is blocked until `v` changes to a full state.

`purge(&v)`

Writes `0`, using the appropriate data type, to variable `v` and sets `v` to an empty state.

For more information, see the `generics`(1) man page.

3.2.1.2 Generic Read Functions

Generic read functions return the value for a variable, depending upon the full-empty state of the variable. When you invoke these functions, the data type of the return value is determined by the type of the first argument in the function call.

The Cray XMT compiler recognizes the following generic read functions.

`readfe(&v)`

Returns the value of variable `v` when `v` is in a full state and sets `v` to an empty state. This allows one or more threads waiting for `v` to change to an empty state to resume execution. If `v` is in an empty state, the read operation is blocked until `v` changes to a full state. This generic function behaves like a read access to a sync variable.

`readff(&v)`

Returns the value of variable `v` when `v` is in a full state and leaves `v` in a full state. If `v` is in an empty state, the read operation is blocked until `v` changes to a full state. This generic function behaves like a read access to a future variable.

`readxx(&v)`

Returns the value of variable `v` but does not interact with the full-empty memory state.

`touch(&v)`

The `touch` function returns the value of future variable *v*, where *v* is associated with a future statement that has been spawned, but whose body may or may not have already begun execution. If the future body that writes *v* has not begun executing, the thread calling `touch` executes the future body. If the future body associated with *v* is currently being executed or has finished executing, `touch(&`*v*`)` acts like a `readff(&v)` function.

You use the `touch` function with future variables that are filled by the execution of code in the body of a future statement. Using Futures in an Application on page 31 Touching a future variable that is in an empty state but not bound to a future results in an execution-time error.

For more information, see the `generics`(1) man page.

3.2.2 Intrinsic Functions

Cray provides intrinsic functions for the Cray XMT system that allow direct access to machine operations from high-level languages. You can find a list of the C intrinsic functions and the machine functions in the `mta_intrinsics`(3) man page. The C intrinsic function names use the name of the machine operation and add a prefix of `MTA_`. So, for example, the machine operation named `FLOAT_ROUND` becomes the C intrinsic function named `MTA_FLOAT_ROUND`. When you use an intrinsic function, it calls its associated machine operation to perform the task at the processor level using assembly language. The result of a machine operation is passed back and becomes the return value of the intrinsic function.

For parameters, when the assembly language version of an instruction names two input registers and an output register, the associated intrinsic function has only two input parameters and returns a result. For example, the machine operation that you use to multiply bit matrices, (`BIT_MAT_OR` *t u v*), uses the intrinsic C function `_int64 MTA_BIT_MAT_OR (_int64` *u*`, _int64` *v*`)` where the *t* parameter in the machine operation becomes the return value for `MTA_BIT_MAT_OR` and the *u* and *v* parameters are the operands. Invoke this intrinsic function by using the following command:

```
t = MTA_BIT_MAT_OR(u, v);
```

For the previous statement, declare *t*, *u*, and *v* as integer variables by using the `_int64` data type. The intrinsic functions use the `_int64` data type for 64-bit signed integers and the `_uint64` data type for 64-bit unsigned integers.

The intrinsic functions that may be most useful are the bit matrix arithmetic functions. For example, if you want to count 1-bits or 0-bits, use the `MTA_BIT_RIGHT_ONE`, `MTA_BIT_LEFT_ONE`, `MTA_BIT_RIGHT_ZERO`, or `MTA_BIT_LEFT_ZERO` intrinsic functions. You can use the `MTA_BIT_OR` and `MTA_BIT_AND` intrinsic functions to perform bitwise `OR` and `AND` operations.

Intrinsic functions support most machine operations that use signed or unsigned integers (`int`), floating-point numbers (`float`), or bit vectors (`bit`) as variables.

If you do not use a constant argument where required, it results in an unresolved reference to the intrinsic function at link time. For example, the intrinsic `MTA_TEST_CC` requires a compile-time constant for its second parameter. If you supply a variable instead, the compiler issues a warning and the invocation is compiled as a call, resulting in a link-time failure.

3.3 Adding Synchronization to an Application

The tasks in this section explain how to add synchronization in your application.

3.3.1 Synchronizing Data Using `int_fetch_add`

Use the `int_fetch_add` generic function to synchronize updates to data that represents shared counters without using locks. This function has the following signature:

`int_fetch_add (`*&v*`,` *i*`)`

The `int_fetch_add` function provides access to the underlying atomic `int_fetch_add` machine operation. This function atomically adds *i* to the value at address *v*, stores the sum at *v*, returns the original value of *v*, and sets the state bit to full. In short, it does the following, as a single atomic operation:

```
t = v; v = v+i;
return t;
```

You can use `int_fetch_add` to identify the last of a group of threads to complete a task, to partition data into groups, or to maintain a stack or queue index.

3.3.2 Avoiding Deadlock

Using sync variables can introduce deadlock into a program if, when the program executes, threads attempt to do more reads than writes to a sync variable. When you are trying to determine how many read operations the program performs, it is important to remember that every reference to a sync variable results in a separate read of that variable, even when the references occur in the same source code statement. For example, in the following cases:

- Your program references a sync variable two or more times on the right side of an assignment statement. For example, if `x$` is a sync variable:

  ```
  sum = x$ + x$;
  ```

- Your program references a sync variable two or more times in a conditional test. For example, if `x$` is a sync variable:

  ```
  if ((x$ >= 10)&&(x$ <= 100)){}
  ```

In these two cases, each reference to `x$` results in a separate read of that variable and requires a separate write to `x$`. The second write to `x$` must be performed by a thread other than the one executing the code in the example. In the first case, it might have been the intention of the programmer to add together two successive values of `x$`. If so, this code presents no problems provided the program contains additional code that executes concurrently with the code in the example and performs the second write to `x$`. In the second case, it is doubtful that the programmer's intention was to compare two different values of `x$`. Also, due to the short-circuiting rules in C and C++, there is no guarantee that the second read will occur. Thus, you could end up with a deadlock whether or not have two writes to `x$`. If you have two writes, but the second read does not occur due to short-circuiting, your code will deadlock due to too many writes. On the other hand, if you have one write, and the second read does occur, your code will deadlock due to too many reads. In both of these cases, if the intention is to read only one value for `x$`, a temporary variable should be used, as in this example:

```
tmpx = x$;
if ((tmpx >= 10) && (tmpx <= 100)){}
```

Deadlock can also occur when two or more concurrent functions access global sync variables in a different order. For example, if `a$` and `b$` are global sync variables, and the function `fnc1` first loads `a$` and then loads `b$`.

```
tmp_a = a$;
tmp_b = b$;
```

In the same program, function `fnc2` first loads `b$` and then loads `a$`.

```
tmp_b = b$;
tmp_a = a$;
```

If the functions run concurrently, then there is a chance of deadlock. If `fnc2` loads `b$` after `fnc1` loads `a$`, but before `fnc1` loads `b$`, then neither function can continue unless a third concurrently running function eventually writes to either `a$` or `b$`. You can avoid this problem by always accessing `a$` and `$b` in the same order each time you use them in functions that may be concurrent.

3.4 Programming Considerations for Floating-point Operations

The base arithmetic for floating-point operations on the Cray XMT uses the IEEE Standard 754 format double precision (64-bit). A 64-bit floating-point number, known as a `Float64` on the Cray XMT, consists of a sign bit, an 11-bit exponent, and 52 bits of fraction. Ordinary numbers (those with a biased exponent not equal to zero or `0x7FF`) have an exponent bias of 1023 (`0x3FF`) and their absolute value can be expressed using the following equation:

```
(1.0 + fraction) << (exponent - 0x3FF)
```

The value is negative if the sign bit is set, positive if it is not set.

A number with a biased exponent of 2047 (0x7FF) is a special floating-point number, known as a SpecialFloat64 on the Cray XMT. If all the fraction bits are zero, the value of the number is plus or minus infinity. Infinity generally occurs in calculations as a result of an overflow or division by zero. For example, 1.0/0.0 is positive infinity, while 1.e300*-1.e300 is negative infinity.

Calculations such as 0.0/0.0 create a result that is called not a number (NaN). Any 64-bit floating-point number with a biased exponent of 0x7FF and a non-zero fraction represents NaN. After NaN enters a computation, it persists through addition, subtraction, multiplication, and division. When a calculation produces a NaN, it indicates an error in your program or data.

In arithmetic comparisons, NaN is not equal to any number, including itself. NaN is neither less than nor greater than any number. In fact, such comparisons raise an exception when one of the numbers being compared is NaN. This implies that the opposite of less than is not greater than but greater than, equal to, or unordered. In this case, unordered allows for the possibility that one of the numbers in the comparison is NaN. The Cray XMT hardware supports comparisons such as less than, equal to, or unordered, and the compilers use these comparisons as necessary when reversing the sense of a test.

There are two representations of zero in the Cray XMT hardware. The number 0x0000000000000000 represents +0.0 while 0x8000000000000000 represents -0.0. Although +0.0 and -0.0 appear to be equal to each other, you can distinguish between them when using them in computations. In particular, 1.0/0.0 equals positive infinity while 1.0/-0.0 equals negative infinity. These values obey computational rules under multiplication, as shown in the following example.

```
0.0*(-1.) = -0.0
(-0.0)*(-1.0) = 0.0
and so on.
```

For any finite nonzero x$, x - x = +0.0. This implies that b - a is not equivalent to -(a - b). For computations with zero, the following rules hold:

```
+0.0 - (+0.0) = +0.0 - (-0.0) = (-0.0) - (-0.0) = +0.0
```
However...
```
-0.0 - (+0.0) = -0.0
```

Underflow in the Cray XMT hardware is gradual in accordance with the IEEE 754 standard. Computations that underflow, producing a rounded result smaller in magnitude than 0x0010000000000000, or about 2.225e-308, do not all flush to zero. If the result has an absolute value greater than or equal to min_denorm, such as 0x0000000000000001, or about 4.94e-324, it is a *subnormal number*. A subnormal number is one with a zero-biased exponent and a nonzero fraction such as 0x0000000000000001 or 0x800FFFFFFFFFFFFF. The absolute value for such a subnormal number is the following:

```
(0.0 + fraction) >> 1022
```

Subnormal numbers are less precise than normalized numbers. The smallest subnormal number, `min_denorm`, has only one significant bit while the largest has 52 significant bits. However, whenever `0.5 <= x/y <= 2.0`, the difference `x - y` is exact, even though it may have less precision than `x` and `y`. This is not true for machines that flush underflow to zero.

The Cray XMT floating-point hardware handles gradual underflow transparently. Unlike many systems, the Cray XMT is not slowed by the presence (or possibility) of subnormal numbers and gradual underflow in a computation.

3.4.1 Differences from IEEE Floating-point Arithmetic

The Cray XMT processors do not have 32-bit floating-point instructions. If you are performing an operation on 32-bit floating-point numbers, you must first use the `MTA_FLOAT_REAL` intrinsic function to convert each 32-bit number in the operation to a 64-bit number. After the operation is complete, you can use the `MTA_REAL_FLOAT` intrinsic function to round the results to 32-bit numbers. This double rounding (first to 64 bits and then to 32 bits) is not the same as a single rounding to 32 bits. For more information about how to use `MTA_FLOAT_REAL` and `MTA_REAL_FLOAT`, see the `mta_intrinsics`(3) man page.

The Cray XMT does not provide you with control over rounding precision for floating-point operations. The level of rounding precision is set on the processor during the manufacturing process.

Traps on the Cray XMT are precise, but operands can be overwritten by the results of an operation performed on the same or a different functional unit. This can make the implementation of post-substitution difficult.

There is no exponent wrapping when an operation enables or takes an overflow or underflow trap. The intent of wrapping is to provide for automatic rescaling when products or quotients are used in subsequent operations. On the Cray XMT, you must use care when rescaling.

The hardware supports fused multiply-add operations that only require a single issue of an instruction. This operation facilitates certain computations by making it easy to extract the lower half of the product of two 64-bit doubles. The problem is that the compiler can evaluate statements such as the following in several different ways, each of which may produce a different result:

```
x = a*b + c*d;
```

The previous statement can be evaluated as either:

```
temp = a*b;
x = temp + c*d;  // For multiply-add operation
```

Or

```
temp = c*d;
x = a*b + temp;  // For multiply-add operation
```

Or

```
temp1 = a*b;
temp2 = c*d;
x = temp1 + temp2;
```

The only way to override the compiler instructions for a particular multiply-add operation is to put each multiply operation on a separate line, as in the third example. You can use the `-no_mul_add` compiler flag to disable multiply-add operations.

Rather than using a multiply-add operation, the compiler may use a common subexpression, as shown in the following example.

```
x = a*b;  //For multiply
y = a*b + c;  //Essentially y = x + c
```

In cases like this, you can use the `#pragma mta single round required` pragma in a C program to indicate to the compiler that it must use a multiply-add operation.

The Cray XMT does not support signaling NaNs. For all data types, the Cray XMT identifies uninitialized floating-point data by throwing poison errors rather than using signaling NaNs. See Appendix A, Error Messages on page 99.

3.4.2 Differences from Cray Floating-point Arithmetic

There are several versions of floating-point arithmetic on Cray systems. Newer Cray systems, such as the Cray XMT, use formats based on IEEE 754. Older Cray systems used a proprietary format that differs from IEEE 754 (and from the Cray XMT implementation of IEEE 754) in significant ways. This older format is known as *Cray floating-point arithmetic*.

Cray floating-point arithmetic uses a 48-bit significand, which has less precision than the 53-bit significand used by the Cray XMT. The significand is the part of a floating-point number that contains its significant digits. Cray floating-point arithmetic has a 15-bit exponent with exponents that contain values between -8192 and 8191. This is a much larger range than the exponents for the Cray XMT that contain values between -1022 and 1023. Cray floating-point operations lack guard digits for subtraction and are known to have certain anomalies in computations.

In general, older Cray code that does not rely on the extra-large exponent range runs without modification on the Cray XMT. Otherwise, some rescaling is required for the Cray XMT. In addition, programs designed for older Cray systems may contain work-around code to handle Cray floating-point anomalies. This code is not necessary on the Cray XMT.

3.4.3 32-bit and 64-bit Implementation of Floating-point Arithmetic

The `double` data type in C uses the format for double-precision (64-bit) arithmetic provided by IEEE Standard 754 guidelines. Cray XMT hardware does not support IEEE Standard 754 extended precision, and all 32-bit arithmetic is done by promotion to 64-bit formats.

Rounding mode on the Cray XMT is controlled on a per-thread basis using mode bits in the stream status word (SSW). A newly created stream inherits the rounding mode of its parent.

Hardware instructions that convert from an `int` or `unsigned int` number to a floating-point number use the same rounding mode as the SSW. You can use the `MTA_FLOAT_UNS` intrinsic function when converting large unsigned integers to a floating-point number. You can use the current rounding mode as the basis for converting a floating-point number to an integer by using the `MTA_FLOAT_ROUND` intrinsic function or use explicit rounding that ignores the mode bits in the SSW by using the `MTA_FLOAT_CEIL`, `MTA_FLOAT_CHOP`, `MTA_FLOAT_FLOOR`, or `MTA_FLOAT_NEAR` intrinsic functions.

Each thread has its own set of floating-point exception flags and traps that can be enabled in its SSW. The normal mode of operation is to run with all floating-point traps disabled.

If you convert a 64-bit floating-point number to a decimal string with at least 17 significant decimal digits and then convert it back to 64-bit floating-point number, the result matches the original. If you convert a decimal string with `n` less than 15 decimal digits to 64-bit floating-point number and then convert it back to `n` decimal digits, the result matches the original string.

Add, subtract, and multiply operations each use one processor instruction on the Cray XMT. Divide operations use eight instructions, and square root operations require ten instructions. There is room in the divide and square-root sequences for other operations, particularly in the memory unit.

3.4.4 Rounding Results of Floating-point Operations

The standard C `math` and C++ `cmath` libraries implement a set of functions that you can use when performing basic mathematical operations such as the `log` function for logarithms. When you use the math library functions on the Cray XMT, these mathematical operations do not necessarily produce correctly rounded results, except for the `sqrt()` function. Function results are generally accurate to within one unit in the last place, but there are exceptions, especially for large arguments. Trigonometric functions do infinitely precise argument reduction.

Numbers are rounded according to the IEEE Standard 754. The default rounding method is overridden when you use the following intrinsic conversion functions: `MTA_FLOAT_CEIL`, `MTA_INT_CHOP`, and `MTA_UNS_FLOOR`.

The current rounding mode for the math library is set to round to the nearest place (`RND_NEAR`). User functions that change the rounding mode must reset it to `RND_NEAR` before calling the math library functions.

Exceptions are handled silently by the math library. No messages are printed, and `errno` is not set by the library. If functions return NaN or infinity, these arguments are propagated silently by the library. Exception flags are raised as appropriate.

3.5 Using Futures in an Application

In your application, a future consists of:

- A future statement that creates a continuation pointing to a series of statements that may be executed by another thread.
- An optional future-qualified variable, known as a *future* variable, that synchronizes execution of other program threads upon completion of the future. The name of the future variable is also the name of the future.
- Parameters used by the spawning thread to pass values to the thread executing the future.
- The future body, which contains the statements pointed to by the continuation that may be executed by another thread. The body may end with a return statement that writes a value to the future variable.

The keyword `future` is used in two ways:

- As a type qualifier for a synchronization variable.

  ```
  future int x$;
  ```

  Upon allocation, the full-empty state of the future variable `x$` is set to full.

- As a statement.

  ```
  future x$(i)
  {
      return printf("i is %d\n", i);
  }
  ```

  In the previous statement, the full-empty state for `x$` is set to empty. The argument `i` is passed in to the future body by value. The stream places the future on a queue that executes the future bodies asynchronously. Any stream can now dequeue the future and execute its body. The return value is stored to `x$`. Finally, the full-empty bit of `x$` is set to full after the return value is stored in `x$`.

Future statements contain the name of a future variable and parameters, a body, and a return statement. The future variable's value is set by the return statement. The future variable is optional; if no future variable is specified, the return statement of the future body supplies no value. For example:

```
int x, y, z
future int i$;

future i$(x, y, z)
{
    /* Some body statements */
    return x*y*z;
}
```

In the previous example, when the computation completes, the return value returns in the future variable `i$`. Subsequent accesses to `i$` are delayed until the future completes.

The use of future variables is limited to scalar data types such as `char`, `int`, `float`, `double`, pointers, and array elements. The body of a future statement may contain any legal statement including function calls and other future statements.

For a recursive operation, you can eliminate some of the overhead of blocking a thread by using the keyword `touch` in your program. This leaves the semantics unchanged, but if the future body has not begun, the calling thread executes it directly.

```
int search_tree(Tree *root, unsigned target) {
  int sum = 0;
  if (root) {
    future int left$;
    future left$(root, target) {
      return search_tree(root->llink, target);
    }
    sum = root->data == target;
    sum += search_tree(root->rlink, target);
    sum += touch(&left$);
  }
  return sum;
}
```

In the previous example, the `touch` operation checks if any thread has started to execute the future body associated with `left$`. If so, it waits for the future body to complete. If not, the thread calling `touch` executes the future body itself.

3.5.1 Improving Performance of Future Statements

When your application is compiled, future statements cause the compiler to create continuations. Continuations are structures that contain pointers to routines that contain the code from the body of the future statement and a list of arguments to pass to that code.

Continuations are normally allocated and deallocated from the heap. However, if the associated future variable is a scalar variable that is located on the stack, the compiler causes the continuation to be placed on the stack. This reduces the overhead associated with allocation and deallocation operations.

The compiler does not do this when there is an array of future variables on the stack because this requires an array of continuations. Continuations can be large so an array of continuations might cause the stack size to become very large. You can force the compiler to place an array of continuations on the stack by using the `stack_continuations` attribute in your application. This may improve the performance of the application.

The attribute has the following syntax:

```
__attribute__((stack_continuations))
```

You can add this attribute to any future-qualified stack-based array declaration in your application.

```
void myFutures()
{
    future int children[10] __attribute__((stack_continuations));
    // ...
}
```

Another way to improve performance is by employing the *autotouch* compilation mode. This compilation mode automatically applies the `touch` generic whenever a future variable is referenced. There are three ways to use `autotouch`:

The `-autotouch` compiler flag enables autotouch for all source modules compiled with the flag.

The pragma directive `mta autotouch` can be applied to a single source module. The `on` option enables automatic touching, the `off` option disables automatic touching, and the `default` option reverts the autotouch mode to the default mode for that source module, which was determined by the compile-line flags.

The gcc-style attribute `future int foo$ __attribute__ ((autotouch (on|off)));` allows you to change the autotouch mode on a per-variable basis. For example, `future int foo$ __attribute__ ((autotouch (on)))`. The `on` option enables `autotouch` for all references to this variable, regardless of the current command-line flags or pragmas. Similarly, the `off` option disables `autotouch` for all references to the variable. This attribute generates a warning if it is applied to a variable without the `future` qualifier.

3.5.2 Anonymous futures

Often, a concurrent computation does not have a return value. An example of such a concurrent computation is an I/O statement or a modification of global values. You can express such a computation using an *anonymous future*. An anonymous future has no name or return statement. If the anonymous future does not access a synchronized variable referenced by the main computation, there will be no dependence between the future and the main computation. If a future does not create a dependence, the future may not execute. An anonymous future does not need to execute or finish for a program to terminate normally.

3.6 Testing Expressions Using Condition Codes

When you use arithmetic expressions in your code, you can test the expressions by using the `MTA_TEST_CC` intrinsic function. This function returns condition codes that identify problems in the expression. It uses the following prototype:

`MTA_TEST_CC(`*expression*`,` *condition-mask*`)`

`MTA_TEST_CC` evaluates the *expression* and generates a condition code. If the resulting condition code is a member of the set of condition values in *condition-mask*, `true` is returned; otherwise, `false` is returned.

The *expression* can be a scalar variable, a single arithmetic operation, or a machine intrinsic. If you use a scalar variable, you must assign a value to it in the statement immediately preceding the call to `MTA_TEST_CC`. In `MTA_TEST_CC`, you test the operation on the right side of the assignment statement. The *condition-mask* should evaluate to a compile-time constant. The condition codes and possible values for the *condition-mask* are listed in Appendix D, Condition Codes on page 133.

Example 1. Testing a shift-left operation for a carried number

`MTA_TEST_CC` allows branching based on any of the condition codes produced by the machine intrinsics. For example, consider the problem of testing to see if one of the upper 32 bits in an integer is set. One approach is to use the `MTA_SHIFT_LEFT` intrinsic function, which generates a carried number if a 1 bit is shifted out. When using `MTA_SHIFT_LEFT`, you can use `MTA_TEST_CC` with the `IF_CY` condition mask to check for a carried number, as shown in the following example.

```
enum{IF_CY = 16+32+64+128};
if(MTA_TEST_CC(MTA_SHIFT_LEFT(i, 32), IF_CY))
{
  printf("One of the upper 32 bits was set\n");
}
```

In the previous example, the compiler would emit a `SHIFT_LEFT_IMM_TEST` operation, followed by a conditional branch on carry.

Example 2. Retrieving a condition code and result of a previous operation

It is also possible to test the condition code generated by some earlier operation, allowing you to make use of both the condition code and the result of the operation. In the following example, `MTA_TEST_CC` is used to test whether there was a carry generated by `MTA_BIT_LEFT_ZEROS`. `MTA_BIT_LEFT_ZEROS` returns the number of consecutive `0` bits on the left end of the word.

```
enum{IF_CY = 16+32+64+128};

const int j = MTA_BIT_LEFT_ZEROS(i);
if(MTA_TEST_CC(j, IF_CY))
{
  printf("i was zero\n");
}

else
{
  printf("i had %d significant zeros\n", j);
}
```

Example 3. Retrieving a condition code set by a previous operation

The operation that sets the condition code does not need to be an intrinsic function. The condition code is usually set by an ordinary addition or multiplication operation, such as the following.

```
enum{IF_CY = 16+32+64+128};

const int k = i + j;
if(MTA_TEST_CC(k, IF_CY))
{
  printf("carry generated\n");
}
```

If the expression is more complex, the condition code is only available from the last operation. For example, the expression in the following example requires two adds but only the second add affects the condition code. Because the compiler can evaluate this code in three different ways, it may not yield the correct result.

```
enum{IF_CY = 16 + 32 + 64 + 128};

const int m = i + j + k;
if(MTA_TEST_CC(m, IF_CY))
{
  printf("carry generated\n");
}
```

3.7 File I/O

The Cray XMT performs I/O to a RAM-based file system (RAMFS) and a network file system (NFS). Neither the RAMFS nor the NFS are high-speed file systems, therefore, any data over 2 gigabytes in size must to be written to a high-speed file system, such as Lustre. You can use the NFS for small amounts of data, such as user files.

During the system reboot, all data is lost from the RAMFS because it is not written to disk. Any data that you need to retain across system boots must be written to the Lustre file system prior to rebooting the system. The XMT-PE provides snapshot functions that you can use to move data between the service nodes and the Cray XMT compute nodes. Once the data is on the service nodes, you can use standard Cray XT commands to move data to the Lustre file system.

The underlying details of the file system are abstracted behind UNIX library calls that you can add to your program to perform I/O. The Cray XMT system provides some support for concurrent I/O to multiple files, but you must provide explicit access control for concurrent I/O to a single file.

The following sections discuss standard language-supported forms of I/O as well as I/O using the low-level UNIX I/O functions. Each section discusses the semantics, particularly parallelism, and performance possibilities.

3.7.1 Language-level I/O

In serial code, the standard I/O functions behave as specified in the ANSI C standard. In parallel code, all calls to the standard I/O package are executed atomically. Atomic execution means that while one call is executing, no other call can interfere with what the first is doing. Each call appears to run from beginning to end without interruption.

```
#pragma mta assert parallel
for (i = 0; i &lt; n; i++)
{
    fprintf(f,"this is iteration %d\n", i);
}
```

The previous code asserts that the loop is parallel. In general, it is not safe for the compiler to parallelize a loop that contains procedure calls, especially calls to I/O functions. The assertion indicates to the compiler that, in this case, it is safe to parallelize the loop. The atomicity guarantee ensures that each line written to `f` by this loop is of the form that follows:

```
this is iteration i
```

Two lines are never mixed together, so the following never occurs:

```
this is this is iteration j
```

The actual sequence of lines is random because the different iterations are all executed in parallel. However, for a sequence of calls such as the following:

```
#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    fprintf(f,"this is ");
    fprintf(f,"iteration %d\n", i);
}
```

The output may look like the following, because only the individual calls to `fprintf` are atomic:

```
this is iteration i
this is this is iteration k
iteration j
```

Example 4. Calling standard I/O functions from parallel code

To avoid the previous problem, you can combine the two calls to `fprintf` or add some sort of explicit synchronization. For example:

```
sync int flag$ = 1;

#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    int j = flag$; // lock
    fprintf(f,"this is ");
    fprintf(f,"iteration %d\n", i);
    flag$ = j;     // unlock
}
```

The previous code manipulates the sync variable `flag$` to create an atomic section that contains two calls to `fprintf`. The actual value loaded from and stored to `flag$` is not important because the code uses `flag$` as a lock.

Example 5. Calling record-oriented I/O functions from parallel code

For record-oriented I/O, such as that which occurs when using a combination of `fseek` together with `fread` or `fwrite`, you can use explicit synchronization to ensure correct behavior, such as in the following code example:

```
sync int flag$ = 1;

#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    Buf buffer;
    int j = flag$; // lock
    fseek(f, i*sizeof(Buf), SEEK_SET);
    fread(buffer, sizeof(Buf), 1, f);
    flag$ = j;     // unlock
                   // Work with buffer
}
```

In the previous code, `flag$` controls access to file `f`, ensuring that the combination of `fseek` and `fread` are executed atomically. In this case, you use `SEEK_SET` because the `SEEK_CUR` (positioning relative to the current position) is not useful in a parallel context.

Example 6. Preventing racing when calling I/O functions

You use a similar technique when using `ferror` with another call to ensure that any error detected by the `ferror` call was not caused by a racing `read` or `write` call from a different thread. For example, in the following code, calls to several I/O functions are grouped together so that they are all executed atomically.

```
sync int flag$ = 1;

#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    Buf buffer;
    int err;
    int j = flag$; // lock
    fseek(f, i*sizeof(Buf), SEEK_SET);
    fread(buffer, sizeof(Buf), 1, f);
    err = ferror(f);
    flag$ = j;     // unlock
    if (!err)
    {
        // Work with buffer
    }
}
```

In the previous code, the result of the call to `ferror` is saved to a variable (`err`) for later testing.

The same considerations apply when using futures or more complex loops, perhaps with the I/O hidden within a nest of procedure calls. Single calls always execute atomically. However, when a sequence of calls pertaining to a single file must be executed atomically, you must manage the sequence of calls explicitly.

Internally, the `stdio` library enforces locking for each `FILE` object (`FILE` is a data type defined in `stdio.h`). This causes output to a number of different files can proceed in parallel, but output to a single file is serialized. Similarly, you can use `sprintf` and `sscanf` independently of calls to other functions because these functions do not use a `FILE` object. For example, for the loop in the following example, every iteration refers to a different `FILE` object, so each call to `fprintf` can run without interfering with files used by another iteration.

```
#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    fprintf(f[i],"this is iteration %d\n", i);
}
```

If many parallel calls refer to the same file, locking forces a serial execution order. For example, in the following code, it makes little sense to run the loop in parallel because the calls to `fprintf` are serialized by the lock on the `FILE` object referred to by `g`. However, the interpretation of the format string is controlled by the lock.

```
#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    fprintf(g,"this is iteration %d\n", i);
}
```

If the loop contains significant computations, such as in the following example, you may want to parallelize the loop.

```
#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    int j = expensive_function(i);
    fprintf(g,"f(%d) = %d\n", i, j);
}
```

You cannot use the `stdio` functions to support concurrent file access. For example, consider the following code:

```
#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    Buf buffer;
    FILE *f = fopen(file_name, "r");
    fseek(f, i*sizeof(Buf), SEEK_SET);
    fread(buffer, sizeof(Buf), 1, f);
    fclose(f);
}
```

There are two problems in this example:

- If *n* is large, the system cannot support so many open files.
- The file position (set by `fseek`) is shared among all open versions of the file, so races may occur.

3.7.2 System-level I/O

There are a number of low-level functions provided by the operating system to support more flexible and efficient I/O. However, you should avoid accessing a given file using both the high-level language-dependent methods and the low-level functions. The high-level functions use buffering that may interact with the low-level functions in unpredictable ways.

Example 7. Calling UNIX I/O functions from parallel code

In serial code, the low-level UNIX functions behave as specified by the Posix standard. In parallel code, all calls are executed atomically. In this case, you must explicitly manage access to a particular file by a sequence of calls, to prevent races. For example:

```
#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    char line[80];
    int len = sprintf(line, "this is iteration %d\n", i);
    write(fd, line, len);
}
```

The previous code asserts that the loop is parallel. In general, it is not safe for the compiler to parallelize a loop that contains procedure calls, especially calls to I/O functions. The assertion tells the compiler that, in this case, it is safe to parallelize the loop. The atomicity guarantee ensures that each line written to `fd` by this loop is of the form that follows:

```
this is iteration i
```

Two lines are never mixed together, so the following never occurs:

```
this is this is iteration j
```

The actual sequence of lines is random because the different iterations are all executed in parallel. However, for a sequence of calls such as the following:

```
char part1[80];
int len1 = sprintf(part1, "this is iteration ");

#pragma mta assert parallel
for (i = 0; i < n; i++)
{
    char part2[80];
    int len2 = sprintf(part2,"%d\n", i);
    write(fd, part1, len1);
    write(fd, part2, len2);
}
```

The output may look like the following, because only the individual calls to `write` are atomic:

```
this is iteration this is iteration i
k
this is iteration j
```

Example 8. Using synchronization with UNIX I/O functions

To correct this problem, you can either rewrite the code in the style of the first example or add some sort of explicit synchronization, as shown in the following example.

```
sync int flag$ = 1;
char part1[80];
int len1 = sprintf(part1, "this is iteration ");

#pragma mta assert parallel

for (i = 0; i < n; i++)
{
    char part2[80];
    int len2 = sprintf(part2, "%d\n", i);
    int j = flag$; // lock
    write(fd, part1, len1);
    write(fd, part2, len2);
    flag$ = j;     // unlock
}
```

The previous code manipulates the sync variable `flag$` to create an atomic section that contains two calls to `write`. The actual value loaded from and stored to `flag$` is not important because the code uses `flag$` as a lock.

Example 9. Using synchronization with UNIX record-oriented I/O functions

For record-oriented I/O, you can use explicit synchronization to ensure the correct behavior by using a combination of `lseek` together with a `read` or `write` operation, such as in the following code example.

```
sync int flag$ = 1;
#pragma mta assert parallel
    for (i = 0; i < n; i++)
    {
        Buf buffer;
        int j = flag$; // lock
        lseek(fd, i*sizeof(Buf), SEEK_SET);
        read(fd, buffer, sizeof(Buf));
        flag$ = j;     // unlock
        //Work with buffer
}
```

In the previous code, `flag$` controls access to file `fd`, ensuring that the combination of `lseek` and `read` are executed atomically. In this case, you use `SEEK_SET` because `SEEK_CUR` is not useful in a parallel context.

The same considerations apply when using futures or more complex loops, perhaps with the I/O hidden within a nest of procedure calls. Single calls always execute atomically. However, when a sequence of calls pertaining to a single file must be executed atomically, you must manage the sequence explicitly.

Internally, the UNIX library enforces locking for each file descriptor so that output to multiple files can occur in parallel, but output to a single file occurs serially. For example, in the following loop, every iteration refers to a different file descriptor, so each call to write runs without interfering with other calls.

```
#pragma mta assert parallel

for (i = 0; i < n; i++)
{
    char line[80];
    int len = sprintf(line, "this is iteration %d\n", i);
    write(fd[i], line, len);
}
```

If many parallel calls refer to the same file, locking forces a serial execution order. For example, in the following code, it makes little sense to run the loop in parallel because calls to `write` are serialized by the lock on the file descriptor `fd`.

```
#pragma mta assert parallel

for (i = 0; i < n; i++)
{
    char line[80];
    int len = sprintf(line, "this is iteration %d\n", i);
    write(fd, line, len);
}
```

If the loop contains a significant computation, such as in the following example, you may want to parallelize the loop.

```
#pragma mta assert parallel

for (i = 0; i < n; i++)
{
    char line[80];
    int j = expensive_function(i);
    int len = sprintf(line, "f(%d) = %d\n"`, i, j);
    write(fd, line, len);
}
```

You cannot use the other low-level UNIX I/O functions to support concurrent access to a single file.

3.8 Porting Programs to the Cray XMT

Use the following information when you prepare to port C and C++ programs to the Cray XMT platform.

64-bit issues

The following list describes important 64-bit issues.

Alignment

On the Cray XMT, many data types are aligned on 8-byte boundaries that other machines align on 2- or 4-byte boundaries. The Cray XMT uses the following alignments:

- 1-byte boundaries: `char`, `__int8`
- 2-byte boundaries: `__short16`, `__int16`
- 4-byte boundaries: `short`, `__short32`, `float`, `__int32`
- 8-byte boundaries: `int`, `long`, `double`, `long double`, and all pointers

Bit shift and bit mask

Be careful when using bit shift or bit mask to extract fields of a value. Problems can occur if the size of the value type on the Cray XMT is different from the size on the machine you are porting from.

Conversion of floating-point data types

In C and C++ programs, floating-point data types are converted to doubles in all expressions. This conversion is also made on the Cray XMT, except for `long doubles` (16-bytes long) which are not converted to `doubles` (8 bytes long).

Unions

Unions sometimes contain assumptions about the relative sizes of data types. For example, on some machines, two `int` values use the same number of bytes as a `long`. However, on the Cray XMT, `int` and `long` values use the same number of bytes. When in doubt, use the `sizeof` operator to determine the size of data types.

Posix compliance

The following list describes issues related to IEEE Portable Operating System Interface (Posix) compliance.

`errno.h` `errno` is thread-specific and not a global variable. Files that use `errno` in the same way that it is used by library calls such as `perror` must include `errno.h`. This is required by ANSI and Posix, but most systems do not comply with this convention. On the Cray XMT, each thread has its own value of `errno`, so you must include `errno.h` for correct behavior.

`time.h` One goal of the Cray XMT is to support a Posix-compliant application programming interface. As a result, when you port non-Posix programs, you may have to change the header files that are included. For example, you may need to include `time.h` instead of, or in addition to, `sys/time.h`.

Executable formats

On the Cray XMT, executable programs are in `ELF` format instead of `a.out` format. Therefore, you should replace `a.out.h` in your programs with `elf64.h`. Another characteristic of the `ELF` format is that uninitialized and initialized global variables are both mixed in memory.

Miscellaneous issues

The following list describes important miscellaneous issues.

`printf` and `$`

Different implementations of `printf` have different ways of interpreting `$`. The implementation of `printf` on the Cray XMT does not have a special interpretation.

C and C++ structure passing

Structures cannot be passed by value from C to C++.

`mmap` `mmap` is based on file data-block size. The data-block size for a Cray XMT file is different from that on BSD 4.4 UNIX. Although you can use `mmap`, the `mmap_fsblk` system call provides richer semantics.

Cray XMT keywords

You can disable Cray XMT specific keywords (for example, `sync` and `future`) by using the compiler flag `-no_mta_ext`. When this flag is not used, the C compiler for the Cray XMT reserves all keywords—even standard C++ keywords such as `new`, `try`, `throw`, and `catch`.

Preprocessor directives

The following directives are supported on the Cray XMT:

```
#define
#elif
#else
#endif
#error
#ident
#if
#ifdef
#ifndef
#include
#line
#undef
#pragma
#pragma fenv_access
#pragma noalias
#pragma once
```

3.9 Debugging the Program

After completing your program, refer to the *Cray XMT Debugger Reference Guide* for debugging information.

Shared Memory Between Processes [4]

You can share memory between multiple programs by creating a shared memory region using the `mmap` system call.

4.1 Mapping a Memory Region for Data Sharing

A shared memory region is identified by a file name. Before your applications can use shared memory, you must create an empty readable, writable file and run `mmap` to map a memory region to use for shared memory. When you run `mmap`, it allocates the specified amount of physical memory and maps it into the caller's address space. Other programs may share the same memory region by specifying the same file name. A process may use the `unmap` system call to unmap the shared memory region.

Example 10. Mapping memory to share among multiple processes

The following example demonstrates how to create a file and map it to a memory location.

```
#include <sys/types.h>
#include <sys/mman.h>
#include <stdio.h>
#include <fcntl.h>

#define SHARED_SIZE (256*1024*1024)

int main(int argc, char *argv[])
{
  int fd = open(argv[1], O_RDWR|O_CREAT);
  if (fd==-1) {
    perror(argv[1]);
    return 1;
  }

  caddr_t data = mmap(0, SHARED_SIZE, PROT_READ|PROT_WRITE,
        MAP_SHARED|MAP_ANON, fd, 0);
  if (data == MAP_FAILED) {
    perror("mmap");
    return 1;
  }
  unsigned *words = (unsigned*)data;
  // words now points to a shared memory segment
}
```

In the previous example, a new readable and writable file is created by using the open system call with the O_RDWR and O_CREAT flags. The fd file descriptor is allocated and refers to the file. The fd is specified as an argument to the mmap system call and identifies the memory region. SHARED_SIZE specifies the size of the memory region to allocate and map into the caller's address space. PROT_READ | PROT_WRITE specifies that the caller has both read and write permissions to the memory. MAP_SHARED specifies that this is a shared memory region. MAP_ANON specifies that the operating system should allocate physical memory that is not backed up to a file. The mmap system call returns a pointer to the starting virtual address at which the memory was mapped. The data in the memory region is initialized to zeros and the memory state is initialized to full.

The physical memory associated with a shared memory region is normally freed when the last process that was sharing the memory unmaps the memory from its address space. The memory is unmapped either by an explicit call to the unmap system call or automatically upon termination of the process. The persist_rememd function causes the remember daemon to create a mapping to the shared memory region. This preserves the shared memory region even after all other user processes have unmapped the region. The data is preserved only until the system reboots, at which time all data that was in the shared memory region is lost. The persist_rememd function will remember the file name and size of the memory region across reboots and will automatically reallocate the shared memory region upon reboot; the data in the shared memory region is initialized to contain zeros and the state is initialized to full. For more information, see the rememd(8) man page.

Additionally, programs that use synchronization must add calls to the mta_lock_thread and mta_set_thread runtime functions, as shown in the following example.

```
mta_lock_thread(); //Set retry > 0
mta_set_thread_datablocked_retry(MAXINT); //Sets retries = INF
```

The mta_lock_thread function locks a thread to its stream so that the thread does not block and release the stream when it takes a retry limit exception. The mta_set_thread function sets the retry limit to the maximum value. The result of calling these two functions is to cause a thread to spin if a sync-qualified or future-qualified variable is not in the appropriate state for a given memory access, until the thread gains access to the shared data. Spinning is the act of checking the full-empty state repeatedly until the full-empty state changes to the state that the memory operation needs to perform its operation.

This is necessary when synchronization operations are performed between multiple separate processes. Threads that are blocked can only be unblocked by threads within the same process because blocking and unblocking requires access to the runtime internal data structures that are only accessible within the process to which the thread belongs.

For more information, see the mmap(2) man page.

4.2 Persisting Shared Memory

The remember daemon `rememd` retains information about shared memory so that programs preserve shared memory throughout the life cycle of the process. Shared memory is allocated by calling `mmap` with the `MMAP_ANON` and `MMAP_SHARED` flags and a valid file descriptor.

When the `rememd` daemon is first started, it reads in all the records from its maps file and calls `mmap` to map the specified memory into its virtual address space. The daemon does not repopulate the memory; it only allocates it and retains a reference. `rememd` does not attempt to map the same memory segment twice. Once it is mapped, `rememd` increments an internal reference count on subsequent remember requests.

Calling `rememd` does not guarantee that the memory is reclaimed as free. If another program is retaining a reference to the memory, it remains allocated. If multiple requests are made to remember the same segment, `rememd` decrements its internal counter for each forget request until the counter is 0 (zero), at which point, it calls `munmap`.

By holding memory references, the `rememd` daemon allows the memory to outlast one or more processes that might want to use the memory.

Programs that wish to make use of the functionality offered by `rememd` are required to link with the `libremem` library. When a method is called, a remote procedure call is made from `rememd` using UNIX domain sockets. The path to use to communicate with the daemon is specified in the configuration file found at `/etc/rememd.conf` or in the path specified by the environment variable `REMEMD_CONFIG_PATH`.

Use the following functions to call `rememd` from a program.

`persist_remember`

Causes the `rememd` daemon to call `mmap` to map the shared memory into its virtual address space and write a record of it to disk. If `rememd` has already mapped this segment, its reference count is incremented instead. This function returns 0 on success, and *errno* on failure.

`persist_mmap_size`

Causes the `rememd` daemon to return the size of the shared memory mapped into its virtual address space. This function returns the size, in bytes, of the memory region on success, and 0 if the region is not found. When an error occurs, *errno* is set and `-1` is returned.

`persist_forget`

Causes the `rememd` daemon to decrement the specified segment's reference count. If the reference count is zero, the `rememd` daemon calls `munmap` to unmap the shared memory from its virtual address space and remove the record of it from disk. This function returns 0 on success, and the *errno* on failure.

The following example shows how to persist memory in your program.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/file.h>
#include <remem/persistmem.h>

const char *remember_path = "/tmp/my_data_handle";

void run_computation(caddr_t addr, size_t len, int ret);

int main(int argc, char **argv) {
 caddr_t mmap_addr = 0;
 size_t mmap_len = 4096;
 int fd = -1;

 // find out if memory is mapped
 ssize_t ret = persist_mmap_size(remember_path);
 if (-1 == ret) { // -1 indicates there was an error
  printf("Unexpected error from libremem: %s\n", strerror(errno));
  exit(1);
 } else if (0 == ret) { // 0 indicates the memory has not been mapped yet
  fd = open(remember_path, O_CREAT | O_RDWR, 0600);
  if (-1 == fd) {
   printf("Unexpected error opening remember_path: %s\n", strerror(errno));
   exit(1);
  }
  mmap_addr = mmap(0, mmap_len, PROT_WRITE, MAP_ANON | MAP_SHARED, fd, 0);
  if (MAP_FAILED == mmap_addr) {
   printf("Unexpected error calling mmap: %s\n", strerror(errno));
   close(fd);
   exit(1);
  }
  int remember_ret = persist_remember(remember_path, mmap_len);
  if(0 != remember_ret) {
   printf("Unexpected error calling persist_remember: %s\n", strerror(remember_ret));
   close(fd);
   munmap(mmap_addr, mmap_len);
   exit(1);
  }
 } else { // if ret is not -1 or 0, then it's the length of the mapped segment
  fd = open(remember_path, O_CREAT | O_RDWR, 0600);
  if (-1 == fd) {
   printf("Unexpected error opening remember_path: %s\n", strerror(errno));
   exit(1);
  }
  mmap_len = ret;
  mmap_addr = mmap(0, mmap_len, PROT_WRITE, MAP_ANON | MAP_SHARED, fd, 0);
  if (MAP_FAILED == mmap_addr) {
   printf("Unexpected error calling mmap: %s\n", strerror(errno));
   close(fd);
   exit(1);
  }
 }

 run_computation(mmap_addr, mmap_len, ret);

 int forget_ret = persist_forget(remember_path, false);
 if(0 != forget_ret) {
  printf("Unexpected error calling persist_remember: %s\n", strerror(forget_ret));
 }
 if(0 != munmap(mmap_addr, mmap_len)) {
  printf("Unexpected error calling munmap: %s\n", strerror(errno));
 }
 if(0 != close(fd)) {
  printf("Unexpected error calling close: %s\n", strerror(errno));
 }

 return 0;
}
```

Developing LUC Applications [5]

This chapter describes how to use the LUC library in your application. The following tasks are discussed:

- Constructing a client
- Constructing a server
- Making remote procedure calls

5.1 Programming Considerations for LUC Applications

- On the service (Linux) nodes, `int` is defined as 4 bytes. On the MTK compute nodes `int` is defined as 8 bytes. To avoid potential issues, programmers should use types that have explicit sizes, for example `int64_t`.
- There is a limit of 256 MB of data that can be transferred in a single call. This applies to both input and output buffers.
- Linux and MTK have different native byte ordering, Linux is little endian (4 bytes) and MTK is big endian (8 bytes). LUC does not byte-swap or otherwise interpret the application's input and output data so you must add byte-swapping into your application that will perform byte swapping for data transfers between the server and client applications.
- The number of threads that are assigned to an object during a call to `startService` should be determined by the length of time function calls made by that object are expected to take. Allocate enough threads so that an operation is never stalled while waiting for a thread to become available.
- The Linux version of the library can only honor a *requestedPid* value other than `PTL_PID_ANY` for the first endpoint in an application process. The exception is that subsequent values of *requestedPid* may be honored if they are equal to the *requestedPid* of the first endpoint for a process.

5.2 Creating and Using a LUC Client

Use the following procedure to create a client object.

Procedure 3. Creating and using a LUC client object

1. Include the header file `<luc/luc_exported.h>`. This header file includes all of the definitions required for both the client and server endpoints, including the `LucEndpoint` class definition, configuration variables, and external function prototype definitions.

2. Declare a pointer to a `LucEndpoint` object. A `LucEndpoint` is the abstract base class for both the Linux and MTK implementations of LUC endpoints and defines the user interface as virtual functions. Internal to the LUC implementation, there are two subclasses that are derived from the `LucEndpoint` abstract class: `LucPortalsEndpoint` is the Linux implementation, and `LucFioEndpoint` is the MTK implementation. These derived classes implement the virtual functions for either Linux/Portals or MTK/FAST I/O. From the user-application perspective, both derived classes present an identical interface.

3. Allocate the object by calling `luc_allocate_endpoint()`. This function takes a service type as an argument and allocates the correct `LucEndpoint` derived class object. When compiling for a Linux node, the Linux version of the object is returned. When compiling for an MTK node, the MTK version of the object is returned.

4. Activate the client endpoint by calling `startService`. This causes LUC to allocate a system wide unique endpoint identifier and to allocate the underlying Fast I/O data streams. If an error is encountered while activating the service, LUC returns an error code

5. Prepare the input and output buffers. The input buffer is provided as input to the remote server function. The output buffer contains the output data from the remote server function. The buffers may reside in nearby or global distributed memory.

6. Invoke a remote function synchronously by calling `remoteCallSync`; provide the server endpoint identifier, the service type, the function index, and the input and output buffers and lengths. The `outputDataLen` parameter specifies the size of the data buffer provided by the caller. On return from the function, this parameter will contain the actual size of the output data, which is less than or equal to the original value provided by the caller.

7. The service type and function index are application defined and can be any integer value. As illustrated in the example that follows, the function indices need not be consecutive. The service types describe the type of service provided by the object.

8. Wait for the remote function to complete and then process the result. The `remoteCallSync()` method will not return until the remote function has completed or an error has occurred. The return value from `remoteCallSync` is either a LUC error code or the return value from the remote function.

9. Stop the service by calling `stopService`. This releases any nearby memory that was allocated by the endpoint, closes all previously opened Fast I/O data streams, and deactivates the object.

10. Delete the object. This invokes the virtual destructor for the derived object. If an endpoint object is deleted before calling `stopService`, the destructor automatically stops the service and deactivates the object.

Example 11. LUC client code example

```
user_application_defs.h  // sample header

// function index definitions
// note that the values do not have to be contiguous
#define FUNC_QUERY1       1
#define FUNC_QUERY3       3
#define FUNC_QUERY8       8

// service type definitions
#define  QUERY_MANAGER 1
#define  QUERY_ENGINE  2
#define  UPDATE_MANAGER 3
#define  UPDATE_ENGINE 4


user_application.cpp //sample client code
#include <luc/luc_exported.h>
#include <user_app_definitions.h>

const int INBUF_SIZE = (1 * 1024 * 1024);    // 1 MB input data
const int OUTBUF_SIZE = (2 * 1024 * 1024);   // 2 MB output data

void client(luc_endpoint_id_t serverID)
{
   LucEndpoint *clientEndpoint;
  luc_error_t result;
  char *outbuf = malloc(OUTBUF_SIZE);
  char *inbuf = malloc(INBUF_SIZE);
  size_t outDataLen = OUTBUF_SIZE;

  clientEndpoint = luc_allocate_endpoint(LUC_CLIENT_ONLY);
  result = clientEndpoint->startService();
  if (result != LUC_ERR_OK)
    {
    // process LUC error
    delete clientEndpoint;
    return;
  }
  result = clientEndpoint->remoteCallSync(serverID,
            QUERY_ENGINE, FUNC_QUERY1,
            inbuf, INBUF_SIZE, outbuf, &outDatLen);

    if(result == LUC_ERR_OK)
    // The RPC was successful.
    // outDataLen contains the size of data returned in outbuf.
    else if result < LUC_ERR_MAX)
    {
      // Result contains a LUC error code.
  }
  else
    {
      // Result is the return value from remote function
  }

    clientEndpoint->stopService();
    delete clientEndpoint;
}
```

5.3 Creating and Using a LUC Server

The server allocates and activates an endpoint object in a manner similar to that of the client. Object deactivation and deletion are also similar. The primary difference is the requirement for the server to register its remote functions. Use the following steps to create a server object.

Procedure 4. Creating and using a LUC server object

1. Include the header file `<luc/luc_exported.h>`, as well as the application defined header file.
2. Declare a pointer to a `LucEndpoint` object.
3. Allocate the object by calling `luc_allocate_endpoint`.
4. Call `registerRemoteCall` to register each function that will be serviced by this endpoint. The first parameter is the service type, the second parameter is the function index, and the third parameter is the address of the function.
5. Activate the server endpoint by calling `startService`. The parameter is the number of LUC worker threads to start. The default is `1`. The MTK version of the library ignores this value and creates one worker thread for each RPC. This method call causes LUC to allocate a number of nearby memory buffers for incoming requests and pre-post these receive buffers with the Fast I/O driver. The worker threads service the client requests as they come in.
6. Wait for a request to halt the service. There are many ways to accomplish this. In the following MTK example, the main application server thread then waits to be told to halt the service — by doing a synchronized read on an empty memory location. When the request is received, the application stops the service and deletes the endpoint. The application coordinates the notification to the server to

shutdown the service. For instance, if a serious application internal error occurs or an application shutdown request is received, the server must be told to halt by the application.

Example 12. LUC Server code example

```
#include <luc/luc_exported.h>
#include <user_app_definitions.h> (see below, step 6)

void server()
{
  LucEndpoint *svrEndpoint;
  luc_error_t err;

  svrEndpoint = luc_allocate_endpoint(LUC_SERVER_ONLY);

  err = svrEndpoint->registerRemoteCall(QUERY_ENGINE,
                                        FUNC_QUERY1, query1);
  if (err != LUC_ERR_OK)
  {
    // Process LUC error code
    delete svrEndpoint;
    return;
  }
  // Register more remote calls as above ....

  err = svrEndpoint->startService();
  if (err != LUC_ERR_OK)
  {
    // process LUC error code
    delete svrEndpoint;
    return;
  }

  readfe(&haltService);  // MTK full-empty synchronization
  svrEndpoint->stopService();
  delete svrEndpoint;
  return;
}
```

5.4 Communication Between LUC Objects

The following example shows how the application uses the client and server objects to communicate.

Example 13. Allocating and using `LucEndpoint` objects to communicate

```
// Application-specific definitions
#define QUERY_ENGINE_ALIVE_FCTN_ID              1
#define QUERY_ENGINE_DATA_BOUNCE_FCTN_ID        2

//
// This asynchronous completion handler conforms to LUC_Completion_Handler
//

void ClientCompletionHandler(luc_endpoint_id_t    destAddr,
                             luc_service_type_t   serviceType,
                             int                  serviceFunctionIndex,
                             void *               userHandle,
                             luc_error_t          remoteLucError)
{
  // In the example given, 'userHandle' will equal 0xf00
  return;
}

void LucClientOnlyUsageModel(void)
{

    // First create an endpoint. This is used to make the remote calls.
    LucEndpoint *client = luc_allocate_endpoint(LUC_CLIENT_ONLY);

    // In order to issue the remote calls, we need to know where to send them.
    // The library uses the abstract 64 bit 'luc_endpoint_id_t' value, so the
    // client application has to get this value from the server by some other
    // means.

    luc_endpoint_id_t       serverEndpointId;

    // This example assumes that 'serverEndpointId' is filled in by some
    // other means; environment variable, command line option, etc.

    // Enable the local endpoint. This will create worker threads and allocate
    // resources.
    luc_error_t lucError = client->startService();
    if (LUC_ERR_OK != lucError)
    {
      // error case
      delete client;
      return;
    }

    // Once the client object has been started successfully, the application
    // can use it to make synchronous and asynchronous calls.

    // A synchronous (blocking) call.
    // The application is responsible for setting serviceType and
    // serviceFunctionIndex to something meaningful (ie. something
    // registered by the object at 'serverEndpointId').

    luc_service_type_t      serviceType = LUC_ST_QueryEngine;
    int                     serviceFunctionIndex = QUERY_ENGINE_ALIVE_FCTN_ID;

    // This particular remote call passes no data.
    lucError = client->remoteCallSync(serverEndpointId,
                                      serviceType,
                                      serviceFunctionIndex,
                                      NULL,  // void   *inputData,
                                      0,     // size_t  inputDataLen,
                                      NULL,  // void   *outputData,
                                      0);    // size_t *outputDataLen);

    if(lucError == LUC_ERR_OK)
    //RPC was successful
    else if (lucError < LUC_ERR_MAX)
    // LUC library generated error code
    else
    // user remote function return value

    //
    // An asynchronous (non-blocking) call.
    // Return data is not supported for asynchronous callers.
    //
    void    *myMeaningfulHandle = 0xf00;

    lucError = client->remoteCall(serverEndpointId,
                                  serviceType,
                                  serviceFunctionIndex,
                                  NULL,  // void  *inputData,
                                  0,     // size_t inputDataLen,
                                  myMeaningfulHandle,
                                  ClientCompletionHandler);

    // The application can do other work while the remote call is in progress.
    // ClientCompletionHandler will fire in some other context at a later time.


    // When the application is finished with the endpoint object, it should
    // be stopped.
    lucError = client->stopService();

    // and destroyed.
    delete client;

    return;
}
// ServerQueryEngineAliveFunction:
// implements {LUC_ST_QueryEngine, QUERY_ENGINE_ALIVE_FCTN_ID}
// conforms to LUC_RPC_Function_InOut prototype
//
luc_error_t ServerQueryEngineAliveFunction(void *                inData,
                                    u_int64_t                inDataLen,
                                    void **                  outData,
                                    u_int64_t *              outDataLen,
                                    void **                  completionArg,
                                    LUC_Mem_Avail_Completion *   completionFctn,
                                    luc_endpoint_id_t            callerEndpoint)
{
  // This function is a simple case. It does not accept or return any data.

  if (*outData)
    *outData = NULL;
  if (*outDataLen)
    *outDataLen = 0;

  // Since this function is not returning data, it does not need to register
  // a memory-available (or dereference) handler.
  *completionFctn = NULL;

  return LUC_ERR_OK;  // successful return code
}
```

```
void LucServerOnlyUsageModel(void)
{
  // First create a communication endpoint. This is used to accept calls from
  // remote clients.
  LucEndpoint *server = luc_allocate_endpoint(LUC_SERVER_ONLY);

  // These values correspond to values used by clients of this service.
  luc_service_type_t serviceType = LUC_ST_QueryEngine;
  int serviceFunctionIndex = QUERY_ENGINE_ALIVE_FCTN_ID;

  // The registration routine simply records the desired function in a
  // table so that future client requests know which function to fire.
  lucError = server->registerRemoteCall(serviceType,
                                        serviceFunctionIndex,
                                        ServerQueryEngineAliveFunction);


  // The LucEndpoint object must be started before it can accept remote
  // function call requests.

  // This example creates two server worker threads; one to do main processing
  // and one to execute the ServerQueryEngineAliveFunction when it's called.
  uint_t  totalThreadCount = 2;

  // This server doesn't need a specific Portals PID value.
  uint_t  requestedPid = PTL_PID_ANY;

  lucError = server->startService(totalThreadCount,
                                  requestedPid);

  // If the server wants to report its endpoint id, via printf or socket-based
  // communication to some other application, it can get its endpoint ID
  // with the following function.
  luc_endpoint_id_t myEndpointId = server->getMyEndpointId();

  // A proper service can go do other work here, wait for a termination
  // signal, or exit this thread (as long as the server object isn't
  // destroyed).

  // The endpoint object will accept and remote function requests
  // until stopped at some later time with stopService.
  lucError = server->stopService();

  delete server;

  return;
}
```

5.5 LUC Client/Server Example

This example implements a server-side sum of values provided by the client, with the sum returned to the client. The program should be run once using the following command to start the server:

```
% exluc -s
```

Then the client can be run multiple times using the following command:

```
% exluc -c id
```

Where *id* is the server endpoint ID printed to the command line when the server starts.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>  // htonl/ntohl byte swapping
#include <luc/luc_exported.h>


// The service type is an application-specific major service id.
// It identifies the general type of service requested by the client.
// One server may implement one or more service types.
// For this example, one service type is defined.

int svc_type = 0;

// The function index is an application-specific minor service id.
// It identifies a specific server function out of the functions defined
// by the server within one of its supported service types.
// Each service type may implement one or more functions.
// For this example, one function within the svc_type service type is defined.
int reduce_func_idx = 0;

#define NREDUCE  10 // number of values to be summed

// Opteron uses little-endian byte order and XMT uses big-endian byte order.
// When an Opteron client uses an XMT server, byte swapping is required
// to convert the data between the two systems.
// This example uses network byte order (big-endian) for all LUC data transfers,
// and converts to host byte order before using LUC data.

#if defined(__MTA__) || defined(NO_BYTE_SWAP)

// Host byte order is the same as network byte order on XMT,
// so no conversion is necssary.

#define NetworkToHost(b,l)
#define HostToNetwork(b,l)
#else

// Byte swap to convert between host and network byte ordering.

void ByteSwap(void *buf, size_t len)
{
  char *c = (char *) buf;
  int i;
  for (i=0;i < len;i+=2)
  {
    char t = c[0];
    c[0] = c[1];
    c[1] = t;
  }
}
```

```
#define NetworkToHost(b,l) ByteSwap((b),(l))
#define HostToNetwork(b,l) ByteSwap((b),(l))

#endif

// The LUC client runs on the XMT login node, and acts as
// the application user interface.
// Return value is 0 for success, 1 for error.
int client(luc_endpoint_id_t serverID)
{
  double input[NREDUCE];                       // input data
  double output;                               // result
  size_t in_size  = sizeof(double) * NREDUCE;  // input size in bytes
  size_t out_size = sizeof(double);            // output size in bytes
  luc_error_t err;                             // result code from LUC calls

  // Initialize the input data.
  for (int i=0;i < NREDUCE;i++) input[i] = i;

  // Create the LUC client endpoint.
  LucEndpoint *clientEndpoint = luc_allocate_endpoint(LUC_CLIENT_ONLY);

  // Initialize the endpoint (connect to the server).
  err = clientEndpoint->startService();
  if (err != LUC_ERR_OK)
  {
    fprintf(stderr,"client: LUC startService error %d\n",err);
    delete clientEndpoint; // free memory
    return 1;              // error
  }

  HostToNetwork(input,in_size);  // convert data to network byte order

  // Send the request to the server and wait for a response.
  // In this example, the array of values to be summed is sent,
  // and the sum is returned as the result.
  err = clientEndpoint->remoteCallSync(serverID, svc_type, reduce_func_idx,
 input, in_size, &output, &out_size);

  if (err != LUC_ERR_OK)
  {
    // err contains a LUC error code.
    fprintf(stderr,"client: LUC remoteCallSync error %d\n",err);
  }
    else
  {
    // out_size contains the size of data returned in outbuf
    NetworkToHost(&output,out_size); // convert data to host byte order
    printf("The sum of the %d values is %lf\n",NREDUCE,output);
  }

  clientEndpoint->stopService();  // disconnect from the server
  delete clientEndpoint;          // free memory

  return (err != LUC_ERR_OK) ? 1 : 0;
}
```

```
// Reduction service.
// This routine is called by the LUC server library
// when a client request of type
// (svc_type,reduce_func_idx) is received.
luc_error_t reduce(void *inPtr, u_int64_t inDataLen,
                   void **outPtr, u_int64_t *outDataLen,
                   void **completionArg, LUC_Mem_Avail_Completion *completionFctn,
                   luc_endpoint_id_t callerEndpoint)
{
  double *input  = (double *) inPtr;  // input data
  double *output = NULL;
  int n = inDataLen / sizeof(double); // number of values to sum

  // Default (error) return will be no output data
  *outPtr = NULL;
  *completionArg = NULL;
  *completionFctn = NULL;

  // Allocate space for the return data
  output = (double *)malloc(sizeof(double));
  if (NULL == output)
  {
    return LUC_ERR_RESOURCE_FAILURE; // or use a custom code
  }

  NetworkToHost(input,inDataLen);   // convert data to host byte order

  // Perform the reduction.
  double sum = 0;
  for (int i=0;i < n;i++) sum += input[i];

  *output = sum;                          // set result value
  HostToNetwork(output,sizeof(double));  // convert result to network byte order
  *outDataLen = sizeof(double);          // set result size
  *outPtr = (void *)output;              // set result / output pointer

  // Tell LUC to call 'free' when it is done without the output data.
  // Pass the 'output' pointer to free()
  *completionArg = output;
  *completionFctn = free;

  return LUC_ERR_OK;

}


// The LUC server can run on the XMT login node or in the compute partition.
// Return value is 0 for success, 1 for error.

int server(int threadCount)
{
  luc_error_t err;  // result code from LUC calls

  // Create the LUC server endpoint.
  LucEndpoint *svrEndpoint = luc_allocate_endpoint(LUC_SERVER_ONLY);

  // Register routines which implement the services.
  err = svrEndpoint->registerRemoteCall(svc_type, reduce_func_idx, reduce);
  if (err != LUC_ERR_OK)
  {
    fprintf(stderr,"client: LUC registerRemoteCall error %d\n",err);
    delete svrEndpoint;
    return 1; // error
  }

  // Begin offering services (begin listening for requests).
  err = svrEndpoint->startService(threadCount);
  if (err != LUC_ERR_OK)
  {
    fprintf(stderr,"client: LUC startService error %d\n",err);
    delete svrEndpoint;
    return 1; // error
  }

  // Print out the endpoint id for the server.  This value is a required
  // input for the client.
  fprintf(stderr,"server: Server ready.
          My endpoint id is %ld\n",svrEndpoint->getMyEndpointId());

  // At this point, the main server thread waits while requests
  // to the server are handled by other threads.
  // A "terminate server" client request can be defined by the
  // application to handle server shutdown, or else the server can
  // simply be killed when the server is no longer needed.
  // For this example, the server waits until it is killed.
  getc(stdin);

  // The server has been requested to shut down.
  svrEndpoint->stopService(); // stop listening for requests
  delete svrEndpoint;  // free memory
  return 0;
}


// The main program either calls the server routine or the client
// routine. The server (-s) should be started first, then the
// client (-c id) can be run multiple times.
// Shut down by killing the server process.
int main(int argc, char **argv)
{
  luc_endpoint_id_t id;
  int i;

  while ((i = getopt(argc,argv,"c:s")) != EOF)
  {
    switch (i)
    {
      case 'c':
        id = strtoul(optarg, NULL, 0);
        return client(id); // make a request to server with this endpoint id

      case 's':
        return server(1); // start server with 1 request-processing thread
    }
  }
  // If no valid options were given, print the program usage message.
  fprintf(stderr,"Usage: exluc -c id | -s\n");
  fprintf(stderr,"-c id Run as a client with the given endpoint id.\n");
  fprintf(stderr,"-s Run as a server, printing the endpoint id.\n");
  return 1;
}
```

5.6 Fast I/O Memory Usage

The MTK Fast I/O Library performs all data transfer operations through nearby memory. Nearby memory is memory on the same node as the Threadstorm processor where the LUC endpoint was started. The library transfers user data into and out of nearby memory buffers automatically. Use configuration variables to control the amount of nearby memory used by the library.

The MTK Fast I/O Library uses one or two regions of nearby memory for each local endpoint as I/O buffers. The library requires one region for all small allocations and allows for an optional region for large allocations. The small region is used for core RPC data structures that are sent over the high speed network. Small data transfer buffers may use the small region as well. The optional large memory region is used for large transfer requests and many concurrent smaller requests. The large region may be sized to support one very large RPC request or several smaller requests.

To control the size of the small memory region use the configuration variable `LUC_CONFIG_MAX_SMALL_NEARMEM_SIZE`. Legal values range from 1 MB (1,048,576) to 256 MB (268,435,456), inclusive, in power-of-two increments. The size of the largest allowable request on this memory region may be specified with the `LUC_CONFIG_MAX_SMALL_MEM_REQUEST` variable. Legal values range from 64 KB (65,536) to one half of the current small memory region size, inclusive, in power-of-two increments.

To control the size of the large memory region use the configuration variable `LUC_CONFIG_MAX_LARGE_NEARMEM_SIZE`. Legal values range from 1 MB (1,048,576) to 2 GB (2,147,483,648), inclusive, in power-of-two increments. While the library allows for a very large nearby memory region, the system may not be configured with enough nearby memory to support a maximum size nearby memory region. The size of the largest allowable request on this memory region may be specified with the `LUC_CONFIG_MAX_LARGE_MEM_REQUEST` variable. Legal values range from 1 MB (1,048,576) to the current large memory region size or 256 MB, whichever is less. The maximum request size must be an integral power-of-two.

To disable the large memory region specify a requested size of zero.

Initialize the memory region variables from the global variables when creating the LUC Endpoint object. Changes to the global variables are propagated to new endpoint objects, not objects that already exist. An endpoint's memory configuration variables may be changed by using the `LucEndpoint::setConfigValue()` method until the endpoint is started. Once the endpoint starts, the size of the nearby memory regions and the maximum transfer sizes are locked in and may not be modified until you stop the endpoint. Attempts to change these configuration variables by using `LucEndpoint::setConfigValue()` fail with `LUC_ERR_INVALID_STATE`. If you try to change the global configuration variables, the changes do not propagate to started endpoints. Attempts to set invalid memory sizes or maximum request sizes fail with `LUC_ERR_BAD_PARAMETER`.

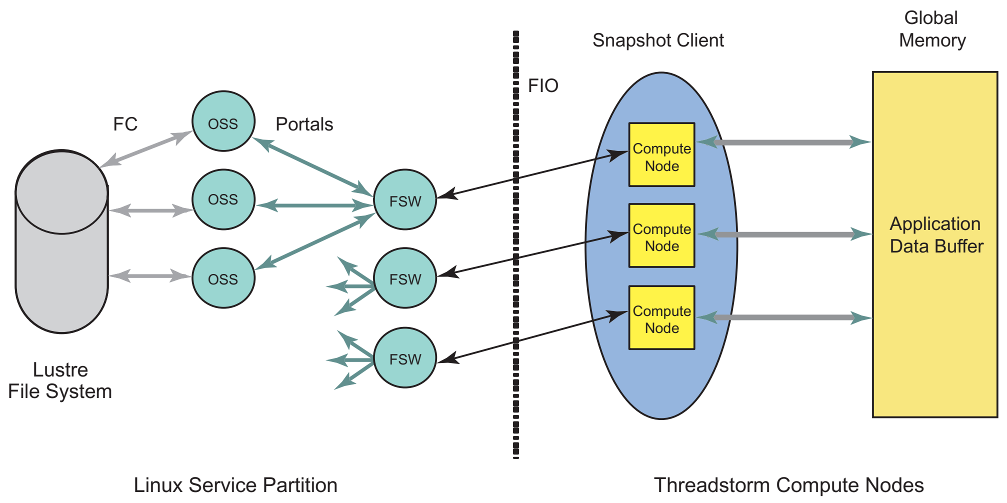
Managing Lustre I/O with the Snapshot Library [6]

6.1 About the Snapshot Library

The Cray XMT snapshot library provides a high speed bulk data transfer facility that moves data between memory regions within an MTK application and files hosted on the XMT Linux service partition. The primary use of the snapshot library is to load and save large data sets that are being stored on a Lustre file system. For example, an application might use the snapshot library to load a large data set at the beginning of a run, process the data, then use the snapshot library to save the processed data in a file at the end of a run. An application might also use the snapshot library to save intermediate copies of the processed data during the course of a run.

The snapshot library uses the Fast IO (FIO) mechanism on the compute partition to transfer data, in parallel, to and from files on the service partition using instances of a helper program called `fsworker` that provide file system access on login nodes. Multiple instances of `fsworker` can be used in parallel to provide higher throughput. This figure shows the most common data communication paths between an application using the snapshot library and a file on the compute partition. The data moves, in four distinct stages, between a global memory buffer in the application and a file on a Lustre file system hosted by the service partition.

Figure 1. Snapshot Library Data Paths



The easiest way to understand this is to imagine data going to a file from the application. In this case, the data is copied by each compute node into the FIO transport and sent to its corresponding `fsworker` on a login node in the Linux service partition. Each `fsworker` then uses Linux system calls to write data into the Lustre file, which results in the data moving across the Portals transport from the login node to one or more Lustre OSS nodes. From there, the data moves through Fibre Channel (FC) to the actual storage device.

Moving data from a file to the application simply reverses the order of the stages and the direction of the data flow through each stage, ultimately resulting in data being copied from compute nodes into the application's global memory buffer.

6.2 The Snapshot Library Interface

Note: Effective with Cray XMT version 2.0 the `snap_*` functions are replaced by `dslr_*` equivalents. The `snap_*` functions are deprecated and will be removed in a future release.

The snapshot library interface consists of these functions:

`dslr_snapshot`	Copies data in parallel from a buffer in the application to a file on the service partition.
`dslr_restore`	Copies data in parallel from a file on the service partition to a buffer in the application.
`dslr_pread`	Allows the application to specify an offset into a file from which to read data. Does not move data in parallel.
`dslr_pwrite`	Allows the application to specify an offset into a file at which to write data. Does not move data in parallel.
`dslr_stat`	Allows the application to obtain file status from a file, similar to the `stat` function.
`dslr_truncate`	Truncates a file to a specified length.

For more information on any of these functions, see the associated man page.

For large data transfers starting at the beginning of a file, the best functions to use are `dslr_snapshot` and `dslr_restore`, because they are able to transfer data in parallel to achieve high throughput. To store data, the application calls `dslr_snapshot`, specifying the buffer to be copied, the length of the data, and the name of the file receiving the data. To read back (restore) data from the file into application memory, the application calls `dslr_restore`, specifying the buffer receiving the data, the length of the data to read, and the name of the file providing the data. Because this name will be used by all instances of `fsworker` to open and read or write the file the file name should be an absolute path name to the location of the file on the service partition. A relative path name could be ambiguous or meaningless to a particular `fsworker`.

A typical application might use `dslr_restore` and `dslr_snapshot` in the following manner:

1. Start up and allocate a large buffer to hold a data set.
2. Call `dslr_restore` specifying the name of the file providing the data, the buffer allocated in step 1, and the length of that buffer.
3. Process and change the data set.
4. Call `dslr_snapshot` to store the data set back to the file (or to a new modified data file).
5. If necessary repeat 3 and 4, using the snapshots as a way to preserve forward progress.

The `dslr_pwrite` and `dslr_pread` functions are provided for transferring smaller amounts of data between a buffer and arbitrary locations in a file. To write data to a file, the application calls `dslr_pwrite` specifying the endpoint-ID of a single `fsworker`, the name of the file, the offset of the data in the file, a pointer to a buffer from which to take the data, and the length of the data to be written. To read data from a file, the application calls `dslr_pread` specifying the endpoint-ID of a single `fsworker`, the name of the file, the offset of the data in the file, a pointer to a buffer into which to put the data, and the length of the data to be read. Again, absolute path names for files are strongly recommended.

A typical application might use `dslr_pread` and `dslr_pwrite` in the following manner:

1. Start up and allocate a small buffer to be initialized from a file.
2. Call `dslr_pread` specifying the name of the file providing the data, the offset of the data in the file, a pointer to the buffer allocated in 1, and the length of the data.
3. Process and change the data.
4. Call `dslr_pwrite` to store the data back to the file (or to a new modified data file).
5. Repeat 3 and 4 as often needed, using snapshots as a way to preserve forward progress in case of failure or for the sake of sharing the system.

It is possible to mix uses of `dslr_snapshot`/`dslr_restore` and uses of `dslr_pwrite`/`dslr_pread` as needed in an application.

⚠ **Caution:** The snapshot library functions can only be used one at a time; they cannot be used in parallel. Any attempt to use snapshot library functions in parallel will eventually result in corruption of the snapshot data and possible uncontrolled failure of the snapshot library or of one or more instances of `fsworker`.

6.3 Maintaining File System and I/O Parallelism

The snapshot library is intended primarily for saving and retrieving large data sets on platforms with a Lustre file system. Lustre supports parallel access and is highly tunable, allowing users and administrators to set many options, including file stripe widths and block sizes. With proper provisioning and tuning, Lustre can sustain many gigabytes per second of throughput. Because the performance of the underlying Lustre configuration bounds the throughput of most snapshot library operations, careful Lustre tuning is essential for optimal snapshot performance.

A detailed discussion of Lustre provisioning, configuration and tuning are beyond the scope of this document. One rule of thumb, however, makes a good starting point when using `dslr_snapshot` and `dslr_restore` in single-file mode with multiple `fsworkers`. Setting the block size to 32 megabytes and a file stripe width of all *object storage server* (OSS) nodes (-1) generally yields good results. Typically, for multi-file mode the directory is striped to a single *object storage target* (OST). The `lfs` command allows a user to set these parameters on a per-directory basis. See the `setstripe/getstripe` documentation in the `lfs` man page for more information. Contact your system administrator for more detailed information on tuning Lustre to the requirements of a particular application.

If the underlying file system is naturally serial (NFS, for example) its performance is constrained by the serial performance of the file system and any contention introduced by trying to use the file system in parallel. Again, the throughput of the snapshot library is bounded by the file system performance, so when using a serial file system a single `fsworker` provides the best throughput for the snapshot library. Note that `fsworkers` are not resilient. If a transaction fails, all involved `fsworkers` must be terminated and restarted. If the file system is full a snapshot function may return success even though the file was not written, or was only partially written.

6.4 Examples

Example 14. Using `dslr_snapshot` and `dslr_restore` to save and restore data in a file.

Note that this example waits for the call to `dslr_snapshot` to complete before calling `dslr_restore`. While this is logical in this example, it is also crucial for correct operation. (See the caution about using snapshot library functions in parallel above.)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <snapshot/client.h>
const size_t DEFAULT_BUFFER_SIZE = 1024 * 1024 * 1024;
const char DEFAULT_FILENAME[] = "/mnt/lustre/myusername/snapshot.data";
int main(int argc, char *argv[])
{
  void *testBuffer = NULL;
  int64_t err;
  int64_t snapError = 0;
// Allocate a large buffer to be transferred.
  if (NULL == (testBuffer = malloc(DEFAULT_BUFFER_SIZE)))
  {
    fprintf(stderr,"Failed to malloc %d byte snapshot buffer.\n",
    DEFAULT_BUFFER_SIZE);
    return -1;
  }
  memset(testBuffer, 'a', DEFAULT_BUFFER_SIZE);
  // Snapshot the testBuffer to disk
  // All file system workers must be able to access the specified path.
  err = dslr_snapshot ((char *)DEFAULT_FILENAME, testBuffer,
  DEFAULT_BUFFER_SIZE, &snapError );
  if (dslr_ERR_OK != err)
    {
    fprintf(stderr,"Failed to snapshot the dataset. Error %d.\n",err);free(testBuffer);
    return -1;
    }
  memset(testBuffer, 0, DEFAULT_BUFFER_SIZE);
  // Restore a snapshot dataset from disk back into memory.
  err = dslr_restore ((char *)DEFAULT_FILENAME, testBuffer,
  DEFAULT_BUFFER_SIZE, &snapError);
  if (dslr_ERR_OK != err)
    {
    fprintf(stderr,"Failed to restore the dataset. Error %d.\n",err);
    free(testBuffer);
    return -1;
    }
  // At this point, the testBuffer should be full of 'a'
  free(testBuffer);
  return 0;
}
```

Example 15. Using `dslr_pwrite` to write data to a file and `dslr_pread` to read back the data

Note that the calls to `dslr_pwrite` and `dslr_pread` accept the value `dslr_ANY_SW` to specify the endpoint ID of the `fsworker`, allowing libsnapshot to use any registered endpoint. Therefore, the `fsworkerID` is automatically set to `dslr_ANY_SW` rather than requiring the user to enter the endpoint either manually or by the environment.

Also note that, while the function call interface appears to invite parallel use of `dslr_pwrite` and `dslr_pread`, the functions cannot be used in parallel. Concurrent calls to these or any other snapshot library functions results in the problems described in the caution statement above. Regardless of how the endpoint is set, only one thread of one instance of `fsworker` will be applied to any given call to `dslr_pwrite` and `dslr_pread`.

While these functions are useful for transferring small quantities of data to or from arbitrary locations in files but, because they are unable to benefit from parallelism, they are not useful for bulk data transfer. You should not expect throughput greater than 100MB/second when using `dslr_pwrite` or `dslr_pread`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <snapshot/client.h>
const size_t DEFAULT_BUFFER_SIZE = 1024 * 1024; // Relatively short buffer
const char DEFAULT_FILENAME[] = "/mnt/lustre/myusername/snapshot.data";
int main(int argc, char *argv[])
  {
  void *testBuffer = NULL;]
  int64_t err;
  int64_t snapError = 0;
  uint64_t fsworkerID = dslr_ANY_SW;
  off_t fileOffset = 0;
  int rc = 0;
  // Allocate a small buffer to be transferred.
  if (NULL == (testBuffer = malloc(DEFAULT_BUFFER_SIZE)))
    {
    fprintf(stderr,"Failed to malloc %d byte snapshot buffer.\n",
    DEFAULT_BUFFER_SIZE);
    return -1;
    }
  memset(testBuffer, 'a', DEFAULT_BUFFER_SIZE);
  // pwrite the testBuffer to disk
  fileOffset = 0;
  err = dslr_pwrite((char *)DEFAULT_FILENAME,
        fsworkerID,
        testBuffer, DEFAULT_BUFFER_SIZE,
        fileOffset,&snapError );
  if (dslr_ERR_OK != err)
    {
    fprintf(stderr,"Failed to pwrite the dataset. Error %d.\n",err);
    free(testBuffer);
    return -1;
    }
  memset(testBuffer, 0, DEFAULT_BUFFER_SIZE);
  // pread the testBuffer from disk.
  err = dslr_pread ((char *)DEFAULT_FILENAME,
        fsworkerID,
        testBuffer, DEFAULT_BUFFER_SIZE,
        fileOffset, &snapError);
  if (dslr_ERR_OK != err)
    {
    fprintf(stderr,"Failed to pread the dataset. Error %d.\n",err);
    free(testBuffer);
    return -1;
    }
  // At this point, the testBuffer should be full of 'a'
  free(testBuffer);
  return 0;
}
```

6.5 Managing File I/O on File Systems Other Than Lustre

Using the snapshot library to read and write files on a file system, such as NFS that does not support high performance parallel I/O can result in overloading the underlying file system with data requests and transfers. Cray does not support this use of the snapshot library on any system with more than a single login node, as even file transfers of a few hundred MB can cause unacceptable network congestions.

The standard operating system I/O functions `OPEN`(2), `close`(2), `read`(2) and `write`(2) are available for reading and writing files on NFS file systems that are cross-mounted to the compute partition. Files larger than 1 GB should always be read or written using the `dslr*` functions to a high performance parallel file system, such as Lustre.

Compiler Overview [7]

This chapter provides an overview of the Cray XMT compilers. You need to understand these concepts before you compile your program.

The Cray XMT platform includes Cray XMT compilers for C and C++ applications. These compilers optimize programs to improve performance. These features include:

Debugging support

The Cray XMT compilers support multiple levels of debugging. Each level receives some degree of optimization, but the level of optimization decreases as the level of debugging support increases. For example, the compilation process suppresses parallelization of loops at the highest debugging level.

Optimization

The Cray XMT compilers optimize parallelization, loop restructuring, and software pipelining, in addition to the classical scalar optimizations.

Inlining

The Cray XMT compilers support automatic and programmer-directed inlining within source files and among multiple source files. In addition, the compilers support inlining from separately compiled libraries. For a discussion of inlining, see Inlining Functions on page 84.

Incremental recompiling and relinking

The Cray XMT compiler detects unmodified functions and avoids recompiling them, even when other functions in the same file have been changed. The Cray XMT compiler uses incremental linking to avoid relinking an entire executable when some, but not all, of the functions have been modified.

Each compiler is organized as a language-dependent front end. Both compilers use a common set of backend subprograms for translating, optimizing, and linking. The Cray XMT C compiler supports ANSI X3.159-1989 standard C. The Cray XMT C++ compiler supports the draft ISO/IEC 14882 C++ standard. Because of the commonality between the Cray XMT C and C++ compilers, they are referred to collectively as the *compiler* in the remainder of this chapter.

7.1 The Compilation Process

There are two major phases of building a program executable from a number of source files.

Compilation	The compiler creates object files by invoking subprograms that translate the source files and optimize functions in the program. The compiler starts by invoking the front end. When the front end finishes, the compiler invokes the translator, which is the subprogram that optimizes and parallelizes code, and generates object files.
Linking	The compiler creates an executable program by invoking subprograms that create links between object files created during the compilation process and any associated libraries. Links can be created between two or more object files, in any combination, including the startup file, any specified object files or compile results, and user-created or standard libraries.

For a traditional UNIX compiler, you use the `cc -c file1.c` command to translate the source file `file1.c` into an object file, which, by default, is called `file1.o`. You then link a set of object files using the `cc file1.o file2.o` command. This creates an executable called `a.out`. Unfortunately, this approach to compilation decreases the efficiency of the resultant executable program because each file of functions is first compiled independently and then linked together in a separate process. Using this approach, information that the compiler uses to optimize functions during the first compilation is not available during the linking phase when the object files are combined to form an executable. As a result, the compiler cannot perform some optimizations between object files that might seem simple to a programmer.

In response to this problem, the Cray XMT compiler supports a compilation mode that enables information to be captured from individual modules and used when compiling multi-module programs. In this mode, each function is compiled in the context of a complete program, and the compiler may use facts about that context to optimize the translation of the function. The compiler retains this information so that when you modify your program's functions in the future, the compiler only needs to recompile the modified functions, resulting in a shorter recompile time. This mode is called *whole-program compilation*. The Cray XMT compiler also supports a mode for the traditional UNIX style of compiling called *separate-module compilation*.

The compilation processes for these modes differ in the following ways:

Whole-program compilation

> This is the preferred method for compiling applications. In whole-program compilation, the compilation phase is made up of several sub-phases. The compiler first parses (partially compiles) each source file. During this phase, the compiler gathers information about every module in the program and saves it to the *program library*. The next phase is the translation phase. During this phase, the program is translated and optimized. The compiler optimizes each function in the program using information from within that function's module or other modules, including linked libraries, that the compiler gathered earlier. Finally, in the linking phase, the compiler links separate modules into a program executable. Information about all modules is stored, and passed between phases, in the program library.

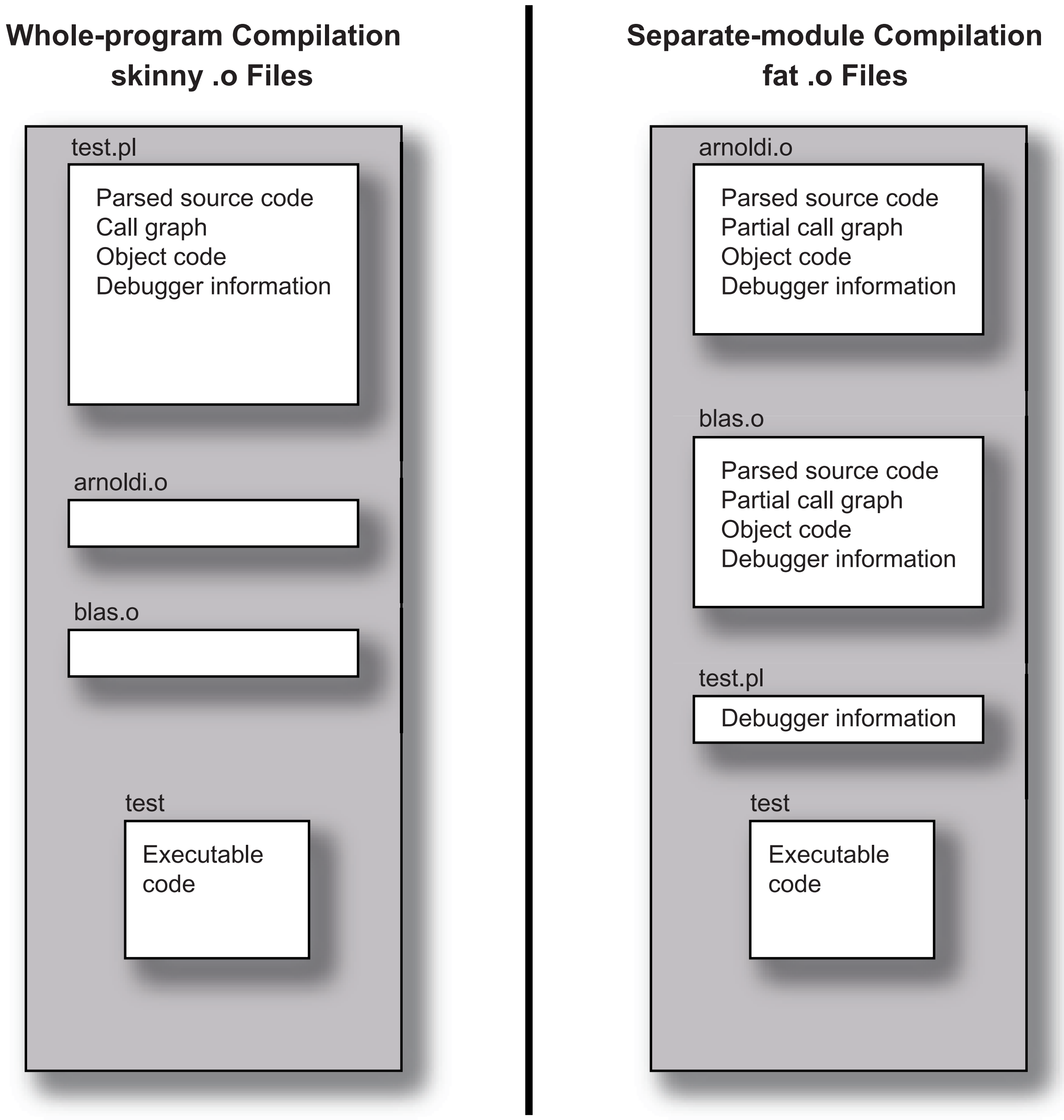
Separate-module compilation

> The compiler creates a separate object file for each source file and optimizes the functions within each source file using information about functions within that file. Then, the separate modules are linked to create a program executable.

Whole-program compilation generally produces more highly optimized code than separate-module compilation. You can compile a program using one mode or the other, or a combination of the two.

The following diagram shows the object files that the compiler creates when compiling the same `arnoldi.cc` and `blas.cc` files in different modes.

Figure 2. Comparison of Whole-program and Separate-module Modes



In whole-program mode, all the traditional object information for a program is contained in a single program library file. The program library has a `.pl` filename extension. The compilation process also produces an object file with a `.o` extension for each source file. This file is used as a time stamp to drive build processes. Each `.o` file corresponds to a module contained in the program library. The object information, or modules, for a program's source files are packaged together. This enables the compiler to optimize each function within the context of the entire program.

In separate-module mode, the `.o` files are true object files. The compiler optimizes each object file, or module, separate from the others. The link step produces a program library, although this program library primarily contains information that directs the debugger to various object files.

Because of the relative sizes of the `.o` files in the two compilation modes, the qualifier *skinny* refers to whole-program mode and its products (such as the `.o` files) and the qualifier *fat* refers to separate-module mode and its products.

During the compilation process, the compiler creates the following files:

`a.out`	The executable file.
`a.out.pl`	The program library.
`LOCK.a.out.pl`	The temporary lock file. The lock file prevents other compilers from accessing a program library when it is already in use. The compiler removes this file after use, unless the compiler terminates before completion.
`*.o`	Relocatable object files.

7.1.1 File Types Accepted by the Compiler

The compiler accepts files that use the following extensions:

`.c`	C file when invoked with `cc`, C++ file when invoked with `c++`.
`.cc, .cpp`	C++ file.
`.o`	In whole-program compilation, time stamp file that does not need to be compiled but participates in any link step. Also referred to as a skinny `.o` file. In separate-module compilation, a true object file. Also referred to as a fat `.o` file.
`.pl`	Program library. Used to support incremental recompiling and debugging. In whole-program compilation, used to support inter-module analysis.
`.a`	Archive or library file.

File prefixes used in the compilation process include the following:

`LOCK`	Temporary lock file used to prevent concurrent updates to the associated program library.

7.2 Invoking the Compiler

You can only use the Cray XMT compiler when the Cray XMT Programming Environment (`mta-pe`) module is loaded. The commands to use to invoke the compiler are `cc` for a C program and `c++` for a C++ program.

You can control the operation of the compiler by setting various options when running the compiler command. The compiler uses driver options, language options, parallelization options, and debugging options.

The driver options control how the compiler invokes subprograms. The compiler mode is set using driver options. The driver options that you use most often are the following:

`-c` *filename*

Compiles a specified source file.

`-o` *filename*

Links files and creates an executable.

`-pl` *filename*

Places object code and other data generated by the compiler into a program library file. This option is used for whole-program compilation.

For example, if you specify both the `-c` and `-pl` driver options, the compiler compiles the program in whole program mode, but it does not link the files into an executable. For more information, see Setting the Compiler Mode on page 80.

The language options control how the front end processes information. For example, the `-E` option indicates that the compiler should preprocess source files but not compile them. The `-no_float_opt` option prevents floating-point optimization.

The parallelization options control parallelism in the program. For example, the `-par1` option compiles a program so that it runs in parallel on a single processor. For more information, see Optimizing Parallelization on page 85.

The debugging options control how the debugger works. For more information, see Setting Debugger Options during Compilation on page 88.

Each compiler uses the same set of command-line options. For a complete list of command-line options, see the `cc`(1) or `c++`(1) man pages.

7.3 Setting the Compiler Mode

To set the compiler mode to whole-program mode, run the `cc` or `c++` command with the `-pl` option. This option builds a program library.

The following examples show how to use the compiler options for various compiler tasks using the whole-program and separate-module modes.

Whole-program:

`c++ -c a.cc -pl prog.pl`	(parses a.cc)
`c++ -c b.cc -pl prog.pl`	(parses b.cc)
`c++ -pl prog.pl -o prog a.o b.o`	(translates a.o, b.o; links prog)

Or, as a shortcut:

`c++ a.cc b.cc -o prog`	(compiles a.cc, b.cc; links prog, and creates prog.pl)

Separate-module:

`c++ -c a.cc`	(parses and translates a.cc)
`c++ -c b.cc`	(parses and translates b.cc)
`c++ -o prog a.o b.o`	(links prog)

7.3.1 Whole-program Mode

With whole-program compilation, the compiler has access to information about all functions in the program while optimizing each function. This information provides the compiler with the context for how the larger program uses each function. For example, when you use the `c++` command to link the files `jacobian.cc` and `blas.cc`, the compiler has access to the entire program during all but the initial compilation phases, and compiles the program in whole-program mode. To do this, use the following command:

```
c++ jacobian.cc blas.cc
```

The previous command produces the skinny `.o` files `jacobian.o` and `blas.o`, the executable `a.out`, and the program library `a.out.pl`.

Whole-program compilation enables inlining among files. The compiler can inline functions in `blas` into call sites in `jacobian`, and vice versa. The compiler can also inline functions into `jacobian` and `blas` from user-defined libraries linked with the program. See Creating New Libraries on page 87.

The compiler builds the program library `a.out.pl` during the compilation phase.

The whole-program compilation mode can be specified while retaining the flexibility of multiple compilation steps that you typically use for separate-module compilation. To do this, use the following sequence of commands:

```
c++ -pl test.pl -c ddot.cc
c++ -pl test.pl -c svd.cc
c++ -pl test.pl -o test svd.o ddot.o
```

The first two commands perform the initial compilation phase of `ddot.cc` and `svd.cc` using the program library `test.pl`. The last command specifies the construction of the `test` executable using the `test.pl` program library and the `svd` and `ddot` modules.

When you use the `-pl` and `-c` options to compile a source file, the compiler performs the following tasks during the compilation phase:

- Checks the source for syntax errors
- Creates an internal representation of each function in the program library
- Produces a skinny `.o` file

During the linking phase, the compiler performs the following tasks to create an executable:

- Performs optimizations using information about the complete program
- Builds objects for each module
- Links the modules together to produce an executable
- Stores objects in the program library to support incremental recompilation

As in traditional UNIX compilation, the `-o` flag specifies the executable name explicitly. To do this, use the following command:

```
c++ -pl test.pl -o test svd.o ddot.o
```

The previous command links the `svd` and `ddot` modules that reside in `test.pl` and creates the executable in a file called `test`.

You can also specify multistep command sequences that use a mix of source and object files when using whole-program mode. To do this, use the following sequence of commands:

```
c++ -pl a.out.pl -c ddot.cc
c++ -pl a.out.pl arnoldi.cc ddot.o
```

The first command partially compiles `ddot.cc`. The second command partially compiles `arnoldi.cc`; completes compilation and optimization of the modules `ddot` and `arnoldi`; links `arnoldi`, `ddot`, and any required libraries; and places the resulting executable in `a.out`. The compiler optimizes each function using information about the `ddot` and `arnoldi` modules.

7.3.2 Separate-module Mode

`-pl` flag to compilation and link lines. Separate-module mode also prevents the propagation of changes made in one module to other modules. This greatly reduces the level of optimization that occurs when using separate-module mode compared to that of whole-program mode.

To compile a single source file into its corresponding object file, use the following command.

```
c++ -c ddot.cc
```

This produces (barring errors in the source file) a traditional, or fat, object file `ddot.o`. To produce the two fat object files `ddot.o` and `daxpy.o`, each of the two source files can be compiled separately. To do this, use the following command.

```
c++ -c ddot.cc daxpy.cc
```

Using the previous command is the same as using the following sequence of commands.

```
c++ -c ddot.cc
c++ -c daxpy.cc
```

When compiling a file in separate-module mode, the compiler performs inter-function optimizations within individual files. As in whole-program mode, when the compiler constructs an executable, it also produces a program library. In separate-module mode, however, the program library is much smaller because it contains only information the debugger uses to locate more detailed debugging information in the separate fat object files.

7.3.3 Mixed Mode

Whole-program and separate-module mode may be used in combination to build a particular program. You can use mixed-mode to isolate code in fat modules from changes made in other skinny or fat modules. You can also use it to share the same piece of precompiled object code among several programs, while still allowing the programs to take advantage of whole-program optimizations performed on unshared code.

The following sequence of commands shows how to use mixed mode.

```
c++ -c arnoldi.cc
c++ -pl test.pl -c jacobian.cc blas.cc
c++ -pl test.pl -o test arnoldi.o jacobian.o blas.o
```

The first command compiles `arnoldi.cc` in separate-module mode, and produces the fat object file `arnold.o` . In this step, the compiler optimizes functions in `arnoldi.cc` without using information from the `jacobian` or `blas` functions.

The second command partially compiles `jacobian.cc` and `blas.cc` in whole-program mode, places the results in `test.pl`, and produces the skinny `.o` files `jacobian.o` and `blas.o` .

The third command performs final compilation and optimizations of functions from `jacobian.cc` and `blas.cc`, then links the functions to form the executable `test`. In this step, the compiler has knowledge of functions in `arnoldi.cc`, `jacobian.cc`, and `blas.cc`.

7.4 Inlining Functions

Inline expansion, commonly known as *inlining*, occurs when the compiler replaces a function reference with the body of the function. The advantages to using inlining include a reduction in memory usage due to the removal of function calls and returns, and the possibility of optimizing code near the function call with the function body. The disadvantages include an increase in the size of the executable and an increase in the level of complexity required during debugging.

When compiling in separate-module mode, the compiler inlines functions that are defined in the same file where they are referenced. When compiling in whole-program mode, the compiler can inline any function in the program or associated libraries. To view functions that are inlined, use the canal or Apprentice2 performance tools. See *Cray XMT Performance Tools User's Guide*.

You can use either command-line switches or compiler directives to control how the compiler inlines functions.

To set inlining from the command line, you can use either the `-inline` *fcn* to force the compiler to inline a specified function or `-no_inline` *fcn* to suppress inlining for the specified function. The option `-no_inline_all` suppresses inlining for all functions in a program.

For C++, the function name *fcn* must use the mangled-name format. Mangled names are internal compiler names with complete type signatures. To do this, use the following command format.

`-inline` *mangledfunctionname*

To obtain the character string for the mangled name, use the `nm -f` command.

To set inlining using a directive in your C or C++ program, you can add pragma statements that require or prohibit inlining of individual functions. To do this, use one of the following directives.

```
#pragma mta inline
#pragma mta no inline
```

You must place one of the previous directives immediately before the function's definition in your program.

The C++ keyword `inline` also inlines a function, but it makes the function local to the file. In this case, if you also add the function's definition to the header file, multiple inclusions would result in many copies of this function being added to the program library. Therefore, the use of the pragma directive is usually preferable to the C++ keyword.

7.5 Optimizing Parallelization

You can control how the compiler makes your program parallel in two ways:

- You can add parallelization directives to your program.
- You can specify a compiler option from the command line that controls parallelization.

Parallelization directives and options tell the compiler how to parallelize various sections of a program. The following types of parallelization are allowed.

Single-processor parallelism

This form of parallelism has low overhead, but does not allow the program to scale beyond a single processor. This type of parallelization takes advantage of only the streams on the processor on which the code is running.

Multiprocessor parallelism

This form of parallelism uses more memory and has a higher startup time than single-processor parallelism. However, the number of streams available is much larger, being bounded by the number of processors on the entire machine rather than the size of a single processor.

Loop future parallelism

Loop future parallelism runs on multiple processors. It is the highest overhead form of parallelism, but is also the only form of parallelism with the ability to dynamically increase thread and processor counts as needed while the parallel region is executing. It provides good load balancing, especially with recursive loops.

When using a directive, the parallelization type is set using the `#pragma mta parallel` directive. See Parallelization Directives on page 124.

When the parallelism type is set using a compiler option, the following options are available.

`par`	Compiles a program to run in parallel on multiple processors.
`par1`	Compiles a program to run in parallel on a single processor.
`parfuture`	Compiles a program to run on multiple processors using loop future parallelism.
`serial`	Compiles a program to run without automatic parallelization.

Parallelism that you specify with `future` statements in your program is always enabled. Compiler options have no effect on `future` statements. If you do not specify a compiler option, the default is to run using the `par` option.

There are also parallelization directives and compiler options available that you can use to enable or disable loop restructuring. Loop restructuring includes loop transformations, loop fusion, loop unrolling, loop distribution, and loop interchange. By default, loop restructuring is enabled when parallelization is enabled, and disabled otherwise. To enable or disable loop restructuring using a directive, use the `#pragma mta restructure` directive. Disabling loop restructuring may inhibit parallelization of some loops.

The previous directive restructures loops from the point where it appears in the file to the end of the file. It can be disabled during the compilation process when you specify the `-nopar` compiler option from the command line.

You can enable loop restructuring from the command line using the `-restructure` compiler option. You can disable loop restructuring using the `-no_restructure` option. You may need to use this if you are also using the `-par`, `-par1`, or `-parfuture` option, because these options automatically enable loop restructuring.

You can also control whether the compiler automatically parallelizes recurrences and reductions. Recurrence is enabled, by default, but you may want to disable it for a section in the program. To do this, use the `#pragma mta recurrence off` command.

For information about the parallelization options, see the `cc`(1) or `c++`(1) man page. For a complete list and explanation of the parallel directives and assertions, see Appendix C, Compiler Directives and Assertions on page 109.

7.6 Incremental Recompilation and Relinking

When a previously built program library and executable are present, the compiler performs incremental recompilation and relinking, regardless of the compilation mode. An incremental recompilation saves time during the compilation process.

The compiler performs incremental recompilation on a function-by-function basis within each source file. If you repeatedly edit and compile several functions in the `blas.cc` file, the compiler detects which functions require recompilation after editing. For example, if you edit a particular function `f`, the compiler only recompiles `f` and any function that inlined `f`. But if you change a globally-visible type declaration, the compiler recompiles all functions that use that type.

In whole-program mode, separate-module mode, or mixed mode, the compiler builds a program library for the executable. The compiler uses the program library during the incremental compilation. If you delete the `.pl` file between compilations, the compiler cannot execute an incremental recompilation. Similarly, deleting the executable file prevents incremental linking.

7.7 Creating New Libraries

You can create a user-defined library in the same way that you build a program in whole-program mode. To do this, use the `-R` option to suppress the creation of an executable.

For example, to build the library `tinyblas.a` from functions in the files `ddot.cc` and `dgemv.cc`, use the following sequence of commands.

```
c++ -pl tinyblas.a -c ddot.cc dgemv.cc
c++ -pl tinyblas.a -R ddot.o dgemv.o
```

In the previous example, the first command creates the initial program library, checks the two source files for syntax errors, and copies them into the program library. The second command finishes compilation of the functions in `ddot` and `dgemv` with inlining enabled between files and from the standard libraries. The `-R` flag directs the compiler to place the generated relocatable object code in the program library and suppresses the build of an executable.

The following sequence of commands provides the same results:

```
c++ -pl tinyblas.a -c ddot.cc
c++ -pl tinyblas.a -c dgemv.cc
c++ -pl tinyblas.a -R ddot.o dgemv.o
```

Or, you can use the following single command:

```
c++ -pl tinyblas.a -R ddot.cc dgemv.cc
```

You can update a library with an incremental compilation. To do this, use the following sequence of commands.

```
c++ -pl tinyblas.a -R ddot.cc dgemv.cc
edit dgemv.cc
c++ -pl tinyblas.a -R ddot.cc dgemv.cc
```

In the previous example, the first compile creates the library as usual. The second compile examines `ddot.cc` (and ignores it because it remains unchanged) and then focuses on `dgemv.cc` , which has presumably been changed by the edit. The compiler recompiles any modified function in `dgemv.cc` and any function that depends on a changed function (perhaps because of inlining). The rest of the library remains the same.

There is no requirement that a library end with an `.a` suffix. The inclusion of the `-R` flag in a separate-module compilation line enables inlining from the standard libraries into the newly created library. The library looks like a traditional (fat) object file.

7.8 Compiler Messages

There are three categories for compiler messages: errors, warnings, and remarks. Errors are the most severe and indicate problems that cause the compiler to halt after parsing without generating object code. Warnings are less severe — the compiler runs to completion and generates object code. Remarks tend to highlight conditions that prevent the code from being portable, but the resulting object code almost always behaves as expected.

7.9 Setting Debugger Options during Compilation

Rather than providing many levels of optimization, the compiler provides the `-g1` and `-g2` options to support progressing levels of debugging. The debugger options include the following:

`-g`, `-g1`	At this level, the debugger displays the values of variables (including global variables and array elements) anywhere in their scope. However, this level causes some loss of optimization. Specifically, the compiler no longer restructures loops, although basic loop parallelization is still possible. The `-g` flag is identical to `-g1`.
`-g2`	This is the highest level of debugging support. This level lets you view and modify variables anywhere in their scope. However, this level significantly inhibits optimization. Specifically, the compiler no longer parallelizes loops.

If you do not specify either option, the compiler runs with all optimizations enabled. Although debugging is not set, you can still perform some debugging operations. For example, you can control trace control flow using breakpoints together with the `step` and `next` commands. You can also view the value of global variables, although these can sometimes be out-of-date.

The compiler also has options that perform tracing. Tracing creates a trace file, `trace.out`, that you use for performance tuning. You use the `-trace` option to turn on tracing and `-trace_level` *n* to trace functions larger than *n* source lines. You can also trace stack allocation by using the `-trace_stack_alloc` compiler option. For more information about the trace option, see the `cc`(1) or `c++`(1) man pages. For information about performance tuning, see *Cray XMT Performance Tools User's Guide*.

If you compile an executable using modules that have been compiled at different debugging levels, the level of debugging support changes between one module and another, whether inlined or not. For more information about using the Cray XMT debugger, see *Cray XMT Debugger Reference Guide*.

7.10 Using Compiler Directives and Assertions

Directives are metalanguage constructs that you can add to a program to influence how the compiler performs a translation. In C and C++, you prefix directives with `#pragma mta`. Macros are allowed after the word `mta` in a pragma, as shown in this example:

```
#define NUMSTREAMS 40 ...
#pragma mta use NUMSTREAMS streams
```

The preceding pragma is equivalent to `#pragma mta use 40 streams`.

You can also write compiler directives in C and C++ code using `_Pragma` rather than `#pragma mta`. In this case, the directive appears syntactically as if it were a single string argument to a function call, as shown in the following command.

```
_Pragma("mta assert parallel")
```

The advantage to using the command form of this directive is that you can use it in macros or similar locations. The disadvantage of this form is that most C and C++ compilers treat it as an actual function, which makes the code less portable.

Directives are grouped into five general categories: compilation directives, parallelization directives, semantic assertions, implementation hints, and language-extension directives. A compilation directive is a command to compile a program in a particular way. Parallelization directives tell the compiler how to parallelize various sections of a program. Semantic assertions provide information to the compiler that could be proved true about the program even though that proof is beyond the capabilities of the compiler. Implementation hints tell the compiler about the expected behavior of the program. Language-extension directives allow you to place Cray XMT specific language features into a program without interfering with the portability of code to other systems.

For more information, see Appendix C, Compiler Directives and Assertions on page 109.

Running an Application [8]

This chapter contains procedures for launching your application on the Cray XMT.

8.1 Launching the Application

You use the `mtarun` command to launch and run a program. The `mtarun` command connects to the `mtarund` daemon that runs on the compute node on the backend. The daemon creates a copy of your environment and runs it on the compute nodes. Your file directories from the login node appear on the compute nodes with the same paths.

From the login node, you use the `mtarun` command to launch a program, as shown in the following example.

`mtarun` *MyProgram.out*

The most common options to use with the `mtarun` command are `-m max_procs` and `-t min_procs`.

The `-m max_procs` option sets the maximum number of processors for the program. This option is the same as setting the `MTA_PARAMS` environment variable to `NUM_PROCS`.

The `-t min_procs` option sets the number of processors to use when the program starts running. By default, a program starts with one processor and adds processors, as needed.

After launching the program, `mtarun` acts as the frontend of the program. `mtarun` provides the following services to the program:

- Standard I/O forwarding. Provided by `mtarun stdin`, `mtarun stdout` and `mtarun stderr`.
- Signal forwarding. `mtarun` forwards all catchable signals.
- Termination management. If the program exits normally, `mtarun` exits with the same exit status. If the remote process is killed by a signal, `mtarun` terminates with the matching exit status and sends a message to `stderr` with information about the signal that caused the program to exit. If `mtarun` terminates prematurely, the `mtarun` daemon uses `SIGKILL` to kill the program.

The `mtarun` command uses a default configuration file, `.mtarunrc`, which exists in your `home` directory. You can modify this file to include any `mtarun` options, separated by spaces. The configurations in this file are overridden by options that you use from the command line.

To monitor process or CPU usage by your program, you use `mtatop`. For more information about using `mtarun` to run the program or `mtatop` to monitor the program, see *Cray XMT System Management*.

> **Note:** When an application that was built for tracing is running, an intermediate process runs to flush trace data back to the service partition as the tracing buffers fill. To ensure that all tracing data is captured, the `mtarun` that launched the application will not exit until this tracing process completes. Depending on the amount of data that needs to be flushed, and the speed of the underlying file system, `mtarun` may not exit for some time after the application has completed. If you kill the `mtarun` process, in the belief that it is hung, you may get incomplete tracing data. For more information on partial tracing data see *Partial Tracing* in the *Cray XMT Performance Tools User's Guide*.

8.2 User Runtime Environment Variables

There are a number of environment variables that you can use with the user runtime known as `MTA_PARAMS`. You can use these environment variables for debugging, dumping registers, setting the number of streams, setting maximums for processors and ready pools, and so on.

For `csh`, use the following command:

```
% setenv MTA_PARAMS "param1 param2"
```

For example, to set the maximum number of processors and to prevent streams from being reserved for the debugger, set `MTA_PARAMS` by using the following command:

```
% setenv MTA_PARAMS "num_procs 100 no_prereserve"
```

For a `bash` shell, use the following command:

```
% export MTA_PARAMS="param1 param2"
```

For example, to set the maximum number of processors to two and indicate that the program must wait for a debugger to attach in the event of a poison, you use the following command on a `bash` shell:

```
% export MTA_PARAMS="num_procs 2 debug_data_prot"
```

For a list of environment variables that you can set, see Appendix G, `MTA_PARAMS` on page 143.

8.3 Improving Performance

For information about improving performance on your program, see *Cray XMT Performance Tools User's Guide*.

Optional Optimizations [9]

9.1 Scalar Replacement of Aggregates

Effective with version 2.0 of the Cray XMT software, the XMT compiler provides an optional optimization pass that performs a code transformation called *scalar replacement of aggregates*. This transformation replaces C++ class objects and C structures (aggregate data types) with collections of temporary scalar variables. Values are copied from the aggregate to the temporary variables and back again as needed. These scalar variables allow the compiler to perform more precise analysis in later phases, and may enable additional optimizations and parallelization of loops.

For example, consider the following code:

```
class myTwoInts {
public:
  int i;
  int j;
};

myTwoInts foobar2(myTwoInts t, int n, int * restrict foo) {
  for (int i = 0; i < n; i++) {
    t.i += foo[i];
  } return t;
}
```

Without scalar replacement the compiler cannot determine whether the references to fields of the object `t` form a loop-carried dependence, thus it is unable to parallelize this loop. By viewing the `canal` report you can see that the loop is not parallelized:

```
| myTwoInts foobar2(myTwoInts t, int n, int * restrict foo) {
 |   for (int i = 0; i < n; i++){
8S |     t.i += foo[i];
 |   }
 |   return t;
 | }
```

After recompiling this code with automatic scalar replacement enabled, the compiler is able to transform the `foobar2` routine into something that resembles the following:

```
myTwoInts foobar2(myTwoInts t, int n, int * restrict foo) {
   __tmp_t_i = t.i;
   for (int i = 0; i < n; i++) {
     __tmp_t_i += foo[i];
   }
   t.i = __tmp_t_i;
   return t;
}
```

Note that the compiler does not bother creating a temporary variable for the unused field `j`.

After this transformation, the compiler is better able to analyze the dependencies in the loop and to determine that the loop can be safely parallelized as a reduction. This can be seen in the `canal` report of the recompiled code:

```
   | myTwoInts foobar2(myTwoInts t, int n, int * restrict foo) {
** scalar replacing t
   | for (int i = 0; i < n; i++) { 18 P:$
18 P:$ | t.i += foo[i];
** reduction moved out of 1 loop
   | }
   | return t;
   | }
```

Scalar replacement of aggregates can enable parallelization of many additional loops. However, it can also add additional memory references which can adversely affect performance. For this reason, the compiler performs scalar replacement only when requested by the programmer. Automatic scalar replacement of aggregates can be enabled either by using a command-line flag at compile time, or by using pragmas in your code. If you compile a file with the `-scalar_replacement` flag, the compiler will automatically attempt to perform scalar replacement on any aggregates that it can prove are safely replaceable unless those aggregates have been marked with an `mta no replace` pragma. (See Semantic Assertions on page 125.) You can use the `noalias` pragmas and `restrict` type qualifiers as needed to indicate to the compiler that certain aggregates, or pointers to aggregates, are safe to replace.

Alternatively, you can enable scalar replacement for individual aggregates by using the `mta assert can replace` pragma. This pragma, which takes a list of aggregates and/or aggregate pointers, serves two purposes. First, it tells the compiler that it is safe to perform scalar replacement on the aggregates or pointers listed. The compiler follows this assertion even if it was unable to prove that the replacement was safe. Second, it is a request to replace the listed aggregates even if the code was not compiled with the `-scalar_replacement` flag. This pragma is useful in situations where the compiler would not be able to verify that a key aggregate is replaceable. You can also use this pragma in situations where, because of the extra memory references, you do not want to enable scalar replacement for an entire source file, but where you need a particular aggregate to be replaced in order to achieve automatic loop parallelization.

For example, consider the loop in the method `doit` below:

```
class foo {
   int * restrict b;
   int n;

public:

#pragma mta no inline
  void doit(int *c) {
    int i;

#pragma mta assert noalias *this
   for (i = 1; i < n; ++i) {
     b[i] = b[i-1] + c[i-1];
   }
  };
};
```

Without scalar replacement, this parallel recurrence loop will not parallelize, because the accesses to the `b` array, which are accesses into a field of the aggregate `*this`, defy alias analysis. By adding an `mta assert can replace` pragma, however, the loop will parallelize as can be seen in the `canal` report:

```
  | #pragma mta no inline
  |   void doit(int *c) {
** scalar replacing *this
  |    int i;
  |
  | #pragma mta assert noalias *this
  | #pragma mta assert can replace *this
  |     for (i = 1; i < n; ++i) {
5 L  |       b[i] = b[i-1] + c[i-1];
  |     }
  |   };
  | };
```

The `can replace` assertion also has a loop- specific variant, `mta assert loop can replace`, which requests scalar replacement for a specific loop instead of an entire function. In this case we copy into the temporaries immediately before the loop, and copy back into the aggregate immediately after the loop. Any accesses to fields of the aggregate inside the loop will be replaced with the temporaries. This can be useful if scalar replacement is unsafe or undesirable for portions of a routine, but needed to achieve good performance in specific loops. The loop variant can also be used to achieve parallelization of the loop in the previous example:

```
  | #pragma mta no inline
  |   void doit(int *c) {
  |     int i;
  |
  | #pragma mta assert noalias *this
  | #pragma mta assert loop can replace *this
  |     for (i = 1; i < n; ++i) {
5 L |       b[i] = b[i-1] + c[i-1];
** scalar replacing *this
  |     }
  |   };
  | };
```

The exact syntax of these pragmas is described in Appendix C.3 of *Cray XMT Programming Environment User's Guide*.

9.2 Optimizing Calls to `memcpy` and `memset`

The compiler option `-enable_memcmd_opt` enables a compiler optimization that replaces calls to `memcpy`/`memset` with versions of the functions that were built for the current parallel mode, which the compiler can inline. This allows the compiler to potentially merge the parallel region in the memory routine with any surrounding parallel region, which can reduce the cost of having to tear down and restart parallel regions in order to call `memcpy` or `memset`. However, when this optimization is enabled and these functions are called from within a parallel loop, this creates nested parallel regions. The result is a potentially significant performance degradation.

A new compiler flag, `-disable_memcmd_opt` was added to disable this optimization in case there were performance problems, such as the case mentioned above. However, because the functions may be getting called indirectly, it may not always be easy to determine that a call to `memcpy` or `memset` is causing a performance problem. For example, this can happen is if a program calls a function in the C++ STL that calls `memcpy`. For this reason, the default behavior of the compiler is to have this optimization disabled and allow users to enable it with the option `-enable_memcmd_opt`. Use this option **only** when you know there is no risk of `memcpy` or `memset` being called from within a parallel loop.

For additional control over the parallelism used by `memcpy` or `memset`, you can call directly versions of of these commands that use a single stream, single processor parallelism and multiprocessor parallelism. The `memcpy` functions are called `memcpy_ss`, `memcpy_sp` and `memcpy_mp`, respectively. The corresponding `memset` functions are called `memset_ss`, `memset_sp` and `memset_mp`, respectively. These functions are declared in `string.h` and are documented in the `memcpy`(3) and `memset`(3) man pages.

Error Messages [A]

Execution-time errors are directly related to exceptions. An exception is an unexpected condition raised by an event in your program, the operating system, or the hardware. Exceptions can trigger a trap when the stream that issued the exception is ready for execution, unless the trap is disabled. In cases where several exceptions occur simultaneously, the trap handler decides the order in which to process the exceptions.

Use the list that follows to identify and troubleshoot common exceptions.

`create` For example, this error will occur when you attempt to create more streams than were reserved. To prevent this error, you can use the `STREAM_RESERVE` operation to reserve the necessary number of streams before running the `STREAM_CREATE` operation again.

`data_alignment`

A data-alignment error has occurred. This error can occur when you access data that the compiler assumes is on an 8-byte boundary when it is not.

`data_hw_error`

A data-memory or network-hardware error has occurred. This occurs when the memory system detects an uncorrectable error while loading data from memory.

`data_prot` A data protection level error has occurred. This error is equivalent to a segmentation error. Possible causes include attempting to access protected data, operating-system data, or data outside your addressable memory space.

`domain_signal`

A domain signal error has occurred. This message indicates the program is not allowing the operating system to interrupt it. This typically indicates a problem in the runtime system.

`float_extension`

An error using a floating-point number has occurred. A floating-point number is using the wrong extension.

`float_inexact`

An error using a floating-point number has occurred. An operation is attempting to use an inexact floating-point number. This type of error indicates an error in the source registers, the operation, or the value written to the destination.

`float_invalid`

An error using a floating-point number has occurred. An operation is attempting to use an invalid floating-point number.

`float_zero_divide`

An error using a floating-point number has occurred. An operation is attempting to divide a floating-point number by 0.

`float_overflow`

An error using a floating-point number has occurred. An operation using a floating-point number has caused an overflow to occur. This type of error indicates an error in the source registers, the operation, or the value written to the destination.

`float_underflow`

An error using a floating-point number has occurred. An operation using a floating-point number has caused an underflow to occur. This type of error indicates an error in the source registers, the operation, or the value written to the destination.

`poison` Use of a *poisoned* register has occurred. A register is poisoned if it contains an uninitialized value. The exception occurs when you attempt to access the value in this register.

Use of a poisoned register can sometimes occur when the compiler uses *speculative loading*. For example, the compiler may optimize a loop for `n` iterations and load `n+1` values. Under normal conditions, the compiler does not use the `n+1` value because the program correctly stops consuming prefetched data after `n` iterations. However, if the program accesses the `n+1` value, it raises the poison exception.

`privileged` A privilege error has occurred. This exception indicates that your program does not have the necessary privilege level to perform an operation.

`prog_hw_error`

A program-memory error has occurred. This indicates that while the processor was loading an instruction, there was a temporary or permanent problem with the physical memory.

`prog_prot` A program-protection error has occurred. This error occurs when the processor attempts to execute an instruction from a PC that is not a valid PC.

`unknown_trap`

A error has occurred that does not fit into any other category on this list.

User Runtime Functions [B]

Functions in the runtime library support implicit and explicit parallelism, event logging, and trap handling. The compiler inserts calls to the runtime library into your code to handle programming constructs, such as the `future` statement, or command-line options, such as the `-trace` flag. In addition, some functions in the runtime library can be called directly by the user. This appendix contains a list of the runtime functions that you can call from your program.

This list provides only a short description of the runtime functions. A more complete description of the functions and the syntax required to use them can be found on the referenced man pages.

`mta_create_team`

Adds teams. See the `mta_create_team`(3) man page.

`mta_create_thread_on_team`
`mta_create_thread_all_teams`
`mta_create_stream`

Creates a new thread on an existing team. See the `mta_create_thread_all_teams`(3) man page.

`mta_disable_auto_growth`
`mta_enable_auto_growth`
`mta_assess_growth`

Controls the automatic growth of processors. See the `mta_disable_auto_growth`(3) man page.

`mta_get_all_rt_teamids`

Returns the team identifiers for all runtime teams. See the `mta_get_all_rt_teamids`(3) man page.

`mta_get_clock`

Provides the number of clock ticks that have passed since the program began. See the `mta_get_clock`(3) man page.

`mta_get_max_teams`

Determines the maximum number of teams available to the program. See the `mta_get_max_teams`(3) man page.

`mta_get_num_teams`

Returns the number of currently executing teams. See the `mta_get_num_teams`(3) man page.

`mta_get_rt_teamid`

Returns the runtime identifier of the caller's team. See the `mta_get_rt_teamid`(3) man page.

`mta_get_team_index`

Returns a user runtime index for a team. See the `mta_get_team_index`(3) man page.

`mta_get_thread_name`
`mta_set_thread_name`
`mta_remove_thread_name`

Retrieves, sets, and removes user-defined thread names. See the `mta_get_thread_name`(3) man page.

`mta_get_threadid`
`mta_get_parent_threadid`

Returns the runtime identifier of the calling thread or its parent thread. See the `mta_get_threadid`(3) man page.

`mta_lock_thread`
`mta_unlock_thread`

Controls thread behavior when a synchronized data fault occurs. See the `mta_lock_thread`(3) man page.

`mta_log_event`
`mta_log_short_event`
`mta_log_long_event`
`mta_log_event_record`
`mta_log_short_event_record`
`mta_log_long_event_record`

Sets user-defined event logging. See the `mta_log_event`(3) man page.

`mta_new_trap1_continuation`
`mta_new_trap1_continuation_block`
`mta_delete_trap1_continuation`
`mta_register_trap1_continuation`
`mta_unregister_trap1_continuation`
`mta_update_trap1_value`

Creates, deletes, binds, or updates trap 1 continuation. See the `mta_new_trap1_continuation`(3) man page.

`mta_print_backtrace`

Prints the thread's call stack. See the `mta_print_backtrace`(3) man page.

`mta_probe_location`

Probes a memory location to determine whether it can be read or written. See the `mta_probe_location`(3) man page.

`mta_register_event_filter`

Installs a filter function for user-defined event logging. See the `mta_register_event_filter`(3) man page.

`mta_register_fatal_error_handler`

Binds a new fatal error handler. See the `mta_register_fatal_error_handler`(3) man page.

`mta_register_task_data`

Stores thread-specific data used to implement a common task. See the `mta_register_task_data`(3) man page.

`mta_register_team_exit_fn`
`mta_unregister_team_exit_fn`

Binds or unbinds a team exit function. See the `mta_register_team_exit_fn`(3) man page.

`mta_register_tertiary_handler`
`mta_get_tertiary_handler`

Binds a new tertiary trap handler or return the current tertiary trap handler. See the `mta_register_tertiary_handler`(3) man page.

`mta_report_trap_counters`

Sets reporting for trap counter statistics. See the `mta_report_trap_counters`(3) man page.

`mta_reserve_task_event_counter`
`mta_get_task_counter`
`mta_get_team_counter`

Reserves or queries hardware counters. See the `mta_reserve_task_event_counter`(3) man page.

`mta_set_crew_limit`

Sets the maximum number of crews that can be simultaneously active. The term crew is applied to the group of processors that are used when parallelizing the iterations of a loop across multiple processors. Applications use this type of parallelization when they are compiled using the multiprocessor mode. See the `mta_set_crew_limit`(3) man page.

`mta_set_domain_signal_mask`

Enables or disables domain signals in the calling thread. See the `mta_set_domain_signal_mask`(3) man page.

`mta_set_implicit_processors`
`mta_get_implicit_processors`
`mta_set_implicit_streams`
`mta_get_implicit_streams`

Stores or retrieves the value for the number of implicit processors or implicit streams that are used for a calling thread for an implicitly parallelized region of code in a program. See the `mta_set_implicit_processors`(3) man page.

`mta_set_private_data`
`mta_get_private_data`

Stores or retrieves private data for a thread. See the `mta_set_private_data`(3) man page.

`mta_set_rt_error_file`

Redirects runtime library messages to a file. See the `mta_set_rt_error_file`(3) man page.

`mta_set_trace_limit`

Modifies the number of times an individual trace event is recorded. See the `mta_set_trace_limit`(3) man page.

`mta_sleep`

Suspends a thread. See the `mta_sleep`(3) man page.

`mta_start_event_logging`
`mta_suspend_event_logging`
`mta_resume_event_logging`
`mta_is_event_logging_on`
`mta_set_event_flush`

Traces buffer controls for user-defined event logging. See the `mta_start_event_logging`(3) man page.

`mta_yield`

Yields an active stream to any other thread that needs the stream. See the `mta_yield`(3) man page.

Compiler Directives and Assertions [C]

This appendix provides a complete list of compiler directives specific to the Cray XMT and accepted by the Cray XMT compiler.

C.1 Compilation Directives

A compilation directive is a command to compile a program in a particular way.

`#pragma mta autotouch [on|off|default]`

This directive automatically applies the `touch` generic whenever a future variable is referenced. The `on` option enables automatic touching, the `off` option disables automatic touching, and the `default` option reverts from autotouch to the default mode for that source module, as determined by the compile-line flags.

`#pragma mta adjust constructor priority` *adj*

This directive modifies the priority assigned to static constructors in a file. The adjusted priority is the priority just before the directive plus *adj*. The adjustment variable *adj* must be an integer in the range of `-255` to `255`, and the new priority must be in the range of `0` to `255`. This directive remains in effect from the point at which it occurs until the end of the file or until another directive of the same kind is encountered.

`#pragma mta complex limited range [on|off|default]`

This directive specifies whether complex multiplication and division may be performed using the usual mathematical formulas for complex arithmetic or safer but slower arithmetic. The usual mathematical formulas for complex arithmetic use the following format:

```
(a,b)*(c,d) = (ac-bd,ad+bc)
(a,b)/(c,d) = ((ac+bd)/(cc+dd), (bc-ad)/(cc+dd))
```

The previous formulas, however, may cause spurious Not a Number (NaN) results or infinities if the norm of either complex number is larger than the maximum expressible real number or if the norm of the denominator of a division is smaller than the smallest expressible real number. Additionally, these formulas may not be as accurate as the safer complex arithmetic performed when complex limited range is `off`. This is especially true when the difference between two intermediate computations is very small, such as `ac-bd`, in the case of multiplication, and `bc-ad`, in the case of division.

This directive applies to whatever follows it textually in the current file. The directive stays in effect until the end of the file or until another directive of the same kind is encountered. When the `on` or `off` options are used, the directive takes precedence over the `-cxlimited` and `-no_cxlimited` command-line options. When the `default` option is used, the directive enables the faster arithmetic if `-cxlimited` is specified on the command line. Otherwise, it disables the faster arithmetic.

`#pragma mta constructor priority` *pri*

This directive assigns a priority level of *pri* to the static constructors within the file, where *pri* is an integer in the range 0 to 255. This priority determines the treatment of constructors using the following rules:

- Static constructors with priority `j` are executed before those of priority i, for i < j. No order is promised between modules compiled with the same constructor priority.
- Static constructors with priority less than 200 are executed after the user runtime has been initialized. In particular, futures and system calls may be performed reliably by static constructors with priority less than 200.
- Static constructors with priority less than 100 are executed after the system libraries have been initialized. For example, input/output operations may be reliably performed by static constructors with priority less than 100.

The `constructor priority` directive overrides any `-constructor_priority` *n* compiler flag used on the command line. If neither the directive nor the compiler flag is used, the constructor priority defaults to 0. The `constructor priority` directive may occur at any point in a source code file provided no `constructor priority` or `adjust constructor priority` directives occur at an earlier point in the same file. The directive remains in effect from the point at which it occurs until the end of the file or until an `adjust constructor priority` directive is encountered.

#pragma mta debug level [0|1|2|default|none]

Set the debug level to the integer constant 0, 1, or 2, or to no debugging by specifying `none`. Or, set the debug level back to the level provided on the command line by specifying `default`. This directive overrides the `-g` , `-g1` , and `-g2` compiler flags. However, this directive does not affect any function that contains a call to `setjmp` or `sigsetjmp`, which is always compiled as if the `-g2` option was specified. This directive has function-level granularity and affects any functions whose beginning follows the directive. This directive applies to whatever follows it textually in the current file. It stays in effect until the end of the file or until another directive of the same kind is encountered.

#pragma mta fence

This directive specifies a boundary in the source code across which the compiler is not allowed to move loads or stores of any aggregate or heap allocated variables. The effect of this directive is to limit the compiler's ability to move statements that have been marked with a `fence` directive. This directive is often used to prevent the compiler from moving calls to timing functions with respect to the code being timed, as in the following example.

```
#pragma mta fence
t0 = mta_get_clock(0);
/* interval of interest */
......
#pragma mta fence
t1 = mta_get_clock(t0);
```

This directive may prevent some compiler optimizations from being performed.

`#pragma mta fenv_access [on|off|default]`

This directive specifies whether the full floating-point environment is available. When `fenv_access` is on, strict rules against the optimization of floating-point operations are enforced. If it is off, extra optimizations are performed, but floating-point exceptions may be lost in certain cases. The compiler is allowed to attempt either one or both of two optimization techniques when `fenv_access` is off. The first technique is to evaluate floating-point operations at compile time. The second is to move floating-point operations to locations where they are executed with less frequency, such as outside a loop. In the following example, the addition in the statement that assigns a value to `G` can be performed at compile-time, but the addition in the statement that assigns a value to `F` cannot.

```
void sub(void) {
  float F;
  float G;
#pragma mta fenv_access off
  G = 2.5 + 3.1;
#pragma mta fenv_access on
  F = 2.5 + 3.1;
}
```

This directive applies to whatever follows it textually in the current file. The directive stays in effect until the end of the file or until another directive of the same kind is encountered. The `off` and `on` options to the `fenv_access` directive takes precedence over the `-no_float_opt` command-line option. The `default` option to the directive enables floating-point environment access (disables floating-point optimization) if the `-no_float_opt` command-line option was used. `Default` disables floating-point environment access (enables optimization) if the command-line option was not used. The directive may also be specified in C as `#pragma fenv_access [on|off|default]`.

`#pragma mta for all streams`

This directive starts up a parallel region (if the code is not already in a parallel region) and cause the next statement or block of statements to be executed exactly once on every stream allocated to the region. If the pragmas appear in code that would otherwise not be parallel, they cause it to go parallel.

You can use this pragma in conjunction with the `use` *n* `streams` to ask the compiler to allocate a certain number of streams per processor to the job.

```
#pragma mta use 100 streams
#pragma mta for all streams
{     // do something
}
```

However, there is no guarantee that the runtime will grant the requested number of streams if, for example, they are not available due to other jobs, the OS, or other simultaneous parallel regions in the current job.

`#pragma mta for all streams` *i* `of` *n*

This directive is similar to the `for all streams` pragma except that it also sets the variable *n* to the total number of streams executing the region, and the variable *i* to a unique per-stream identifier between `0` and *n*`-1`. For example:

```
int i, n;
int check_in_array[MAX_PROCESSORS * MAX_STREAMS_PER_PROCESSOR];
for (int i = 0; i < MAX_PROCESSORS * MAX_STREAMS_PER_PROCESSOR; i++)
  check_in_array[i] = 0;


#pragma mta for all streams i of n
{
  check_in_array[i] = 1;
  printf("Stream %d of %d checked in.\n", i, n);
}
```

Note that the integer variables *i* and *n* are declared separately from the pragma. For more information on the `for all streams` pragmas see *Using the Cray XMT for all streams Pragmas* in the CrayDoc Knowledge Base at http://docs.cray.com/kbase.

`#pragma mta fused muladd [on|off|default]`

This directive specifies whether the compiler is allowed to combine floating-point operations into a fused multiply-add operation. Default behavior is to allow fused multiply-add operations to be performed only when float optimization is turned on. When this option is turned on, the compiler is allowed to, but not required to, fuse multiply-add operations into one instruction. This directive applies to whatever follows it textually in the current file. The directive stays in effect until the end of the file or until another directive of the same kind is encountered. When the `on` or `off` option is used, the directive takes precedence over the `-no_mul_add` command-line option. When the `default` option is used, the directive disables the fused multiply-add operation if the `-no_mul_add` command-line option was used; it enables the fused multiply-add operation if no command-line option was used. The `single round required` directive overrides the `fused muladd off` directive.

`#ident "<string-constant>"`

This directive inserts `string-constant` into the executable file generated from this code. Strings that have been incorporated into the executable in this manner can be retrieved from the executable using commands such as `strings` or in some cases `what`. One possible use of this directive would be to incorporate a version string such as the following into the executable.

```
#ident "compiling.texinfo,v 1.15 2007/02/10 23:20:09"
```

This directive can be placed anywhere in a C file and is the equivalent to declaring a static string constant.

`#pragma mta [no] inline`

When this directive is inserted immediately before a function declaration, the compiler inlines that function wherever possible throughout the user source program. If used with the `no` option, inlining of the specified function is prevented. When the `[no] inline` directive is not used, the compiler uses a standard, internal heuristic to decide whether a function should be inlined. When there is a conflict between the `no inline` directive and the command-line options `-no_inline_all`, `-inline_all`, `-inline <name>` or `-no_inline <name>`, `no inline` takes precedence, regardless of whether it was specified on the command line or in a directive. The command-line option `-no_inline_directed` disables the `inline` directive but does not affect the `no inline` directive.

`#pragma mta instantiate [none|all|used|local|default]`

When used inside a template declaration, the effect of this directive is limited to the uses of that template. When used outside a template declaration, this directive sets the template instantiation mode for the text following the directive and stays in effect until the end of the file or until another directive of the same kind is encountered. This directive takes one of the following options:

`none`	No instantiations are created for any template entities.
`used`	All template entities that were used in the compilation, including all static data members for which there are template definitions, are instantiated.
`all`	All template entities that are declared or referenced in the compilation unit are instantiated. For each fully instantiated template class, all of its member functions and static data members are instantiated, whether used or not. Nonmember template functions are instantiated even if the reference was only a declaration.
`local`	Those template entities that were used in the compilation are instantiated. This option is similar to the `used` option, except that in this case, the functions are given internal linkages. That is, the compiler instantiates the functions and static data members used in the compilation as local static functions and local static variables.
`default`	The instantiate mode switches back to either the mode specified by the `-instantiate` switch on the compiler command line, or, if no command line switch was present, to the `none` option, which is the default behavior when no mode is specified.

Where the mode specified with the `instantiate` pragma differs from that specified with the `-instantiate` switch on the compiler command line, the `instantiate` pragma takes precedence.

`#pragma mta max concurrency` *c*

The `max concurrency` *c* directive indicates that the next loop should limit the concurrency to *c*. This directive can be used on any parallel loop. For single processor parallel loops, the directive limits the number of streams used by the parallel loop to no more than *c*. For multiprocessor parallel loops, the directive estimates the number of processors to use for the loop to `max(1,`*c/num_streams*`)`, where *num_streams* is the number of streams the compiler requests for each processor. For loop future parallel loops, the directive limits to *c* the number of futures created. The directive is ignored for explicityly serial loops and cannot be used on a loop that also uses the `use` *n* `streams` directive. This directive is useful for managing nested parallelism in application that have multiple parallel loops running concurrently, and to reduce or prevent contention for resources. For more information on using this pragma see *Limiting Loop Parallelism in Cray XMT Applications* in the CrayDoc Knowledge Base at http://docs.cray.com/kbase.

`#pragma mta max` *n* `processors`

The `max` *n* `processors` pragma limits the number of processors used by a multiprocessor parallel loop. This is useful for load balancing in applications that have multiple parallel loops running concurrently. For more information on using this pragma see *Limiting Loop Parallelism in Cray XMT Applications* in the CrayDoc Knowledge Base at http://docs.cray.com/kbase.

`#pragma mta max` *n* `streams per processor [may merge]`

This directive sets a limit of *n* on the number of streams per processor that will execute a parallel loop. This limit applies to an entire parallel region. Thus, by default, the compiler will not combine loops with different maximum stream specifications into the same region. This includes cases where one loop has a specified maximum and the other loop does not. However, if you add the optional `may merge` parameter, the compiler will ignore maximum stream specifications when deciding how to construct parallel regions (i.e., loops that would have been placed in the same region with no max streams pragma will still be placed in the same region if max streams pragmas with may merge are added). You can view how parallel regions are constructed in the canal report (see the Cray XMT Performance Tools User's Guide). For example, consider the following two loops:

```
for (int i = 0; i < size_foobar; i++) {
  bar[i] = size_foobar - i;
}

for (int i = 0; i < size_foobar; i++) {
  foo[i] += bar[i]/2;
}
```

The output from `canal` shows that they are both placed into parallel region 1:

```
   | for (int i = 0; i < size_foobar; i++) {
3 P | bar[i] = size_foobar - i;
    | }
    |
    | for (int i = 0; i < size_foobar; i++) {
5 P | foo[i] += bar[i+c]/2;
    | }
...
Parallel region 1 in main
...
Loop 2 in main in region 1
...
Loop 3 in main at line 4 in loop 2
...
Loop 4 in main in region 1
...
Loop 5 in main at line 8 in loop 4
```

If you add a `max streams` pragma to one of the loops, they are no longer placed in the same region:

```
    | for (int i = 0; i < size_foobar; i++) {
3 P |  bar[i] = size_foobar - i;
    | }
    |
    | #pragma mta max 50 streams per processor
    | for (int i = 0; i < size_foobar; i++) {
6 P |   foo[i] += bar[i+c]/2;
    | }
 ...
Parallel region 1 in main
...
Loop 2 in main in region 1
...
Loop 3 in main at line 4 in loop 2
...
Parallel region 4 in main
       Using max 50 streams per processor
...
Loop 5 in main in region 4
...
Loop 6 in main at line 9 in loop 5
```

Notice that `canal` also tells us that the requested maximum was applied to region 4, which is the region that contains the loop with the `max streams` pragma.

However, when you add the `may merge` option these two loops remain in the same region:

```
    | for (int i = 0; i < size_foobar; i++) {
3 P |  bar[i] = size_foobar - i;
    | }
    |
    | #pragma mta max 50 streams per processor may merge
    | for (int i = 0; i < size_foobar; i++) {
5 P |   foo[i] += bar[i+c]/2;
    | }
 ...
Parallel region 1 in main
       Using max 50 streams per processor
...
Loop 2 in main in region 1
...
Loop 3 in main at line 4 in loop 2
...
Loop 4 in main in region 1
...
Loop 5 in main at line 9 in loop 4
```

Note that the compiler has placed both loops into the same region and that the stream limit was applied to the entire region. If multiple limits are specified for the same region the compiler uses the smallest limit.

Two restrictions apply to the use of this pragma:

- You cannot use this pragma with loop future loops.
- If this pragma is used within the same region as a `use` *n* `streams` pragma with a conflicting value (for example a `use` value that is higher than the `max` value) the `max` *n* `streams per processor` pragma will take precedence over the `use` *n* `streams` pragma.

`#pragma no mem init`

This directive affects only the declaration statement immediately following the directive and tells the compiler not to specially initialize the full/empty bit (or bits) of any sync- or future-qualified variables defined in that declaration statement. The directive affects only the definition of variables, including class instance variables; it may not be used on field declarations inside classes. For example:

```
struct C
{/* note that a '#pragma mta no mem init'would be ineffective here */
  sync int k;
};
main() {
  #pragma mta no mem init
  static C c;
/* use the pragma on the instance of the class rather
than on the class definition */
}
```

When the `no mem init` directive is not used, the compiler initializes the full/empty bit of a sync-qualified variable to full if the variable itself is initialized or to empty if the variable itself is not initialized. When the `no mem init` directive is used immediately before a declaration statement, the full/empty bits for any variables defined in that declaration are initialized to full if the variable itself is initialized. If the variable itself is not initialized, the initial state of the full/empty bit is undefined (although, in practice, uninitialized variables stored as static or global variables end up with their full/empty bit initialized to full.) For example:

```
/* full-empty bit is set to full for a[0] and empty for a[1].*/
sync int a[2]={0};
#pragma mta no mem init
/*full-empty bit is set to full for b[0] and is undefined for b[1].*/
sync int b[2]={0};
main(){}
```

#pragma mta no scalar expansion

This directive instructs the compiler not to expand scalar variables to vector temporaries in the next loop. Such expansion allows you to distribute the loop to enhance available parallelism or make effective use of registers. However, if the loop iterates only a few times, the increase in memory usage for the expansion may outweigh the benefits. In this case, you can use the `no scalar expansion` pragma to prevent expansion. For example, in the following code, the use of `no scalar expansion` ensures that the definition of `T` and its use remain in the same loop.

```
void no_scalar_example(double X[], const int N)
{
  extern double Y[], Z[];
#pragma mta no scalar expansion
  for (int i = 0; i < N; i++) {
    const double T = Y[i*2];
    X[i] = T + Z[i*3];
  }
}
```

#pragma mta once

This directive, when placed inside an included file, instructs the C preprocessor to include this file only once in any single compilation unit regardless of the number of `#include` directives encountered. In the following example, the file `foo.h` is included in the file `foo.c` one time only.

```
file foo.h:
#pragma mta once
int i;

file foo.c:
#include "foo.h"
#include "foo.h"
#include "foo.h"

main() {

}
```

This directive may also be specified as `#pragma once`. The directive may occur at any point in the file to be included.

#pragma mta single round required

This directive specifies that the compiler generate a fused multiply-add instruction for every expression (or subexpression) of the form X + Y*Z, X - Y*Z, or Y*Z - X. This selection can be ambiguous, as shown in the following:

```
A = B*C + D*E
```

In this case, the compiler is forced to choose one of two possible implementations. To avoid ambiguity when control of rounding is important, you should use a sequence of simpler assignments to make the meaning clear. The scope of this directive is the entire source file. The use of this directive overrides the `-no_mul_add` compiler flag and the `#pragma mta fused muladd off` directive.

`#pragma mta trace [on|off|default]`

Enables or disables tracing of functions or returns to the default heuristic if `trace default` is used. In order to actually use the tracing information, however, a compiler flag must be set. By default, a heuristic is used to decide whether to trace a function based upon its size. This directive remains in effect until end-of-file or until overridden by another directive of the same type. This directive affects any function whose beginning follows the directive textually in the current file.

`#pragma mta trace level [`*int-const*`]`

This directive enables the tracing of functions that contain at least *int-const* lines, and disables the tracing of functions that contain fewer lines. This directive is disabled unless either the `-trace` or `-trace_level` option was specified on the command line. But after it is enabled, this directive takes precedence over the `-trace` and `-trace_level` command-line options. This directive remains in effect until end-of-file or until overridden by another directive of the same type. This directive affects any function whose beginning follows the directive textually in the current file.

`#pragma mta trace "<`*string-name*`>"`

This directive generates a user-defined tracepoint in the executable code. The tracepoint generated is named the value passed in *string-name*. Using the `-notrace` option on the compiler command line causes this directive to be ignored. For more information, see *Cray XMT Performance Tools User's Guide*.

`#pragma mta update`

This directive tells the compiler that the next statement is an update to a variable, and that the update should be done atomically. By default, the compiler does not necessarily make updates atomic. Using this directive does not place any restrictions on code movement around this update statement such as would occur if the variable were declared to be a sync-qualified variable. The variable to be updated may be of any simple arithmetic or logical type. The variable to be updated must occur as the target on the left side of the statement and must occur exactly once as a subexpression on the right side of the statement. For example,

```
void update_example(double A[], int i, int j){
  extern double V;
  extern double X;

// This is allowed
#pragma mta update
  V = 1.0 + X + 3.0*V;

// This is allowed
#pragma mta update
  A[i] = A[i] + A[j];

// But this is not allowed
#pragma mta update
  A[i] = A[i] + A[i]; // compiler reports an error
}
```

This directive applies to the next statement only.

The following four directives control how the compiler parallelizes the loop that immediately follows.

`#pragma mta block schedule`

When this directive appears before a loop that the compiler parallelizes, each thread assigned to the execution of the loop performs a contiguous subset of the total iterations. Each thread executes the same number of iterations, within 1. For example, if 100 iterations are performed by 20 threads, the first thread executes the first 5 iterations of the loop, the second thread executes the next 5 iterations, and so forth.

`#pragma mta block dynamic schedule`

This scheduling method combines aspects of both block and dynamic scheduling. At execution time, threads are assigned one block of iterations at a time through the use of a shared counter. After completing an assigned block, each thread receives its next block by accessing the counter. The number of blocks executed by each thread depends on the execution time of the particular iterations in the blocks assigned to the thread.

`#pragma mta interleave schedule`

When this directive appears before a loop that the compiler parallelizes, each thread assigned to the execution of the loop performs a subsequence of the total iterations, where the members of the subsequence are regularly spaced. Each thread executes the same number of iterations, within 1. For example, if 100 iterations are performed by 20 threads, the first thread executes iteration 1, iteration 21, iteration 41, and so forth. This scheduling leads to better load balancing for triangular loops. For example:

```
void interleave_example(const double X[100][100],
  const double Y[100], double Z[100], const int N)
{
#pragma mta interleave schedule
  for (int i = 0; i < N; i++) {
    double sum = 0.0;
    for (int j = 0; j < i; j++) {
      sum += X[i][j] * Y[j];
    }
    Z[i] = sum;
  }
}
```

Here, a block schedule results in poor load balancing with the first threads finishing before the last threads. With an interleaved schedule, the work is much better balanced.

`#pragma mta dynamic schedule`

At execution time, threads are assigned one iteration at a time through the use of a shared counter. After completing an assigned iteration, each thread receives its next iteration by accessing the counter. The number of iterations executed by each thread depends on the execution time of the particular iterations assigned to the thread. One thread may happen to receive all the long-running iterations, and thus might execute fewer iterations than any other thread. This method is preferred when the execution time for individual iterations may vary greatly, although its overhead makes it less desirable for general use.

`#pragma mta use` *n* `streams`

This directive indicates that the compiler should request at least *n* threads per processor for the next loop. When multiple loops are contained in the same parallel region, the largest *n* is used. In the absence of a directive, the compiler determines the number of threads needed to saturate the processor. This directive affects the next loop only.

C.2 Parallelization Directives

The compiler recognizes the following parallelization directives.

`#pragma mta parallel [on|off|default| single processor|multiprocessor|future]`

This directive enables or disables automatic generation of parallel code for a section of the program as well as choosing the form of parallelism to use. The `single processor`, `multiprocessor`, and `future` flags indicate the type of parallelism to use. The `off` flag turns off parallelism until it is turned back on or reaches the end of the file. The `on` flag turns on parallel-code generation using the last specified form of parallelism. The `default` flag uses the command-line option or the default form of parallelism. By default, automatic generation of multiprocessor parallel code is enabled. This directive applies to whatever follows it textually in the current file. It stays in effect until the end of the file or until another directive of the same kind is encountered. The directive is ignored if the `-nopar` flag is used on the command line.

`#pragma mta recurrence [on|off|default]`

This directive enables/disables automatic parallelization of recurrences and reductions. By default, recurrence-relation parallelization is enabled. Recurrence relations are parallelized, however, only in areas in which parallelization is otherwise allowed. This directive applies to whatever follows it textually in the current file. It stays in effect until the end of the file or until another directive of the same kind is encountered. The directive is ignored if the `-nopar` flag is used on the command line.

`#pragma mta restructure [on|off|default]`

This directive enables/disables loop restructuring and loop transformations. By default, loop restructuring is allowed in areas in which parallelization is allowed and it is turned off in areas in which parallelization is not allowed. This directive applies to whatever follows it textually in the current file. It stays in effect until the end of the file or until another directive of the same kind is encountered. The directive is ignored if the `-nopar` flag is used on the command line.

`#pragma mta loop loop_mod[, loop_mod, ...]`

This directive takes a comma-separated list of parallelization modes, `loop_mod`, consisting of no more than one selection from each of the following sets of possible loop modes:

`restructure, norestructure`

Enables/disables loop restructuring.

`recurrence, norecurrence`

Allows/disallows automatic parallel processing of recurrences.

`single processor, multiprocessor, future, serial`

Enables either a single or multiple processor or a future form of parallelism or disables parallelism.

This directive enables the appropriate parallelization mode (or modes) for the next loop only. It is ignored if the `-nopar` flag is used on the command line.

`#pragma mta serial`

This directive disables parallelization for a section of the program. It is equivalent to the `parallel off` directive. It is ignored if the `-nopar` flag is used on the command line.

C.3 Semantic Assertions

Semantic assertions provide information to the compiler that could be proved true about the program even though that proof is beyond the capabilities of the compiler. Asserting this information often yields more effective compilation.

In the following list, the term *variable-list* refers to a comma-separated list of variable names.

The compiler recognizes the following semantic assertions:

`#pragma mta assert can replace` *variable-list*

This directive asserts that it is safe to use scalar replacement of the aggregates (objects or `structs`) in *variable-list* and the aggregates pointed to by pointers in *variable-list*. This pragma is also a request for scalar replacement of those aggregates even if the code was not compilied with the `-scalar_replacement` option.

Items in *variable-list* must be aggregates or pointers to aggregates. Any pointers must either be marked with a `noalias` pragma or qualified with the `restrict` type qualifier. In addition, pointers must point only to a single aggregate during a given invocation of the routine in which the pragma appears. See *Scalar Replacement Section of Optimization Guide* for more information.

`#pragma mta assert loop can replace` *variable-list*

This directive asserts that it is safe to use scalar replacement of the aggregates (objects or `structs`) in *variable-list* and the aggregates pointed to by pointers in *variable-list* for the loop that immediately follows the pragma. This pragma is also a request for scalar replacement of those aggregates even if the code was not compilied with the `-scalar_replacement` option.

Items in *variable-list* must be aggregates or pointers to aggregates. Any pointers must either be marked with a `noalias` pragma or qualified with the `restrict` type qualifier. In addition, pointers must point only to a single aggregate within the loop. See *Scalar Replacement Section of Optimization Guide* for more information.

`#pragma mta assert no replace` *variable-list*

This directive tells the compiler not to use scalar replacement of the aggregates (objects or `structs`) in *variable-list* and any aggregates pointed to by pointers in *variable-list*. This is useful for fine-tuning files that are compilied with the `-scalar_replacement` option. See *Scalar Replacement Section of Optimization Guide* for more information.

`#pragma mta assert parallel`

This directive can appear before a loop construct and asserts that the separate iterations of the loop may execute concurrently without synchronization. It does not guarantee that the compiler parallelizes the loop, but it is a strong suggestion to the compiler. This directive affects the next loop only. The directive is ignored if the `-nopar` flag is used on the command line.

`#pragma mta assert local` *variable-list*

This directive can appear inside a loop or inside the body of a function, or at the top of the loop or function. For a loop, it asserts that at the beginning of each iteration, the compiler can treat the listed variables as undefined, and that their values are not referenced after the completion of that iteration. For a function, it asserts that the variables are undefined on entry to the function, and that their values are not referenced after exiting the function. The behavior of this directive is the same regardless of whether the loop or function to which it is attached executes in a parallel or serial context.

```
void assert_local_example(double B[], const int N)
{
  double A[2];
  for (int i = 0; i < N; i++) {
#pragma mta assert local A
    A[0] = i;
    A[1] = 2*i;
    B[i] = A[0]*A[1];
  }
}
```

In the previous example, the directive asserts that `A` is used as a scratch array in the loop. This directive must be inside the loop in order to affect the loop.

`#pragma mta assert no dependence` *variable-list*
`#pragma mta assert nodep` *variable-list*

This directive can appear before a loop construct and asserts that if a word of memory is accessed during execution of the loop through any load or store derived from a variable in *variable-list*, the word is accessed from exactly one iteration of the loop. You can also use the word `nodep` in place of `no dependence`. For example:

```
void nodep_example(const int INDEX[], double IA[100][100],
                   const int N)
{
  // You know that index[I] is never 1.
#pragma mta assert noalias *IA
#pragma mta assert no dependence *IA
  for (int i = 0; i < N; i++) {
    IA[i][1] = IA[i][INDEX[i]];
  }
}
```

`#pragma mta assert may reorder` *variable-list*
`#pragma mta may reorder` *variable-list*

This directive allows the compiler to reorder accesses of the variables in *variable-list* with respect to other volatile and global references in the code. This directive is used to remove unnecessary restrictions that may be placed on the order of execution. For example, in the following code, if `SYNCARRAY$` is a sync-qualified array, the order of accesses to the various elements of the array are serialized, and the loop is not parallelized:

```
void may_reorder_example(sync int SYNCARRAY$[10000])
{
  for (int i = 0; i < 10000; i++) {
    SYNCARRAY$[i] = 0;
  }
}
```

However, if we add a `#pragma mta may reorder SYNCARRAY$` directive before the loop, each reference to `SYNCARRAY$` may occur before or after any of the other references. Explicit serialization is not imposed, and the loop is parallelizable.

```
void may_reorder_example(sync int SYNCARRAY$[10000])
{
#pragma mta may reorder SYNCARRAY$
  for (int i = 0; i < 10000; i++) {
    SYNCARRAY$[i] = 0;
  }
}
```

`#pragma mta assert may not reorder` *variable-list*
`#pragma mta may not reorder` *variable-list*

This directive is used to deactivate the preceding `may reorder` directive. The following example tells the compiler that accesses to `SYNCARRAY$` can be reordered only in the loop shown.

```
void maynot_reorder_example(sync int SYNCARRAY$[10000])
{
  int i;
  for (i = 0; i < 10000; i++) {
#pragma mta may reorder SYNCARRAY$
    SYNCARRAY$[i] = 0;
#pragma mta may not reorder SYNCARRAY$
  }
}
```

`#pragma mta assert noalias` *variable-list*
`#pragma mta noalias` *variable-list*

This directive tells the compiler that the variables in *variable-list* are not used as aliases for any other variables. This information allows the compiler to perform a more accurate dependence analysis of loops involving these variables and to more aggressively parallelize the code. This directive must follow the declaration of the variables in *variable-list* and must lie within the scope in which these variables are defined. The directive may also take the form `#pragma noalias` *variable-list.*

`#pragma mta assert par_newdelete`

This directive is placed before the definition of a new array to indicate that when the elements of the array are constructed, the constructors should be invoked in parallel. To do this, use the following syntax for automatic or external definitions.

```
#pragma mta assert par_newdelete
aclass foo[100];
```

In this case, the destructors are not fired in parallel; there is no way to cause destructors to be fired in parallel for these kinds of definitions.

Alternatively, you can use the following syntax for dynamically allocated arrays.

```
#pragma mta assert par_newdelete
foo = new aclass[100];
```

This directive is placed before the deletion of a dynamically allocated array to indicate that when the elements of the array are destructed, the destructors should be invoked in parallel. To do this, use the following syntax:

```
#pragma mta assert par_newdelete
delete [] foo;
foo = 0;
```

C.4 Implementation Hints

The following directives provide implementation hints to the compiler about the expected behavior of the program. The intent is to provide guidance for effective optimization.

`#pragma mta expect count` *integer-expression*

This directive can appear before a loop construct. The *integer-expression* is a constant expression and serves as an estimate of the number of times the loop will iterate. The compiler optimizes the implementation of the loop based on this value. A constant integer-expression is one that can be evaluated completely by the front end of the compiler. It may not use the following:

- An expression that syntactically looks like a function call (such as `sizeof` or C++ style-type conversions)
- Floating-point literals
- GNU extensions

It may refer to members of enumerations.

`#pragma mta expect [true|false]`

This directive can appear before a logical `if` and specifies the expected value of the associated predicate. You can use this directive for branch prediction and choosing the best parallel implementation of a containing loop depending on sparse versus dense branching.

`#pragma mta expect case` *n*

This directive is similar to the `expect [true|false]` directive except that *n* is an integer. This directive must only appear before a switch statement. It tells the compiler that case arm *n* is expected.

The compiler tests for case *n* first, and all other cases after that. *n* must be an integer constant, in any radix. It may not be an integer expression, nor may it be a member of an enumeration.

`#pragma mta expect (`*predicate*`)`

This directive can appear before any executable statement and suggests that the compiler should optimize code near that point. This suggestion is based on the assumption that the *predicate* typically evaluates to true. This directive is deprecated and should not be used.

`#pragma mta expect parallel`

Deprecated form of `expect parallel context` directive that follows.

`#pragma mta expect parallel context`

This directive is inserted immediately before a function declaration. It tells the compiler that the following function is expected to be called in a highly parallel context. In this case, the compiler reduces the total number of instructions issued by the function rather than the serial execution time. By default, the compiler assumes that a function is called in a serial context unless the function is marked with the `expect parallel context` directive or the `-parcontext` flag was used on the compiler command line. This directive affects the next function only.

`#pragma mta expect serial context`

This directive is inserted immediately before a function declaration. It tells the compiler that the following function is expected to be called in a serial context. In this case, the compiler reduces the serial execution time for the function. By default, the compiler assumes that a function is in a serial context unless the `-parcontext` flag was used on the compiler command line or the function is marked with the `expect parallel context` directive in the code. The `expect serial context` directive overrides the `-parcontext` compiler flag for the function immediately following the directive. This directive affects the next function only.

Condition Codes [D]

You can test the condition codes generated by an expression by using the `MTA_TEST_CC` intrinsic. The eight possible condition code values and their default meanings are shown in the following table. The **Examples** column show the operations that meet the criteria for the condition code, where `0`, `p`, and `n` stand for zero, a positive integer, and a negative integer, respectively. For more information, see Testing Expressions Using Condition Codes on page 34 and Chapter 4 of the *Cray XMT Principles of Operation*.

Table 2. Condition Codes

Name	Meaning	Examples
`COND_ZERO_NC`	Zero, no carry	`0 = 0+0`
`COND_NEG_NC`	Negative, no carry	`n = p+n, n = p-p`
`COND_POS_NC`	Positive, no carry	`p = p+p, p = p-n`
`COND_OVFNAN_NC`	Overflow/NaN, no carry	`n = p+p, n = p-p`
`COND_ZERO_C`	Zero, carry	`0 = n+p, 0 = n-n`
`COND_NEG_C`	Negative, carry	`n = n+n, n = n-p`
`COND_POS_C`	Positive, carry	`p = n+p, p = n-n`
`COND_OVFNAN_C`	Overflow/NaN, carry	`p = n+n, p = n-p`

Most of the important condition masks have one or more names. The named condition masks are shown in Table 3. For more information, see *Cray XMT Programming Model*.

Table 3. Condition Masks

Name	Description
Condition Mask: Manifest	
`IF_ALWAYS`	Always
`IF_NEVER`	Never
Condition Mask: Equality	
`IF_EQ`	y = z (integer, unsigned, float)

Name	Description
`IF_ZE`	x = 0 (integer, unsigned, float)
`IF_F`	x = 0 (logical)
`IF_NE`	y != z (integer, unsigned, float)
`IF_NZ`	x != 0 (integer, unsigned, float)
`IF_T`	x != 0 (logical)
Condition Mask: Integer Comparison	
`IF_ILT`	y < z (integer)
`IF_IGE`	y >= z (integer)
`IF_IGT`	y > z (integer)
`IF_ILE`	y <= z (integer)
`IF_IMI`	x < 0 (integer)
`IF_IPZ`	x >= 0 (integer)
`IF_IPL`	x > 0 (integer)
`IF_IMZ`	x <= 0 (integer)
Condition Mask: Unsigned Comparison	
`IF_ULT`	y < z (unsigned)
`IF_UGE`	y >= z (unsigned)
`IF_UGT`	y > z (unsigned)
`IF_ULE`	y <= z (unsigned)
Condition Mask: Float Comparison	
`IF_FLT`	y < z (float)
`IF_FGE`	y >= z (float)
`IF_FGT`	y > z (float)
`IF_FLE`	y <= z (float)
Condition Mask: Other Tests	
`IF_IOV`	x overflowed (integer)
`IF_FUN`	y and z are unordered (float)
`IF_CY`	Carry
`IF_NC`	No carry
Condition Mask: Specific Conditions	
`IF_0`	Zero, no carry
`IF_1`	Negative, no carry
`IF_2`	Positive, no carry

Name	Description
`IF_3`	Overflow/NaN, no carry
`IF_4`	Zero, carry
`IF_5`	Negative, carry
`IF_6`	Positive, carry
`IF_7`	Overflow/NaN, carry
`IF_N0`	Not Zero, no carry
`IF_N1`	Not Negative, no carry
`IF_N2`	Not Positive, no carry
`IF_N3`	Not Overflow/NaN, no carry
`IF_N4`	Not Zero, carry
`IF_N5`	Not Negative, carry
`IF_N6`	Not Positive, carry
`IF_N7`	Not Overflow/NaN, carry

Data Types [E]

This chapter provides information about the C and C++ language data types that you can use with Cray XMT compilers.

The floating-point types are `float`, `double`, and `long double`. Their sizes are 4, 8, and 16 bytes, respectively.

The integer types `short` and `unsigned short` are each 4 bytes long. The data types `int`, `long`, `long long`, and their unsigned equivalents are each 8 bytes long. The compiler flag `-short16` converts all `short` and `unsigned short` integers to 2 bytes. The compiler flag `-i4` converts all `short` and `unsigned short` integers to 2 bytes and all `int` and `unsigned int` to 4 bytes.

The two character types `char` and `unsigned char` are each 1 byte long. Additionally, the C++ compiler supports a 1-byte boolean type, `bool`, and the boolean constants `true` and `false`. The compiler flag `-no_bool` turns off recognition of these three keywords.

The Cray XMT C and C++ compilers also support the ten nonstandard integer types in the following list. The `-short16` and `-i4` compiler flags do not affect the size of these types, so it is preferable that you use these in exported include files.

`__short16`	A 2-byte (16-bit) value.
`unsigned __short16`	A 2-byte (16-bit) value.
`__short32`	A 4-byte (32-bit) value.
`unsigned __short32`	A 4-byte (32-bit) value.
`__int16`	A 2-byte (16-bit) value.
`unsigned __int16`	A 2-byte (16-bit) value.
`__int32`	A 4-byte (32-bit) value.
`unsigned __int32`	A 4-byte (32-bit) value.
`__int 64`	An 8-byte (64-bit) value.
`unsigned __int64`	An 8-byte (64-bit) value.

Keywords [F]

The C and C++ languages reserve certain words for use as keywords. You cannot use these words for any other purpose. For example, you cannot use them as identifiers such as variable names. Some of these reserved words are required by the standards for the C and C++ languages; others support programming on the Cray XMT.

Table 4. C/C++ Keywords Recognized by the Cray XMT Compiler

`auto`	`default`	`float`	`return`	`switch`	`while`
`break`	`do`	`for`	`short`	`typedef`	
`case`	`double`	`goto`	`signed`	`union`	
`char`	`else`	`int`	`sizeof`	`unsigned`	
`const`	`enum`	`long`	`static`	`void`	
`continue`	`extern`	`register`	`struct`	`volatile`	

When you use the `-traditional` compiler switch on the C command line, it disables the keywords `const`, `signed` and `volatile`.

Table 5. Standard C++ Keywords Recognized by the Cray XMT Compiler

`and`	`const_cast`	`namespace`	`protected`	`try`
`and_eq`	`delete`	`new`	`public`	`typeid`
`bitand`	`dynamic_cast`	`not`	`reinterpret_cast`	`typename`
`bitor`	`explicit`	`not_eq`	`static_cast`	`using`
`bool`	`false`	`operator`	`template`	`virtual`
`catch`	`friend`	`or`	`this`	`wchar_t`
`class`	`inline`	`or_eq`	`throw`	`xor`
`compl`	`mutable`	`private`	`true`	`xor_eq`

The `-no_bool` compiler switch disables the `bool`, `false` and `true` keywords. The `-no_wchar` compiler switch disables the `wchar_t` keyword. The `-cfront` compiler switch disables the `bool`, `explicit`, `false`, `true` and `typename` keywords. The `-no_alternative_tokens` compiler switch disables the alternate operator keywords `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor`, and `xor_eq`.

In addition to the keywords required by the language standards, the Cray XMT platform uses several additional reserved words. Most of the additional keywords reserved by Cray for use on the Cray XMT have two forms: one beginning with an alphabetic character and one beginning with a double underscore (__). Use the `-no_mta_ext` compiler switch to disable Cray XMT keywords beginning with a letter of the alphabet. However, Cray XMT keywords beginning with a double underscore are not affected by the `-no_mta_ext` compiler switch. In addition, the keywords `__int16`, `__int32`, `__int64`, `__short16` and `__short32` are not affected by the `-i4` and `-short16` compiler switches. For this reason, you sometimes see the double underscore format in header files to preserve the meaning of the keywords.

When using the type qualifier keywords to qualify a pointer type, follow the same rules as for the standard C and C++ type qualifiers. For example, in the following declaration:

```
int * sync f;
```

`f` is a sync variable of type pointer to `int`, but in the following declaration:

```
sync int * f;
```

`f` is a pointer to a sync variable of type `int`.

The following reserved words have been added by Cray to both the C and C++ languages for use on the Cray XMT platform.

`future` `__future`	Both a type qualifier and a statement. Future variables are initially set to a full state. A future variable is set to an empty state when the `future` statement executes and set to a full state when the return statement of the future executes. A read or write operation runs successfully when a future variable is set to a full state and leaves the variable set to a full state. For an example that shows the use of the `future` variable and `future` statement, see *Cray XMT Programming Model*.
`__int16`	Integer type. A 2-byte value; may be signed or unsigned. See Appendix E, Data Types on page 137.
`__int32`	Integer type. A 4-byte value; may be signed or unsigned. See Appendix E, Data Types on page 137.
`__int64`	Integer type. An 8-byte value; may be signed or unsigned. See Appendix E, Data Types on page 137.
`restrict`	Type qualifier. Similar in function to the `noalias` compiler directive. See Semantic Assertions on page 125. When you declare a pointer with the `restrict` type, it indicates that the code does not use aliases for that pointer and the compiler can perform additional optimizations, such as the implicit parallelization of loops.
`__short16`	Integer type. A 2-byte value; may be signed or unsigned. See Appendix E, Data Types on page 137.
`__short32`	Integer type. A 4-byte value; may be signed or unsigned. See Appendix E, Data Types on page 137.
`sync` `__sync`	Type qualifier. The system atomically reads sync variables when in a full state and then sets them to an empty state. The system atomically writes sync variables when in an empty state and then sets them to a full state. The system automatically sets uninitialized sync variables to an empty state unless you use the `-no_purge` compiler switch; the system sets initialized sync variables to a full state.
`task` `__task`	Reserved for future use.

The following reserved words have been added by Cray to the C language for use on the Cray XMT platform. Keywords beginning with an underscore have also been added by Cray to the C++ language. The keywords `new`, `delete`, and `protected` are required by the C++ standard and did not need to be added to that language.

`new`
`__new` Unary operator; has the same format as the `new` operator in the C++ language. Allocates space for an object of the specified type, initializes the full-empty bit of any sync or future variables that the new object contains, and returns the address of the new object. The system initializes the sync variable to an empty state and the future variable to a full state. The actual contents of these variables, as for any variables contained by the new object, is undefined.

`delete`
`__delete` Unary operator; has the same format as the `delete` operator in the C++ language. Deallocates space that was previously allocated using the `new` operator.

`protected`
`__protected`

Reserved for future use.

MTA_PARAMS [G]

The environment variable `MTA_PARAMS` is used by the Cray XMT user runtime. The following list contains the values that you can set for `MTA_PARAMS`.

`debug_data_prot`

Waits for the debugger to attach rather than exiting when a data protection or poison error occurs. This parameter is useful while troubleshooting a specific problem. However, Cray does not recommend that you use this parameter during normal operations because any error that occurs causes the runtime to wait for the debugger to attach. This results in the runtime holding on to resources previously used by the program.

`do_backtrace`

Dumps registers of all active streams when a trap occurs. This parameter may be useful during troubleshooting, although it generates a lot of information. If the runtime system has become corrupted, the registers may fail to dump.

`echo` Prints a list of parameters to the screen. This parameter toggles on and off.

`exit_on_trace_fail`

Sets the default behavior to kill the program execution when tracing fails to initialize.

`ft_traps` *options*

Enables various floating-point traps depending upon which options you set. See the section of Programming Considerations for Floating-point Operations in *Cray XMT Programming Model*. You can select from the following list of options:

`i`	Invalid. Traps invalid floating-point numbers.
`z`	Zero-divide. Traps operations that are attempting to divide a floating-point number by 0.0. This type of operation would create a NaN.
`o`	Overflow. Traps overflows that occur.

`u`	Underflow. Traps underflows that occur. Underflows produce a rounded result smaller in magnitude than `0x0010000000000000`, or about `2.225e-308.`
x	Inexact. Traps subnormal numbers.

`max_readypool_retries` *n*

Sets the maximum number *n* of retries that an idle thread can take when checking random ready pools for new work.

`mmap_buffer_size` *n*

Sets the variable size of the persistent mmap buffers, where *n* is the size in words. The maximum value, which is also the default, is 16,777,216 words (16 GB). The size of the persistent buffers determines how much tracing data can be gathered before requiring a dump of the gathered data to the `trace.out` file.

`must_dump_size` *n*

Specifies the minimum number of words that must be present in a trace buffer before allowing the trace buffer to dump to the mmap buffer. The default value is 512 words. If an application terminates prematurely and the trace.out file is missing information, reduce the size of this buffer to force more frequent dumping.

`num_procs` *n*

Sets the maximum number of processors to use. This parameter is the same as using the command `mtarun -m` *n*.

`num_readypools` *n*

Sets the maximum number *n* of ready pools available for the entire task. Ready pools are used to schedule futures.

`no_prereserve`

Prevents the runtime from reserving 3 streams to use for attaching the debugger.

`pc_hash` *n*, *m*, *l*

Specifies the hash size *n*, age threshold *m*, and dump threshold *l* of an event. The has size determines the number of event types that can be hashed at one time. The age threshold determines the age at which an event is considered stale, in which case it will be discarded rather than reported. The age threshold also determines the frequency with which events are captured in event records. The dump threshold is the minimum number of events that must have been hashed to a particular location before that location is captured as an event record when the next age threshold sample is taken.

`stream_limit` *n*

Sets the maximum number of streams to use on each processor. The system imposed limit is `100` streams. However, while debugging a program, it may be easier to perform debugging if this parameter is set to a smaller number. The minimum value is `5`.

LUC API Reference [H]

The XMT-PE contains two user-level libraries for LUC, `libluc.a`, that use a C++ interface. One version of `libluc.a` is built for Linux applications and one is built for MTK applications. Both versions present the same interface to LUC applications.

For LUC applications, you use the `<luc/luc_exported.h>` header file.

H.1 `LucEndpoint` Class

The `LucEndpoint` class defines a `LucEndpoint` object.

The `LucEndpoint` class provides the interface methods that the application uses to call functions on a remote server.

```
class LucEndpoint {
public:

     /********************************************
     * Shared functions
     ********************************************/

    // initialize the service and start the client or server thread
    virtual luc_error_t startService(uint_t threadCount=1,
         uint_t myRequestedPid=PTL_PID_ANY);

    // stop the client or server thread and shutdown the service
    virtual luc_error_t stopService(void);

    // returns the endpoint ID
    virtual luc_endpoint_id_t  getMyEndpointId(void);

 // set per-endpoint configuration values
    virtual luc_error_t setConfigValue(luc_config_key_t key,  uint64_t value);

    // read per-endpoint configuration values
    virtual luc_error_t getConfigValue(luc_config_key_t key,  uint64_t *value);

    /********************************************
     * Client functions
     ********************************************/

    // client asynchronous RPC
    virtual luc_error_t remoteCall(luc_endpoint_id_t serverEndpoint,
       luc_service_type_t serviceType,
       int serviceFunctionIndex,
       void *userData,
       size_t userDataLen,
       void * userHandle,
       LUC_Completion_Handler userCompletionHandler);

    // client synchronous RPC
    virtual luc_error_t remoteCallSync(luc_endpoint_id_t serverEndpoint,
           luc_service_type_t serviceType,
           int serviceFunctionIndex,
           void *inputData,
           size_t inputDataLen,
           void *outputData,
           size_t *outputDataLen);

    /********************************************
     * Server functions
     ********************************************/

    virtual luc_error_t registerRemoteCall(luc_service_type_t  serviceType,
        int serviceFunctionIndex,
        LUC_RPC_Function_InOut theFunction);
};
```

H.2 `luc_allocate_endpoint` Function

Use `luc_allocate_endpoint` to construct `LucEndpoint` objects. The default value for `LucServiceType` is `LUC_CLIENT_SERVER`. See LUC Type Definitions on page 159.

```
LucEndpoint *luc_allocate_endpoint(LucServiceType_t etype);
```

H.3 LUC Methods

The `LucEndpoint` class uses the following methods:

- `startService`
- `stopService`
- `getMyEndpointID`
- `remoteCall`
- `remoteCallSync`
- `registerRemoteCall`
- `setConfigValue`
- `getConfigValue`

H.3.1 `startService` Method

Initializes the `LucEndpoint` object.

Syntax

```
luc_error_t startService(uint_t threadCount=1,
        ptl_pid_t requestedPid = PTL_PID_ANY);
```

This method puts the object into a state where it can initiate and respond to RPC requests. It initializes internal network components and creates the required number of threads. The MTK version of the library allocates I/O buffers for the endpoint as part of this initialization.

For client only objects, the `threadCount` parameter is ignored.

The MTK version of the library ignores both parameters.

Parameters

threadCount

Specifies the number of server threads that are assigned to an object.

Note: The MTK LUC library ignores the `threadCount` parameter.

requestedPid

Specifies a Portals process ID to use when setting up the endpoint. By default, the LUC library chooses a Portals process ID to use.

Note: MTK ignores the *requestedPid* parameter.

Return Codes

`LUC_ERR_OK` The service was stopped.

`LUC_ERR_ALREADY_STARTED`

User attempted to `startService` on a previously started `LucEndpoint` object

H.3.2 `stopService` Method

Stops the `LucEndpoint` object.

Syntax

```
luc_error_t stopService(void);
```

Undoes the work of `startService`. `stopService` waits for running threads to finish, then terminates them. It frees up any memory and network resources associated with the endpoint that were allocated in a previous `startService` call.

Return Codes

`LUC_ERR_OK` The service was stopped.

`LUC_ERR_NOT_STARTED`

The service has not yet been started. To start the service, use the `startService` method.

H.3.3 `getMyEndpointID` Method

Returns the ID of the `LucEndpoint` object.

Syntax

```
luc_endpoint_id_t GetMyEndpointId(void);
```

Gets the ID of the endpoint. This method is valid only after `startService` has returned.

Return Codes

This method returns the endpoint's identifier on successful completion.

`LUC_ENDPOINT_INVALID`

The endpoint is invalid because the service has not yet been started. To start the service, use the `startService` method.

H.3.4 `remoteCall` Method

Makes an asynchronous remote procedure call.

Syntax

```
luc_error_t remoteCall(luc_endpoint_id_t serverEndpoint,
       luc_service_type_t serviceType,
       int serviceFunctionIndex,
       void *userData,
       size_t userDataLen,
       void * userHandle,
       LUC_Completion_Handler userCompletionHandler);
```

The asynchronous RPC mechanism is useful in cases where the caller does not need assurance that the remote call actually happened. Locally detected errors may be returned but remote errors are not returned directly. Remote-side success or failure are returned if the caller provides a completion handler. The completion handler is guaranteed to execute once and only once—when the remote call is known to have executed or has been abandoned.

This method call is valid only on started objects. Multiple concurrent callers of this method and the synchronous version are supported.

Parameters

serverEndpoint

Specifies the endpoint identifier for the desired server of this RPC.

serviceType
serviceFunctionIndex

These parameters specify the particular remote function to invoke on a server. The server uses the same values in its `registerRemoteCall` method.

userData
userDataLen

Specifies an optional pointer to input data and the length of the data.

userHandle Contains the value passed to the specified `userCompletionHandler` when it is invoked.

userCompletionHandler

Contains a function pointer for a function to be called when the remote procedure call completes.

Return Codes

`LUC_ERR_OK` The remote procedure call was launched.

`LUC_ERR_IO_ERROR`

An underlying transport error occurred. The remote procedure call may or may not have launched.

`LUC_ERR_TOO_LARGE`

The remote procedure is trying to return more data than the client can accept. This return code is generated when servers return data to an asynchronous caller.

`LUC_ERR_NOT_STARTED`

The service has not yet been started. To start the service, use the `startService` method.

`LUC_ERR_BAD_ADDRESS`

Indicates an attempt to use a NULL input or output buffer while specifying a non-zero size for the corresponding buffer. This error is returned by the `remoteCall` and `remoteCallSync` methods.

H.3.5 `remoteCallSync` Method

Makes a synchronous remote procedure call.

Syntax

```
luc_error_t remoteCallSync(luc_endpoint_id_t serverEndpoint,
           luc_service_type_t serviceType,
           int serviceFunctionIndex,
           void *inputData,
           size_t inputDataLen,
           void *outputData,
           size_t *outputDataLen);
```

The synchronous procedure call is used in synchronous programming models or in cases where the caller expects the remote function to return data.

This method is valid only on started objects. Multiple concurrent callers of this method and the asynchronous version are supported.

Parameters

serverEndpoint

Specifies the endpoint identifier for the desired server of this RPC.

serviceType
serviceFunctionIndex

Specifies the particular remote function to invoke on a server and its service type. The server uses the same values in its `registerRemoteCall` method.

inputData
inputDataLen

Specify an optional pointer to input data and the length of the data.

outputData (input parameter)

Specifies an optional buffer for return data from the RPC.

outputDataLen (*input/output parameter*)

As an input parameter, specifies the maximum amount of data that the application will accept from the RPC (the allocated size of `outputData`). When `remoteCallSync` returns, this value will be changed to the actual amount of returned data.

Return Codes

`LUC_ERR_OK` The remote procedure call was completed. Data may have been returned.

`LUC_ERR_NOT_STARTED`

The service has not yet been started. This error is returned by the `stopService`, To start the service, use the `startService` method.

`LUC_ERR_BAD_ADDRESS`

Indicates an attempt to use a NULL input or output buffer while specifying a non-zero size for the corresponding buffer.

asynchronous return codes

Any of the asynchronous calls return codes and completion handler codes may be returned to indicate failures. Refer to the return codes section in `remoteCall` Method on page 151.

application defined return codes

Any return codes defined by the application.

H.3.6 `registerRemoteCall` Method

Registers a remote application function with the server.

Syntax

```
luc_error_t registerRemoteCall(luc_service_type_t  serviceType,
        int serviceFunctionIndex,
        LUC_RPC_Function_InOut theFunction);
```

This method registers the specified function to be executed whenever an incoming request matches the specified service type and function index to be associated with the application function.

This method operates independent of `startService` and `stopService`. It may be called for an object in any state. (Remote procedure calls are unregistered **only** when the object is destroyed).

Parameters

serviceType

Specifies the service type of the service being provided.

serviceFunctionIndex

Specifies the specific function (by index) being provided by `theFunction`.

theFunction

Specifies the application defined function to be called by LUC when RPC requests arrive at the endpoint with a matching `serviceType` and `serviceFunctionIndex`.

Return Codes

`LUC_ERR_OK` The function was registered successfully.

`LUC_ERR_BAD_PARAMETER`

The specified service type or function index is out-of-range.

`LUC_ERR_ALREADY_REGISTERED`

The specified service type or function index is already occupied.

`LUC_ERR_OTHER`

The prototype can handle only a fixed number of function registrations for each server object.

H.3.7 `setConfigValue` Method

Sets configuration values for LUC.

Syntax

```
luc_error_t setConfigValue(
    luc_config_key_t key,
    uint64_t value);
```

Parameters

key Identifies the configuration option to set. The following options can be set:

`LUC_CONFIG_LOG_LEVEL`

This configuration key alters the amount of LUC internal debugging information that is printed to standard error.

Values to use for this option:

`LUC_DBG_NONE` — The library logs assertions that are fatal to the application.

`LUC_DBG_LOW` — The library logs fatal assertions and errors.

`LUC_DBG_MEDIUM` — The library logs errors and warnings.

`LUC_DBG_HIGH` — The library logs errors, warnings, and verbose information about RPCs and the endpoints.

`LUC_CONFIG_SERVER_RPC_COUNT`

This configuration key sets the number of RPCs that a server endpoint should be able to handle at once.

Values to use for this option: 1 to 13106, inclusive.

`LUC_CONFIG_CLIENT_RPC_TIMEOUT`

The number of seconds that a server endpoint will wait for an expected message from a client before failing the RPC.

Values to use for this option: Any number greater than zero.

`LUC_CONFIG_SERVER_RPC_TIMEOUT`

The number of seconds that a server endpoint will wait for an expected message from a client before failing the RPC.

Values to use for this option: Any number greater than zero.

`LUC_CONFIG_MAX_NEARMEM_SIZE`

This configuration key adjusts the amount of nearby memory allocated for the endpoint's small I/O buffer region. This buffer region may not be disabled.

This key is not valid for Linux endpoints.

Values to use for this option: powers-of-two from 1 MB to 256 MBs, inclusive.

`LUC_CONFIG_SWAP_CLIENT_INBOUND`
`LUC_CONFIG_SWAP_CLIENT_OUTBOUND`
`LUC_CONFIG_SWAP_SERVER_INBOUND`
`LUC_CONFIG_SWAP_SERVER_OUTBOUND`

This configuration key uses boolean flags to enable byte swapping on messages sent to a LUC client, from a LUC client, to a LUC server, and from a LUC server, respectively.

These are not valid for Linux endpoints.

Values to use for this option: 0 and 1.

`LUC_CONFIG_CLIENT_RPC_COUNT`

This configuration key sets the maximum number of concurrent client RPCs on a single endpoint.

Values to use for this option: 1 to 13106, inclusive.

`LUC_CONFIG_MAX_LOCAL_ENDPOINTS`

This configuration key sets the maximum number of started LUC endpoints that may exist in a single Linux process.

This key is not valid for MTK endpoints.

Values to use for this option: 1 to 512, inclusive.

`LUC_CONFIG_MAX_LARGE_NEARMEM_SIZE`

This configuration key adjusts the amount of nearby memory allocated for the endpoint's large I/O buffer region.

This key is not valid for Linux endpoints.

Values to use for this option: powers of two from 1 MB to 2 GB, inclusive. A special value of zero (0) may be used to disable this memory region and force all I/O memory requests to be handled by the small memory buffer.

`LUC_CONFIG_MAX_LARGE_MEM_REQUEST`

This configuration key sets the largest internal memory request that will be handled by the endpoint's large I/O buffer region.

This key is not valid for Linux endpoints.

Values to use for this option: powers of two from 1 MB to 256 MBs, inclusive.

`LUC_CONFIG_SMALL_NEARMEM_SIZE`

This configuration key adjusts the amount of nearby memory allocated for the endpoint's small I/O buffer region.

This key is not valid for Linux endpoints.

Values to use for this option: powers of two from 1 MB to 256 MBs, inclusive. This buffer region may not be disabled.

`LUC_CONFIG_MAX_SMALL_MEM_REQUEST`

This configuration key sets the largest internal memory request that will be handled by the endpoint's small I/O buffer region.

This key is not valid for Linux endpoints.

Values to use for this option: powers of two from 64 KBs to 256 MBs, inclusive.

value Identifies the value to set for the corresponding configuration key.

Return Codes

`LUC_ERR_OK` The operation was successful.

`LUC_ERR_INVALID_KEY`

The *key* parameter is not one of the predefined LUC configuration keys (`LUC_CONFIG_*`).

`LUC_ERR_INVALID_STATE`

The `setConfigValue` method cannot change the key value because of the endpoint's current state. The endpoint must be stopped to set the nearby memory region configuration values.

H.3.8 `getConfigValue` Method

Returns the value for a specified configuration option for LUC.

Syntax

```
luc_error_t getConfigValue(
    luc_config_key_t key,
    uint64_t *value);
```

Parameters

key	Identifies the configuration option to get. For a list of configuration options, see `setConfigValue` Method on page 155.
value	Returns a pointer to the value for the corresponding configuration key.

Return Codes

`LUC_ERR_OK`	The operation was successful.
`LUC_ERR_INVALID_KEY`	The *key* parameter is not one of the predefined LUC configuration keys (`LUC_CONFIG_*`).

H.4 LUC Type Definitions

`LucServiceType` defines the type of the `LucEndpoint` object.

```
typedef enum {
    LUC_SERVER_ONLY  = 1,
    LUC_CLIENT_ONLY,
    LUC_CLIENT_SERVER
} LucServiceType_t;
```

Endpoints may be constructed to behave as a client and a server or they can be specialized to be one or the other. The `LucServiceType` `typedef` describes what type of `LucEndpoint` object is being created.

LUC remote procedure calls can be grouped by their intended service type. The following service types are predefined. The programmer can specify other application specific values or use the predefined values.

```
typedef u_int32_t luc_service_type_t;

#define   LUC_ST_QueryManager   0
#define   LUC_ST_QueryEngine    1
#define   LUC_ST_Coordinator    2
#define   LUC_ST_Restore        3
#define   LUC_ST_Snapshot       4
#define   LUC_ST_UpdateManager  5
#define   LUC_ST_UpdateEngine   6
#define   LUC_ST_OutputLog      7
#define   LUC_ST_Any            8
#define   LUC_ST_ErrorLog       9
```

Error return codes are described with the methods that return them. The programmer can specify other application specific error return codes or use the predefined values.

```
typedef int32_t luc_error_t;
```

H.5 LUC Callback Functions

The `LucEndpoint` class uses the following callback functions:

- `LUC_RPC_Function_InOut`
- `LUC_Mem_Avail_Completion`
- `LUC_Completion_Handler`

H.5.1 `LUC_RPC_Function_InOut`

The LUC runtime calls `LUC_RPC_Function_InOut` callback when a remote client makes a request.

The application must call the `registerRemoteCall` method to register `LUC_RPC_Function_InOut` callback functions.

The application should return `LUC_ERR_OK` when successful. The application should not return or redefine any other predefined return codes.

Syntax

```
typedef luc_error_t (*LUC_RPC_Function_InOut)(void *inData,
                      uint64_t inDataLen,
                      void **  outData,
                      uint64_t *outDataLen,
                      void **  completionArg,
                      LUC_Mem_Avail_Completion *completionFctn,
                      luc_endpoint_id_t   callerEndpoint);
```

Parameters

inDataLen (input parameter)

Specifies a pointer to a buffer containing input data to the remote function. `NULL` if there is no input data.

inDataLen (input parameter)

Specifies the length of the `inData` buffer.

outData (output parameter)

Specifies a pointer to the output data returned by the application. `NULL` if there is no output data.

outDataLen (output parameter)

Specifies the length of the data returned by the application if there is returning data.

completionArg (output parameter)

Specifies the value to pass to `completionFctn`.

completionFctn (output parameter)

Specifies a pointer to a `LUC_Mem_Avail_Completion` callback function called when the buffer is available. Used when `LUC_RPC_Function_InOut` returns data to the LUC runtime and needs to be notified that the buffer is available for use.

callerEndpoint (input parameter)

Specifies the input endpoint identifier of the client's `LucEndpoint` object passed to the remote function.

H.5.2 LUC_Mem_Avail_Completion

The `LUC_Mem_Avail_Completion` callback function notifies `LUC_RPC_Function_InOut` that its buffer is available for use.

Syntax

```
typedef void (*LUC_Mem_Avail_Completion)(void * userHandle);
```

Parameters

userHandle	LUC passes in the `completionArg` value returned by the initiating `LUC_RPC_Function_InOut` function.

H.5.3 `LUC_Completion_Handler`

The `LUC_Completion_Handler` callback function is used by a client for asynchronous remote procedure calls.

`LUC_Mem_Avail_Completion`

Syntax

```
typedef void (*LUC_Completion_Handler)
                     (luc_endpoint_id_t originalDestAddr,
                      luc_service_type_t originalServiceType,
                      int originalFunctionIndex,
                      void *  userHandle,
                      luc_error_t  remoteError);
```

The LUC runtime will call the function specified in the `remoteCall` method that follows this signature when the remote call has completed.

Parameters

originalDestAddr
originalServiceType
originalFunctionIndex

Specifies the destination address, service type, and function index. LUC passes in the values used by the `remoteCall` method that initiated this RPC being completed.

userHandle LUC passes in the value specified by the `remoteCall` method that initiated this RPC being completed.

remoteError

The ultimate error code for the RPC from either the LUC library or the server application's registered function. All of the values returned by `remoteCallSync` (including application defined return codes) may be specified here.

H.6 LUC Return Codes

The meaning of some predefined return codes are dependent on the method that returns the code. Applications may define application specific codes.

`LUC_ENDPOINT_INVALID`

Indicates that the object has not been started and does not have a valid endpoint identifier.

`LUC_ERR_OK`

- The function was registered successfully.
- This object is ready to accept remote requests.
- The remote procedure call was launched.
- The remote procedure call was completed.
- The endpoint has been stopped successfully.
- The function was prepared for transmission. The application's completion handler is guaranteed to fire with a **real** status at some later point.

`LUC_ERR_MAX`

Special value set to be the highest numerical error code generated by the library. Applications may specify their own error codes to be greater than this value.

`LUC_ERR_BAD_ADDRESS`

Indicates an attempt to use a NULL input or output buffer while specifying a non-zero size for the corresponding buffer. This error is returned by the `remoteCall` and `remoteCallSync` methods.

`LUC_ERR_NOT_REGISTERED`

The caller tried to make an RPC call to an unregistered service type/function index pair that was not registered with `registerRemoteCall`.

`LUC_ERR_OTHER`

- The library can handle only a fixed number of function registrations for each server object. The library supports the registration of 64 functions for each endpoint.
- Failed to create the desired threads.

`LUC_ERR_ALREADY_REGISTERED`

The specified service type or function index is already occupied.

`LUC_ERR_BAD_PARAMETER`

- The specified service type or function index is out of range.
- The specified configuration value is out of range.

`LUC_ERR_RESOURCE_FAILURE`

A transient resource allocation failure has occurred. The caller should retry the operation at a later time.

`LUC_ERR_TOO_LARGE`

The remote procedure is trying to return more data than the client is able to accept. This return code will be generated whenever servers try to return data to an asynchronous caller.

`LUC_ERR_LIBRARY`

The (Linux) LUC Library received an unexpected error from the Portals Library.

`LUC_ERR_ALREADY_STARTED`

User attempted to `startService` on a previously started `LucEndpoint` object

`LUC_ERR_TIMEOUT`

Client failed to get a response from the server in a timely manner. The server is busy or a message was lost in transit.

`LUC_ERR_NOT_IMPLEMENTED`

Method not implemented. Returned by `remoteCall` and `remoteCallSync` for objects that were created as `LUC_SERVER_ONLY`. Returned by `registerRemoteCall` for objects that were created as `LUC_CLIENT_ONLY`.

`LUC_ERR_FIO`

The (MTK) LUC Library received an unexpected error from the Fast I/O System Call Library.

`LUC_ERR_INVALID_ENDPOINT`

The endpoint parameter to the method was invalid.

`LUC_ERR_ALREADY_STOPPED`

User attempted to `stopService` on a previously stopped, or never started, `LucEndpoint` object.

`LUC_ERR_IO_ERROR`

An underlying transport error occurred. The remote procedure call may or may not have fired.

`LUC_ERR_NOT_STARTED`

The service has not yet been started. This error is returned by the `stopService`, `remoteCall`, and `remoteCallSync` methods. To start the service, use the `startService` method.

`LUC_ERR_CANCELLED`

Endpoint was stopped while this RPC was in progress.

`LUC_ERR_INVALID_KEY`

The *key* parameter for the `setConfigValue` or `getConfigValue` methods is not one of the predefined LUC configuration keys (`LUC_CONFIG_*`). LUC configuration keys are defined in `getConfigValue` Method on page 158.

`LUC_ERR_INVALID_STATE`

The `setConfigValue` method cannot change the key value because of the endpoint's current state. The endpoint must be stopped to set the nearby memory region configuration values.

Glossary

barrier

In code, a barrier is used after a phase. The barrier delays the streams that were executing parallel operations in the phase until all the streams from the phase reach the barrier. Once all the streams reach the barrier, the streams begin work on the next phase.

block scheduling

A method of loop scheduling used by the compiler where contiguous blocks of loop iterations are divided equally and assigned to available streams. For example, if there are 100 loop iterations and 10 streams, the compiler assigns 10 contiguous iterations to each stream. The advantages to this method are that data in registers can be reused across adjacent iterations, and that there is no overhead due to accessing a shared iteration counter

dependence analysis

A technique used by the compiler to determine if any iteration of a loop depends on any other iteration (this is known as a loop-carried dependency).

dynamic scheduling

In a dynamic schedule, the compiler does not bind iterations to streams at loop startup. Instead, streams compete for each iteration using a shared counter.

fork

Occurs when processors allocate additional streams to a thread at the point where it is creating new threads for a parallel loop operation.

full-empty state

Indicates whether a variable contains a value (full) or not (empty). Generic read and write operations use this state to determine whether they can perform an operation on the variable. For example, a writeef operation can only write a value to a variable if the state is empty. After the write operation, it sets the state to full.

future

Implements user-specified or explicit parallelism by creating a continuation that points to a sequence of statements that may be executed by another idle thread. Futures also optionally contain a return value. Execution of code that uses the return value is delayed until the future completes. The thread that spawns the future uses parameters to pass data to the thread that executes the future. In a program, the term future is used as a type qualifier for a synchronization variable used to return the value of a future or as a keyword for a future statement.

induction variable

A variable that is increased or decreased by a fixed amount on each iteration of a loop.

inductive loop

A loop that contains no loop-carried dependencies and has the following characteristics: a single entrance at the top of the loop; controlled by an induction variable; and has a single exit that is controlled by comparing the induction variable against an invariant.

interleaved scheduling

A method of executing loop iterations used by the compiler where contiguous iterations are assigned to distinct streams. For example, for a loop with 100 iterations and 10 streams, one stream performs iterations 1, 11, 21,... while another stream performs iterations 2, 12, 22, ..., and so on. This method is typically used for triangular loops because it reduces imbalances. One disadvantage to using this method is that there is loss of data reuse between loop iterations because adjacent iterations are not executed by the same stream.

join

Occurs when threads that are forked for a parallel operation finish the operation. As threads finish and drop the streams they are running on, the streams join back together until there is a single stream running the thread.

linear recurrence

A special type of recurrence that can be parallelized.

loop-carried dependences

The value from one iteration of a loop is used during a subsequent iteration of the loop. This type of loop cannot be parallelized by the compiler.

recurrence

Occurs when a loop uses values computed in one iteration in subsequent iterations. These subsequent uses of the value imply loop-carried dependences and thus usually prevent parallelization. To increase parallelization, use linear recurrences.

reduction

A simple form of recurrence that reduces a large amount of data to a single value. It is commonly used to find the minimum and maximum elements of a vector. Although similar to a recurrence, it is easier to parallelize and uses less memory.

region

An area in code where threads are forked in order to perform a parallel operation. The region ends at the point where the threads join back together at the end of the parallel operation.