Netra™ Data Plane Software Suite 1.1 User's Guide

Sun Microsystems, Inc.
www.sun.com

Part No. 820-2374-10
July 2007, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Please
Recycle

Adobe PostScript

Contents

Tables

Preface

The *Netra Data Plane Software Suite 1.1 User's Guide* provides information regarding the operation and use of the Netra™ Data Plane Software Suite 1.1. This document is written for software engineers, developers, programmers, and users who have advanced experience with low-level programming.

How This Document Is Organized

Chapter 1 is an introduction to Netra Data Plane Software Suite 1.1, and provides installation and theoretical information.

Chapter 2 discusses some of the basic aspects of the `tejacc` compiler.

Chapter 3 is a tutorial to `tejacc` programming.

Chapter 4 summarizes using the debugger.

Chapter 5 discusses the Teja Profiler used in the Netra Data Plane software.

Chapter 6 describes interprocess communication (IPC) support.

Appendix A provides frequently asked questions regarding the Netra Data Plane software and how it interacts with the `tejacc` compiler.

Appendix B provides guidelines for diagnosing and tuning network applications.

Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:

  `http://docs.sun.com`

Shell Prompts

Shell	Prompt
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	`#`

Typographic Conventions

Typeface*	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

* The settings on your browser might differ from these settings.

Related Documentation

The documents listed as online are available at:

http://www.sun.com/documentation

Application	Title	Part Number	Format	Location
Operation	*Netra Data Plane Software Suite 1.1 User's Guide*	820-2374-10	PDF	online
Reference	*Netra Data Plane Software Suite 1.1 Reference Manual*	820-2375-10	PDF	online
Last-minute information	*Netra Data Plane Software Suite 1.1 Release Notes*	820-2376-10	PDF	online
Documentation Location	*Netra Data Plane Software Suite 1.1 Getting Started Guide*	820-2377-10	PDF	online

Reference Documentation

- *Developing and Tuning Applications on UltraSPARC T1 Chip Multithreading Systems*

  http://www.opensparc.net/publications/published-by-sun/developing-and-tuning-applications-on-ultrasparcr-t1-chip-multithreading-systems.html
- *Sun Studio 11: C User's Guide*

  http://docs.sun.com/app/docs/doc/819-3688
- *Netra Data Plane Software Suite 1.1 Reference Manual* (`tejacc` 4.0 for Sun CMT Reference Manual)

  Available in the Netra Data Plane Software Suite 1.1 release package
- *UltraSPARC T1 Supplement to UltraSPARC Architecture 2005 Specification*

  http://opensparc-t1.sunsource.net/index.html

Documentation, Support, and Training

Sun Function	URL
Documentation	http://www.sun.com/documentation/
Support	http://www.sun.com/support/
Training	http://www.sun.com/training/

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

http://www.sun.com/hwdocs/feedback

Please include the title and part number of your document with your feedback:

Netra Data Plane Software Suite 1.1 User's Guide, part number 820-2374-10

CHAPTER 1

Software Overview

This chapter is an introduction to the Netra Data Plane Software Suite 1.1, and provides installation and theoretical information. Topics include:

Product Description

The Netra Data Plane Software Suite 1.1 is a complete board software package solution. The software provides an optimized rapid development and runtime environment on top of multi-strand partitioning firmware for Sun CMT platforms. The software enables a scalable framework for fast-path network processing, encompassing the following features:

- Event-driven scheduling with run to completion states
- Explicit parallelization
- Static memory allocation
- Code generation based on hardware description and mapping
- Efficient communication pipes between pipeline states

The Netra Data Plane Software Suite 1.1 uses the `tejacc` compiler. `tejacc` is a component of the Teja NP 4.0 Software Platform, used to develop scalable, high-performance C applications for embedded multiprocessor target architectures.

tejacc operates on a system-level view of the application, through three techniques not usually found in a traditional language system:

- tejacc obtains the characteristics of the targeted hardware and software system architecture by executing a user-supplied architecture specification (context).
- tejacc simultaneously examines multiple sets of source files along with their relationships to the target architecture.
- tejacc recognizes APIs used in the application code, and generates them based on the system-level context.

The result is superior code validation and optimization, enabling more reliable and higher performance systems.

Software Installation

File Contents

The Netra Data Plane Software Suite 1.1 is distributed in the Netra_Data_Plane_Software_Suite_1.1.zip package for SPARC® platforms. TABLE 1-1 describes the contents of the unzipped package:

TABLE 1-1 SUNWndps and SUNWndpsd Package Contents

Directory	Contents
/opt/SUNWndps/bsp	Contains header files and low level code that initializes and manages Sun Fire T1000, Sun Fire T2000, Netra T2000 and Netra CP3060 systems.
/opt/SUNWndps/lib	Contains system-level libraries, such as CLI, IPC and LDoms/LDC.
/opt/SUNWndps/src	Contains RLP, ipfwd, udp (early access) and PacketClassifier reference applications and ophir driver sources. The description of Ethernet APIs can be found in src/dev/net/include/ethapi.h.
/opt/SUNWndps/tools	Contains the compiler and runtime system.
/opt/SUNWndpsd/bin/tnsmctl	Contains the Netra Data Plane CMT/IPC Share Memory Driver. Includes: /kernel/drv/sparcv9/tnsm /kernel/drv/tnsm.conf

Platform Firmware Prerequisites

For the Netra Data Plane Software Suite 1.1 to be supported, your system must have the appropriate, or newer, firmware installed.

▼ Checking Your OpenBoot PROM Firmware Version

- **As superuser, use the** `banner` **command to verify your version of the OpenBoot™ PROM firmware.**

  See the following three examples for each system supported:

```
ok banner
Sun Fire T2000, No Keyboard Copyright 2007 Sun Microsystems, Inc.
All rights reserved.
OpenBoot 4.26.1, 8064 MB memory available, Serial #64545116.
Ethernet address 0:3:ba:d8:e1:5c, Host ID: 83d8e15c.
```

```
ok banner
Netra T2000, No Keyboard Copyright 2007 Sun Microsystems, Inc. All
rights reserved.
OpenBoot 4.26.1, 8064 MB memory available, Serial #69940576.
Ethernet address 0:14:4f:2b:35:60, Host ID: 842b3560.
```

```
ok banner
Netra CP3060, No Keyboard Copyright 2007 Sun Microsystems, Inc.
All rights reserved.
OpenBoot 4.26.1, 16256 MB memory available, Serial #69061958.
Ethernet address 0:14:4f:1d:cd:46, Host ID: 841dcd46.
```

▼ Checking Your System Controller Firmware Versions

1. **As superuser, obtain the system controller prompt.**

```
# #.
sc>
```

Note – If your system has not been configured to access the system controller over the network, refer to your system's documentation for information on how to do so.

2. **Use the** `showsc -v version` **command to check your system controller firmware versions.**

   For example:

```
sc> showsc -v version
Advanced Lights Out Manager CMT v1.1.6
SC Firmware version: CMT 1.1.6
SC Bootmon version: CMT 1.1.6
VBSC 1.1.5
VBSC firmware built May 9 2006, 11:28:17
SC Bootmon Build Release: 01
SC bootmon checksum: 7416BA67
SC Bootmon built May 9 2006, 11:43:18
SC Build Release: 01
SC firmware checksum: 68D5EDEB
SC firmware built May 9 2006, 11:43:31
SC firmware flash update AUG 16 2006, 01:38:31
SC System Memory Size: 32MB
SC NVRAM Version=f
SC hardware type: 4
FPGA Version: 4.2.4.7
```

If your system controller firmware versions are lower or older than that provided in the example, contact your Sun Service representative to inquire about a firmware upgrade.

Package Dependencies

The software in the package has the following dependencies:

- The `SUNWndps` package depends on Sun Studio 11, Java version 1.5.0_04 and `gmake`. You must install these packages before applications are built.
- You must perform the debugger symbol resolution on the host using a tool called `dbghelper.pl`. This tool depends on and requires `dis`, `dbx`, and `perl` to be installed on the system.

Package Installation Procedures

Note – The `SUNWndps` software package is only supported on a SPARC system running the Solaris 10 Operating System.

Note – If you have previously installed an older version of the Netra Data Plane Software Suite 1.1, remove it before installing the new version. See "To Remove the Software" on page 6

▼ To Install the Software Into the Default Directory

1. **After downloading the Netra Data Plane Software Suite 1.1 from the web, as superuser, change to your download directory and go to** Step 2**.**

2. **Expand the zip file. Type:**

```
# unzip Netra_Data_Plane_Software_Suite_1.1.zip
```

3. **Install the** `SUNWndps` **and** `SUNWndpsd` **packages. Type:**

```
# /usr/sbin/pkgadd -d . SUNWndps SUNWndpsd
```

   The software is installed in the `/opt` directory.

4. **Use a text editor to add both the** `/opt/SUNWndps/tools/bin` **and** `/opt/SUNWndpsd/tools/bin` **directories to your** `PATH` **environment variable.**

▼ To Install the Software in a Directory Other Than the Default

1. **After downloading the Netra Data Plane Software Suite 1.1 from the web, as superuser, change to your download directory and go to** Step 2**.**

2. **Expand the zip file. Type:**

```
# unzip Netra_Data_Plane_Software_Suite_1.1.zip
```

3. **Add the** `SUNWndps` **and** `SUNWndpsd` **packages to** *your_directory*. **Type:**

```
# pkgadd -d `pwd` -R /usr/local/your_directory SUNWndps SUNWndpsd
```

The software is installed in the `/usr/local/`*your_directory* directory.

4. **Open the** `/usr/local/`*your_directory*`/opt/SUNWndps/tools/bin/tejacc.sh` **file in a text editor and find the following line:**

```
export TEJA_INSTALL_DIR=/opt/SUNWndps/tools
```

5. **Change the line in** Step 4 **to:**

```
export TEJA_INSTALL_DIR=/usr/local/your_directory/opt/SUNWndps/tools
```

6. **Use a text editor to add the** `/usr/local/`*your_directory*`/opt/SUNWndps/tools/bin` **directory to your** `PATH` **environment variable.**

▼ To Remove the Software

- **To remove the** `SUNWndps` **and** `SUNWndpsd` **packages, as superuser, type:**

```
# /usr/sbin/pkgrm SUNWndps SUNWndpsd
```

The Netra Data Plane Software Suite 1.1 is removed.

Note – For more details about using the `pkgadd` and `pkgrm` commands, see the man pages.

Building and Booting Reference Applications

The instructions for building reference applications are located in the application directories. TABLE 1-2 lists the directories and instructional file.

TABLE 1-2 Reference Application Instruction Files

Reference Applications	Building Instruction Location
`ipfwd`	`/SUNWndps/src/apps/ipfwd/README`
`ipfwd_ldom`	`/SUNWndps/src/apps/ipfwd_ldom/README` `/SUNWndps/src/apps/ipfwd_ldom/README.config`
`remotecli`	`/SUNWndps/src/apps/remotecli/README.remotecli`
`udp`	`/SUNWndps/src/apps/udp/README`
Teja(R) Tutorial	`/SUNWndps/tools/examples/PacketClassifier/README`

The application image is booted over the network. Ensure the target system is configured for network boot. The command syntax is:

`boot` *network_device*`:[dhcp|bootp,][`*server_ip*`],[`*boot_filename*`],[`*client_ip*`],[`*router_ip*`],[`*boot_retries*`],[`*tftp_retries*`],[`*subnet_mask*`],[`*boot_arguments*`]`

TABLE 1-3 describes the optional parameters.

TABLE 1-3 Boot Optional Parameters

Option	Description
network_device	The network device used to boot the system.
`dhcp|bootp`	Use DHCP or BOOTP address discovery protocols for boot. Unless configured otherwise, RARP is used as the default address discovery protocol.
server_ip	The IP address of the DHCP, BOOTP, or RARP server.
boot_filename	The file name of the boot script file or boot application image.
client_ip	The IP address of the system being booted.

TABLE 1-3 Boot Optional Parameters *(Continued)*

Option	Description
router_ip	The IP address of a router between the client and server.
boot_retries	Number of times the boot process is attempted.
tftp_retries	Number of times that the TFTP protocol attempts to retrieve the MAC address.
subnet_mask	The subnet mask of the client.
boot_arguments	Additonal arguments used for boot.

Note – For the `boot` command, commas are required to demark missing parameters unless the parameters are at end of the list.

▼ To Boot an Application Image

1. **Copy the application image to the `tftpboot` directory of the boot server.**
2. **As superuser, obtain the `ok` prompt on the server that is to boot.**

   For example:

```
# sync; init 0
```

3. **From the `ok` prompt, type one of the following commands:**
   - To boot using RARP, type:

```
ok> boot network_device:,boot_filename [-v]
```

   - To boot using DHCP, type:

```
ok> boot network_device:dhcp,server_ip,boot_filename [-v]
```

Note – The `-v` argument is an optional verbose flag.

Programming Methodology

In Teja NP, you write an application with multiple C programs that execute in parallel and coordinate with each other. The application is targeted to multiprocessor architectures with shared resources. Ideally, the applications are written to be used in several projects and architectures. Additionally, the Teja NP attains maximum performance in the target mapping.

When writing the application, you:

- Are aware of the multiple threads of the application.
- Must protect critical regions of the code by using mutual exclusion primitives.
- Must communicate structured data using polled queues or event-driven channels.
- Must allocate memory efficiently in a unified manner using memory pools.

`tejacc` provides the constructs of threads, mutex, queue, channel, and memory pool within the application code. These constructs enable you to specify coordinated parallel behavior in a target-independent, reusable manner. When the application is mapped to a specific target, `tejacc` generates optimized, target-specific code. The constructs and their associated API is called *late-binding*.

One technique for scaling performance is to organize the application in a parallel-pipeline matrix. This technique is effective when the processed data is in the form of independent packets. For this technique, the processing loop is broken up into multiple stages and the stages are pipelined. For example, in an N-stage pipeline, while stage N is processing packet k, stage (N - 1) is processing packet (k + 1), and so on. In order to scale performance even further and balance the pipeline, each stage can run its code multiple times in parallel, yielding an application-specific parallel-pipeline matrix.

There are several issues with this technique. The most important being where to break the original processing loop into stages. This choice is dictated by several factors:

- Natural partitioning points in the application functionality
- Structure of the application code
- Balance in the execution time of the different stages
- Ease of design and transferability of the context information from one stage to the next

The context carried over from one stage to the next is reduced when the stack is empty at the end of that stage. Applications written with modular functions are more flexible for such architecture exploration. During the processing of a context, the code might wait for the completion of some long-latency operation, such as I/O. During the wait, the code could switch to another available data context. While applicable to most targets, such a technique is important when the processor does

not support hardware multithreading. If the stack is empty when the context is switched, the context information is minimized. Performance is improved as code modularity becomes more granular.

Expressing the flow of code as state machines (finite state automata) enables multiple levels of modularity and fine-grained architecture exploration.

Reusing Existing C Code

Standardized C programs can be compiled using `tejacc` without change. Two methods are available to reuse C code with `tejacc`:

- Create libraries from existing C code and compile new C code to call these libraries. This method requires that the libraries are available for the target system and code changes are to be minimized.
- Substitute system and application calls with calls to the Teja user application API and compile using `tejacc`. Use this method when the libraries are not available for the target system or when performance improvements are desired.

Increasing the execution performance of existing C programs on multicore architectures requires targeting for parallel-pipeline execution. This process is iterative.

In the first iteration, some program functions are mapped to a second and additional processors, executing in parallel. All threads of execution operate on the *same* copy of the shared global data structures, with mutual exclusion primitives for protection.

In the second iteration, each thread operates on its *own* copy of the global data structures, leaving the others as shared. The threads coordinate with each other using both mutual exclusion and communication messages.

In the final iteration, each thread runs its functions in a loop, operating on a stream of data to be processed.

By using this method, the bulk of the application code is reused while small changes are made to the overall control flow and coordination.

`tejacc` Compiler Basic Operation

C code developers are familiar with a compiler that takes a C source file and generates an object file. When multiple source files are submitted to the compiler, it processes the source files one by one. The `tejacc` compiler extends this model to a system-level, multifile process for a multiprocessor target.

tejacc Compiler Mechanics

The basic function of tejacc is to take multiple sets of user application source files and produce multiple sets of generated files. When processed by target-specific compilers or assemblers, these generated file sets produce images that are loaded into the processors of the target architecture. All user source files must adhere to the C syntax (see "Language" on page 31 for the language reference). The translation of the source to the image is governed by options that control or configure the behavior of tejacc.

tejacc is a command-line program suitable for batch processing. For example:

```
tejacc options -srcset mysrcset file1 file2 -srcset yoursrcset file2 file3 file4
```

In this example, there are two sets of source files, *mysrcset* and *yoursrcset*. The files in *mysrcset* are *file1* and *file2*, and the files in *yoursrcset* are *file2*, *file3,* and *file4*. *file2* intentionally appears in both source sets.

file2 defines a global variable, *myglobal*, whose scope is the source file set. This situation means that tejacc allocates two locations for *myglobal*, one within *mysrcset* and the other within *yoursrcset*. References to *myglobal* within *mysrcset* resolve to the first location, and references to *myglobal* within *yoursrcset* resolve to the second location.

A source set can be associated to one or more application processes. In that case, the source set is compiled several times and the global variable is scoped to the respective process address space. An application process can also have multiple source sets associated to it.

Each source set can have a set of compile options. For example:

```
tejacc options -srcset mysrcset -D mydefine file1 file2 -srcset yoursrcset -D mydefine -I
mydir/include file2 file3 file4
```

In this example, when *mysrcset* is compiled, tejacc defines the symbol *mydefine* for *file1* and *file2*. Similarly, when *yoursrcset* is compiled, tejacc defines the symbol *mydefine* and searches the *mydir*/include directory for *file2*, *file3* and *file4*.

When a particular option is applied to every set of source files, that option is declared to tejacc before any source set is specified. For example:

```
tejacc -D mydefine other_options -srcset mysrcset file1 file2 -srcset yoursrcset -I
mydir/include file2 file3 file4
```

In this example, the definition of *mydefine* is factored into the options passed to `tejacc`.

TABLE 1-4 lists some options to `tejacc`:

TABLE 1-4 Some Options to `tejacc`

Option	Comment
`-include` *includefile*	Where *includefile* is included in each file in each source set to facilitate the inclusion of common system files of the application or the target system.
`-I` *includedir*	Where *includedir* is searched for each file in each source set.
`-d` *destdir*	Where the compilation outputs are placed in a directory tree with *destdir* as the root.

`tejacc` Compiler Configuration

In addition to the `tejacc` mechanics and options, the behavior of `tejacc` is configured by user libraries that are dynamically linked into `tejacc`. The libraries describe to `tejacc` the target hardware architecture, the target software architecture, and the mapping of the variables and functions in the source set files to the target architecture. TABLE 1-5 describes some of the configuration options of `tejacc`.

TABLE 1-5 Configuration Options to `tejacc`

Option	Comment
`-hwarch` *myhwarchlib*, *myhwarch*	Load the *myhwarchlib* shared library and execute the function *myhwarch()* in it. The execution of *myhwarch()* creates a memory model of the target hardware architecture.
`-swarch` *myswarchlib*, *myswarch*	Load the *myswarchlib* shared library and execute the function *myswarch()* in it. The execution of *myswarch()* creates a memory model of the target software architecture.
`-map` *mymaplib*, *mymap*	Load the *mymaplib* shared library and execute the function *mymap()* in it. Executing the *mymap()* function in the `mymaplib` shared library creates a memory model of the application source code mapping to the target architecture.

The three entry point functions into the shared library files take no parameters and return an `int`.

The shared library files can be used for multiple configuration options, but the entry point for each option must be unique, take no parameters, and return an `int`. The trade-off is the ease of maintaining fewer libraries versus the speed of updating only one of several libraries.

Once the memory models are created, `tejacc` parses and analyzes the source sets and generates code for the source sets within the context of the models. Using the system-level information `tejacc` obtains from the models, in conjunction with specific API calls made in the user's source files, `tejacc` can apply a variety of validation and optimization techniques during code generation. The output by `tejacc` is source code as input to the target-specific compilers. Although the compiler-generated code is available for inspection or debugging, you should not modify this code.

`tejacc` Compiler and Teja NP Interaction

Figure 1.1 shows the interaction of `tejacc` with the other platform components of Teja NP.

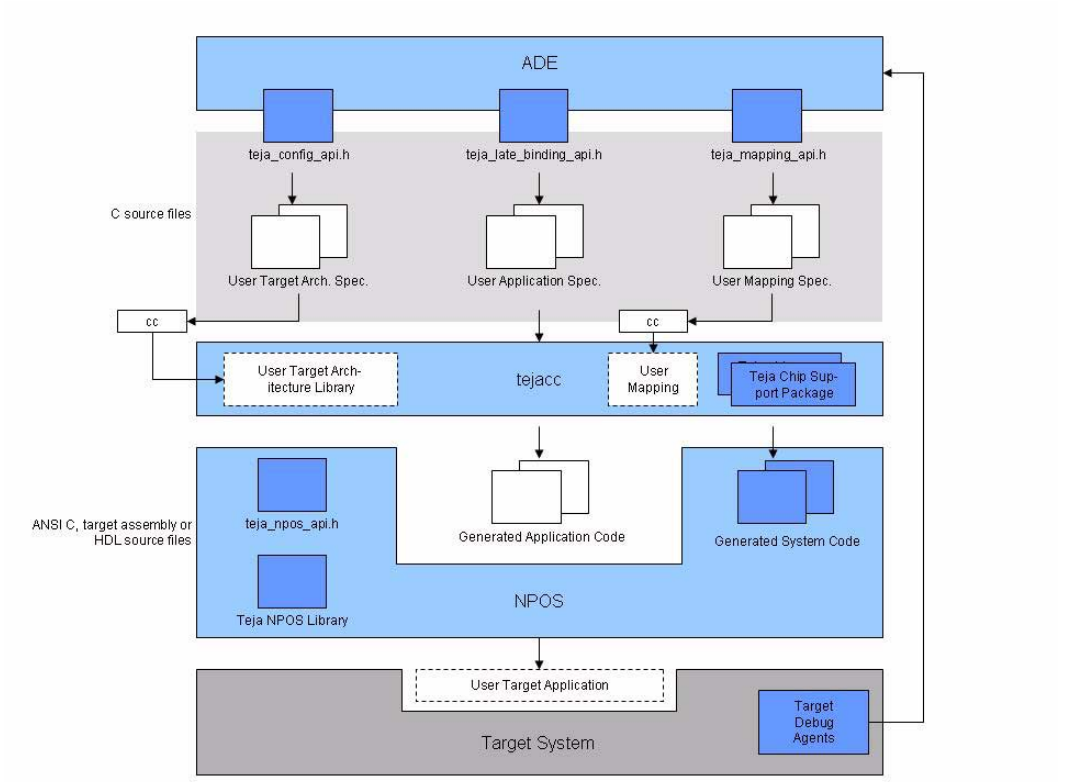

FIGURE 1-1 Teja NP 4.0 Overview Diagram

You create the dynamically linked shared libraries for the hardware architecture, software architecture, and map by writing C programs using the Teja Hardware Architecture API, the Teja Software Architecture API, and the Teja Map API respectively. The C programs are compiled and linked into dynamically linked shared libraries using the C compiler.

Your application source files might contain calls to the Teja Late-Binding API and the Teja NPOS API. `tejacc` is aware of the Late-Binding API. Depending on the context of the target hardware, software architecture, and the mapping, `tejacc` generates code for the Late-Binding API calls. The calls are optimized for the specific situation described in the system context. `tejacc` is not aware of the NPOS API, and calls to it pass to the generated code where the calls are either macro expanded (if defined in the NPOS library include file) or linked to the target-specific NPOS library.

Teja NP also provides a graphical application development environment (ADE) to visualize and manipulate applications. Description of the ADE is not within the scope of this document.

Architecture Elements

Hardware Architecture Overview

The Hardware Architecture API is used to describe target hardware architectures. A hardware architecture is comprised of processors, memories, buses, hardware objects, ports, address spaces, address ranges, and the connectivity among all these elements. A hardware architecture might also contain other hardware architectures, thereby enabling hierarchical description of complex and scalable architectures.

Most users will not need to specify the hardware architectures as the Teja NP platform is predefined. Only in the situation of a custom hardware architecture is the API used.

Note – The hardware architecture API runs on the development host in the context of the compiler and is not a target API.

Hardware Architecture Elements

Hardware architecture elements are building blocks that appear in almost all architectures. Each element is defined using the relevant create function, of the form: `teja_type_create()`. You can assign values to the properties of each function using the `teja_type_set_property()` and `teja_type_get_property()` functions.

TABLE 1-6 describes basic hardware architecture elements.

TABLE 1-6 Basic Hardware Architecture Elements

Element	Description
Hardware architecture	A hardware architecture is a container of architecture elements. It has a user-defined name, which must be unique in its container, and a type, which indicates whether its contents are predefined by `tejacc` or defined by the user. Various types of architectures are predefined in the `teja_hardware_architecture.h` file and are understood by `tejacc`. You cannot modify a predefined architecture. User-defined architectures are sometimes desirable to prevent application developers from modifying an architecture. You can achieve this by first populating the architecture and then calling the `teja_architecture_set_read_only()` function.
Processor	A processor is a target for running an operating system. A processor is contained in an architecture, which provides it a name and type.
Memory	A memory is a target for mapping program variables. A memory is contained in an architecture, which provides it a name and type.
Hardware object	A hardware object is a logic block that is either known to `tejacc` (for example, TCAM) or is a target for user-defined hardware logic. A hardware object is contained in an architecture, which provides it a name and type.
Bus	A bus is used to interconnect elements in a hardware architecture. `tejacc` uses connection information to validate the user application and reachability information to optimize the generated code. A bus is contained in an architecture, which provides it a name, type, and indicates whether the bus is exported. That is, the bus is visible outside of the containing architecture.

Relationships

An architecture can contain other architectures, processors, memories, hardware objects, and buses. The respective create function for a given element indicates the containment relationship. An architecture, a processor, a memory, and a hardware object can connect to a bus using `teja_type_connect()` functions.

Utility Functions

Utility functions are provided to look up a named element within an architecture, set the value of a property, and get the value of a property. These actions are accomplished with the `teja_lookup_type()`, `teja_type_set_property()`, and

`teja_type_get_property()` functions respectively. Properties are set to select or influence specific validation, code generation, or optimization algorithms in `tejacc`. Each property and its effect is described in the *Netra Data Plane Software Suite 1.1 Reference Manual*.

Advanced Hardware Architecture Elements

Some hardware architecture elements are available for advanced users and might not be needed for all targets. Each element is defined using the relevant create function, of the form: `teja_type_create()`. You can assign values to their properties using the `teja_type_set_property()` and `teja_type_get_property()` functions.

TABLE 1-7 describes advanced hardware architecture elements.

TABLE 1-7 Advanced Hardware Architecture Elements

Element	Description
Port	A bus is a collection of signals in the hardware with a certain protocol for using the signals. When an element connects to a bus, ports on the element tap into the bus. The port exposes a level of detail hidden by the bus. In some configurable target architectures, this action is necessary because certain signals need to be connected to handles within the user's architecture specification. A port is also a handle on an architecture for connecting to another port. A port is contained in an architecture, which provides the port a name and direction. Elements such as processors, memory, buses, or hardware objects also have ports, though these ports are predefined within the element. When a port is connected to a signal, it is given a value that is the name of that signal. See the `teja_type_set_port()` function in the *Netra Data Plane Software Suite 1.1 Reference Manual*. A port on an architecture might connect to a signal within the architecture as well. See the `teja_architecture_set_port_internal()` function in the *Netra Data Plane Software Suite 1.1 Reference Manual*.

TABLE 1-7 Advanced Hardware Architecture Elements *(Continued)*

Element	Description
Address space and address range	In a complex network of shared memories and processors sharing them, the addressing scheme is not obvious. Address spaces and ranges are used to specify abstract requirements for shared memory access. `tejacc` assigns actual values to the address spaces and ranges by resolving these requirements. An address space is an abstract region of contiguous memory used as a context for allocating address ranges. An address space is contained in an architecture, which provides it a name, a base address, and a high address. The `teja_address_space_join()` facility can join two address spaces. When their constraints are merged, more stringent resolution is required, as each of the original address spaces refers to the same joined address space. An address range is a region of contiguous memory within an address space. An address range is contained in an address space that specifies its size. The address range might be generic, or constrained by specific address values, alignment, and other requirements.

Software Architecture and Late-Binding Overview

A software architecture is comprised of operating systems, processes, threads, mutexes, queues, channels, memory pools, and the relationships among these elements.

A subgroup of the software architecture elements is defined in the software architecture description and used in the application code. This subgroup consists of mutex, queue, channel, and memory pool. The software architecture part of the API runs on the development host in the context of the compiler. The application part of the API runs on the target. The API that uses elements of the subgroup in the application code is called late-binding and it is treated specially by `tejacc`.

The Late-Binding API offers the functionality of mutual exclusion, queuing, sending and receiving messages, memory management, and interruptible wait. The functions in this API are known to `tejacc`. `tejacc` generates the implementation of this functionality in a context-sensitive manner. The context that `tejacc` uses to generate the implementation consists of:

- Global system description of hardware and software
- Constant parameters that are known at compile time
- User provided hints

You can choose the implementation of a late-binding object. For example, a communication channel could be implemented as a shared memory circular buffer or as a TCP/IP socket. You can also indicate how many producers and consumers a

certain queue has, affecting the way late-binding API code is generated. For example, if a communication channel is used by one producer and one consumer, `tejacc` can generate the read/write calls from and to this channel as a mutex-free circular buffer. If there are two producers and one consumer, `tejacc` generates an implementation that is protected by a mutex on the sending side.

The advantage of this method over precompiled libraries is that system functions contain only the minimal necessary code. Otherwise, a comprehensive, generic algorithm must account for all possible execution paths at runtime.

If the channel ID is passed to the channel function as a constant, then `tejacc` knows all the characteristics of the channel and can generate the unique, minimal code for each call to that channel function. If the channel ID is a variable, then `tejacc` must generate a switch statement and the implementation must be picked at runtime.

Regardless of the method you prefer, you can modify the context without touching the application code, as the Late-Binding API is completely target independent. This flexibility enables different software configurations at optimization time without changing the algorithmic part of the program.

Note – The software architecture API runs on the development host in the context of the compiler and is not a target API. The Late-Binding API runs on the target and not on the development host.

Late-Binding Elements

You declare each of the late-binding objects (mutex, queue, channel, and memory pool) using the `teja_type_declare()` function. You can assign values to the properties of most of these elements using the `teja_type_set_property()` and `teja_type_get_property()` functions.

Each of these objects has an identifier indicated by the user as a string in the software architecture using the `declare()` function. In the application code, the element is labeled with a C identifier and not a string. `tejacc` reads the string from the software architecture and transforms it in a `#define` for the application code. The transformation from string to preprocessor macro is part of the interaction between the software architecture and the application code.

Multiple target-specific (custom) implementations of the late-binding objects are available. Refer to the *Netra Data Plane Software Suite 1.1 Reference Manual* for a full list of these custom implementations. Every implementation has the same semantics but different algorithms. Choosing the right custom implementation and related parameters is important at optimization time.

For example, with mutex, one custom implementation might provide fair access while another might be unfair. In another example, a channel with multiple consumers might not broadcast the same message to all consumers.

TABLE 1-8 describes the late-binding elements.

TABLE 1-8 Late-Binding Elements

Late-Binding Element	Description
Mutex	The mutex element provides mutual exclusion functionality and is used to protect critical regions of code. The Late-Binding API for mutex consists of: • `teja_mutex_lock()` – Lock a mutex. • `teja_mutex_trylock()` – Try and lock a mutex without blocking. • `teja_mutex_unlock()` – Unlock a mutex.
Queue	The queue element provides thread safe and atomic enqueue and dequeue API functions for storing and accessing nodes* in a first-in-first-out method. The Late-Binding API for queue consists of: • `teja_queue_dequeue()` – Dequeue an element from a queue. • `teja_queue_enqueue()` – Enqueue an element to a queue. • `teja_queue_is_empty()` – Check for queue emptiness. • `teja_queue_get_size()\` – Obtain queue size
Memory pool	Memory pools provide an efficient, thread-safe, cross-platform memory management system. This system requires you to subdivide memory in preallocated pools. A memory pool is a set of user-defined, same-size contiguous memory nodes. At runtime, you can get a node from, or put a node to, a memory pool. This mechanism is more efficient at dynamic allocation than the traditional `malloc()` and `free()` calls. Sometimes the application needs to match accesses to two memory pools. Given a buffer from one memory pool, obtain the memory pool's index value and then obtain the node with the same index value from the other memory pool. The Late-Binding API for memory pool consists of: • `teja_memory_pool_get_node()` – Get a new node from the pool. • `teja_memory_pool_put_node()` – Return a node to the pool. • `teja_memory_pool_get_node_from_index()` – Provide a pointer to a node, given its sequential index. • `teja_memory_pool_get_index_from_node()` – Provide the sequential index of a node, given its pointer.

TABLE 1-8 Late-Binding Elements *(Continued)*

Late-Binding Element	Description
Channel	The Channel API is used to establish connections among threads, to inspect connection state, and to exchange data across threads. Channels are logical communication mediums between two or more threads. Threads sending messages to a channel are called producers, threads receiving messages from a channel are called consumers. Channels are unidirectional, and they can have multiple producers and consumers. The semantics of channels are that of a pipe. Data is copied into the channel at the sender and is copied out of the channel at the receiver. It is possible to send a pointer over a channel, as the pointer value is simply copied into the channel as data. When pointers are sent across the channel, ensure that the consumer has access to the same memory or is able to convert the pointer to access that same memory. The Late-Binding API for channel consists of: • `teja_channel_is_connection_open()`[d] – Check if a connection on a channel is open. • `teja_channel_make_connection()` – Establish a connection on a channel. • `teja_channel_break_connection()` – Break a connection on a channel. • `teja_channel_send()` – Send data on a channel. • `teja_wait()` – Wait on timeout and a list of channels. If data arrives on channels before timeout expires, read it.

* The first word of the node that is enqueued is allowed to be overwritten by the queue implementation.

\ `teja_queue_get_size()` is only meant for debugging purposes.

d Connection functions are only available on channels that support the concept of connection, such as the TCP/IP channel. For connectionless channels, these operations are empty.

Other Elements

Each of the non-late-binding elements can be defined using the relevant `teja_type_create()` create function.

Use the `teja_type_set_property()` and `teja_type_get_property()` functions to assign values to the properties of most of these elements.

TABLE 1-9 describes other elements.

TABLE 1-9 Other Elements

Other Element	Description
Operating system	An operating system runs on processors and is a target for running processes. An OS has a name and type. One of the operating system types defined in `tejacc` states that no operating system is run on the given processors, implying that the application will run on bare silicon.
Process	A process runs on an operating system and is a target for running threads. All threads in a process share an address space. The process has a name and lists the names of source sets that contain the application code to be compiled for the process.
Thread	A thread runs in a process and is a target for executing a function. A thread has a name.

Utility Functions

Utility functions are provided to look up a named element within an architecture, set the value of a property, and get the value of a property. These actions are accomplished with the `teja_lookup_type()`, `teja_type_set_property()`, and `teja_type_get_property()` functions respectively. Set properties to select or influence specific validation, code generation, or optimization algorithms in `tejacc`. Each property and its effect is described in the *Netra Data Plane Software Suite 1.1 Reference Manual*.

User API Overview

This section gives an overview of the Teja NP API for writing the user application files in the source sets given to `tejacc`. This API is executed on the target.

Overview of the Late-Binding API

The Late-Binding API provides primitives for the synchronization of distributed threads, communication, and memory allocation. This API is treated specially by the `tejacc` compiler and it is generated on the fly based on contextual information. See "Late-Binding Elements" on page 19 for more details. A complete reference of this API is provided in the *Netra Data Plane Software Suite 1.1 Reference Manual*.

NPOS API Overview

The NPOS API consists of portable, target-independent abstractions over various operating system facilities such as thread management, heap-based memory management, time management, socket communication, and file descriptor registration and handling. Unlike late-binding APIs, NPOS APIs are not treated specially by the compiler and are implemented in precompiled libraries.

The memory management functions offer `malloc` and `free` functionality. These functions are computationally expensive, and should only be used in initialization code or non-relative critical code. On bare hardware targets the `free()` function is an empty operation, so only `malloc()` should be used to obtain memory that is not meant to be released. For all other purposes, the memory pool API should be used.

The thread management functions offer the ability to start and end threads dynamically.

The time management functions offer the ability to measure time.

The socket communication functions offer an abstraction over connection and non-connection oriented socket communication.

The signal handling functions offer the ability to register Teja signals with a handler function. Teja signals can be sent to a destination thread that runs in the same process as the sender. These functions are cross-platform, so they can also be used on systems that do not support UNIX-like signaling mechanism. Signal handling functions are more efficient than OS signals, and unlike OS signals, their associated handler is called synchronously.

Any function can be safely called from within the handler. This ability removes the limitations of asynchronous handling. Even in case the registered signal is a valid OS signal code, when the application receives an actual OS signal, the handler will still be called synchronously. If a Teja process running multiple threads receives an OS signal, every one of its threads will receive the signal.

Since Teja signals are handled synchronously, threads can only receive signals and execute their registered handler when the thread is in an interruptible state given by the `teja_wait()` function.

Any positive integer is a valid Teja signal code that can be passed to the registration function. However, if the signal code is also a valid OS code, such as `SIGUSR1` on UNIX, the signal is also registered using the native OS mechanism. The thread will react to OS signals as well as to Teja signals.

A typical Teja signal handler reads any data from the relevant source and returns the data to the caller. The caller is `teja_wait()`, which in turn exits and returns the data to user program.

Registration of file descriptors has some similarities to registration of signals. The operation registers a `fd` with the system and associates the `fd` with a user-defined handler and optionally with a context, which is a user-defined value (for example, a pointer). Whenever data is received on the `fd`, the system automatically executes the associated handler and passes to it the context.

Just like signal handlers, file descriptor handlers are called synchronously, so any function can be safely called from within the handler. This ability removes the limitations of asynchronous handling.

Since `fd` handlers are called synchronously, threads can only receive `fd` input and execute their registered handler when the thread is in an interruptible state given by the `teja_wait()` function.

An `fd` handler reads the data from the fd and returns it to `teja_wait()`, which in turn returns the data to the user application.

A complete reference of the NPOS API is provided in the *Netra Data Plane Software Suite 1.1 Reference Manual*.

Finite State Machine API Overview

The Finite State Machine API enables easy modularization and pipelining of code. Finite state machines are used to organize the control flow of code execution in an application. State machine support is through various macros, which are expanded before they reach `tejacc`. While `tejacc` does not recognize these macros, higher level tools such as Teja NP ADE might impose additional formatting restrictions on how these macros are used.

A complete reference of the state machine API is given in the *Netra Data Plane Software Suite 1.1 Reference Manual*. The API includes facilities to:

- Declare a state machine
- Begin and end the state machine
- Declare the state machine's states
- Begin and end each state with the block of code to be executed in that state
- Declare the start state
- Transition from one state to the next

Map API Overview

The Map API is used to map elements of the user's source files to the target architecture. TABLE 1-10 describes these relationships.

TABLE 1-10 Mapping of Elements

Elements	Mapping
Functions	Are mapped to threads with the `teja_map_function_to_thread()` function.
Variables	Are mapped to memories or process address spaces with the `teja_map_variable_to_memory()` and `teja_map_variables_to_memory()` functions.
Processors	Are initialized with the `teja_map_initialization_function_to_processor()` function.
Mapping-specific properties	Are assigned with the `teja_mapping_set_property()` function.

If a variable is mapped multiple times, the last mapping is used. This functionality enables you to specify a general class of mappings using a regular expression and then refine the mapping for a specific variable.

CHAPTER 2

`tejacc` Basics

This chapter discusses some of the basic aspects of the `tejacc` compiler. Topics include:

Command-Line Options

The `tejacc` command-line syntax is as follows:

`tejacc` *common_options* `[-srcset` *name srcset_options source_files*`]+`

where:

- *common_options* are the options that apply to `tejacc` or options that apply to all source files.
- *name* is the name of the source set.
- *srcset_options* are the options that are applied only to the source set.
- *source_files* are the files used to create the source set.

`-srcset` creates a source set that can be mapped to one or more processes. Additionally, one or more source sets can be created.

tejacc Command-Line Options

TABLE 2-1 tejacc Options

Option	Description
`-hwarch hwarch_lib,hwarch_function`	The *hwarch_function* from the dynamic shared library *hwarch_lib* is executed to create a memory model of the target hardware architecture representation on which the generated application is run. There are no default values for this option and it is mandatory.
`-swarch` *swarch_lib*, *swarch_function*	The *swarch_function* from the dynamic shared library *swarch_lib* is executed to create a memory model of the target software architecture representation on which the generated application is mapped. There are no default values for this option and it is mandatory.
`-map` *map_lib*, *map_function*	The *map_function* from the dynamic shared library *map_lib* is executed to create a mapping between the user application, software architecture, and hardware architecture. There are no default values for this option and it is mandatory.
`-D` *name*`[=`*definition*`]`	Redefines *name* as a macro, with *definition* or 1 if not specified. This option is applied to the preprocessing stage of the compilation.
`-include` *includefile*	Processes *includefile* as if `#include "file"` appeared as the first line of the primary source file.
`-I` *includedir*	Adds the directory *includedir* to the head of the list of directories to be searched for header files.
`-E`	Prints preprocessed output to the `stdout` and stops any further processing.
`-w`	Suppresses all warnings.
`-d` *destdir*	Specifies the destination directory for the generated code. The default value is the *current_dir* `/code`.
`-O`	Enables optimizations. All applicable optimizations are used for code generation.
`-fcontext-sensitive-generation`	Enables context-sensitive code generation optimization. The generated Late-Binding API implementation has separate implementations for every context and enables inlining through the target compiler.
`-pg`	Enables profiling. Calling the profiling API in the source files generates target-specific code to enable profiling and collect data. If the `-pg` option is not specified, the profiling API is not called.

TABLE 2-1 `tejacc` Options *(Continued)*

Option	Description
`-h, ?h, -help, ?help`	Prints `tejacc` usage.
`-srcset` *srcset_namesrcset_specific_optionssource_files*	Defines a sourceset consisting of one or more source files. The sourceset is used to map to one or more processes. *srcset_specific_options* are applied only to the files listed in the *source_files*. The `-D`, `-I` and `-include` options are also part of the source set specific options.
`-finline=`*comma separated list of functions*	This option is only applicable to the source set and tries to inline the functions that are specified in the list. There are no errors or warnings if a listed function is not found in the sources.

Optimization

Optimization Options

You can use the following command-line switches to `tejacc` to enable optimization:

- `-O` — enables all optimizations
- `-fcontext-sensitive-generation` — enables context sensitive generation only

TABLE 2-2 lists the available optimizations for the `tejacc` compiler.

TABLE 2-2 Optimizations for `tejacc`

Optimization	Comment
Context-sensitive generation	Affects all late-binding functions. See "Late-Binding Elements" on page 19. These functions are generated from contextual information such as constant parameters known to the compiler and global information from software architecture, hardware architecture, and mapping.

TABLE 2-2 Optimizations for `tejacc` *(Continued)*

Optimization	Comment
Global inlining	Functions marked with the `inline` keyword get inlined throughout the entire application, even across files.
Reachability	Unused functions and variables are not generated, saving code space.
Target compiler optimizations	—

Context-Sensitive Generation

All late-binding APIs and profiler APIs benefit from context-sensitive generation.

▼ To Enable Optimization

1. **Add the appropriate switch to the `tejacc` command-line.**

   Refer to "Optimization Options" on page 29.

2. **Use constants in late-binding calls that you want to optimize.**

   In particular:

   - For channel, mutex, queue, and mempool functions, make sure the late-binding object you are passing is constant. You can increase the performance for channels with a circular buffer-based implementation. When you use a fixed and constant message size (1, 2, 4, or 8) for all `teja_channel_send` calls on a given circular buffer based channel c, the code generator detects the condition and uses a unique and very fast implementation of the buffer.
   - For `teja_wait`, ensure that the four time parameters are constant and that any channels passed are constant.

   If these two conditions are not met for a given function call, that function call is generated without context-sensitive optimization.

Language

Language Characteristics

The `tejacc` compiler front-end parses a subset of extended C as defined by `gcc`. However, there are some limitations, as follows:

- The compiler does not parse K and R syntax for function declaration.
- `tejacc` does not assign integer types to variables by default.
- The compiler does not support undeclared functions and does not default to type `int`.
- `tejacc` implements strict type checking and might return warnings or errors in the situation of a type mismatch.
- Though the `tejacc` compiler recognizes a subset of extended C, for interoperability, the compiler supports the language that is used by the target complier.

Include Files

For each user source file, the `teja_include_all.h` file is always included before any other include or C code is preprocessed. The `teja_include_all.h` file is located in the `include/runtime/`*target_processor_name*`/`*target_os_name* directory. This directory also contains other target-dependent include files.

Late-Binding Object Identifiers

Late-binding objects such as channels, memory pools, queues, and mutexes, are created in the software architecture. The late-binding API described in the file `teja_late_binding.h` provides operations on these objects and is called inside the user application source code.

The mechanism to access late-binding objects in the user application code is to use them as C preprocessor symbols that have the same names as the strings that were used to create the late-binding objects in the software architecture. The `tejacc` compiler creates a set of defines for these late-binding object identifiers and passes them to the command-line during the compilation.

The list of C preprocessor symbols are generated in the `reports/`*process_name*`_predefined_symbols.h` file.

CHAPTER 3

Tutorial

This chapter is a tutorial to `tejacc` programming. This chapter addresses the following topics:

Application Code

The application used for the tutorial has two threads, `tick` and `tock`. The `tick` thread sends a countdown (9, 8, ..., 0) to the `tock` thread using a channel. Both of the threads run in a single process called `ticktock`.

The application code is a file called `ticktock.c`. The application code has a `ticker` function for the `tick` thread, and a `tocker` function for the `tock` thread. TABLE 3-1 lists the `ticktock.c` file and provides comment.

TABLE 3-1 `ticktock.c` File and Comments

`#include <stdio.h>` `#include "teja_late_binding.h"`	*`stdio.h` and `teja_late_binding.h` are included, this action declares the Teja NP late-binding API.*

`void`
`ticker(void)`
`{`
`  short i;`
`  char * node = 0;`
`  int ret;`
`  for(i=9; i>=0; i--) {`
`  teja_wait_time(1, 0);`
`  node = (char *) teja_memory_pool_get_node (tick_memory_pool);`
`  if (!node) {`
`  printf ("Memory pool is empty!");`
`  continue;`
`  }`
`  sprintf(node, "%d...", i);`
`  do {`
`  ret = teja_channel_send(ticktock_channel, i, &node, size of (char *));`
`  if (ret < 0) {`
`  printf("Failed to send %s\n", node);`
`  } else {`
`  printf("%s sent\n", node);`
`  }`
`  } while (ret < 0); /* if channel full, spin & keep trying */`
`  }`
`}`
	The `ticker` function uses two late-binding objects: a memory pool called `tick_memory_pool` and a channel called `ticktock_channel`. These are declared in the software architecture definition The function loops ten times, sending the count over the `ticktock_channel` once every second. `teja_wait_time` is a macro of `teja_wait` defined in the `teja_late_binding.h` file.

`void`
`tocker(void)`
`{`
`  short i;`
`  char * node = 0;`
`  while(1) {`
`  teja_wait(TEJA_INFINITE_WAIT, 0, 0, (int) 1E8,`
`  &i, (void*) &node, size of (char *), ticktock_channel, NULL);`
`  if (i > 0) {`
`  printf("Received %s\n", node);`
`  teja_memory_pool_put_node (tick_memory_pool, node);`
`  } else if (i == 0) {`
`  printf("BLAST OFF!!!\n");`
`  break;`
`  }`
`  }`
`}`
	The `tocker` function loops forever, and in each iteration waits forever for a message to come in over the `ticktock_channel`. The `teja_wait` function is instructed to poll every tenth of a second (1E8 nanoseconds). `TEJA_INFINITE_WAIT` is defined in the `teja_late_binding.h` file.

`int`
`init(void)`
`{`
`  printf("init\n");`
`  return 0;`
`}`
	This simple example needs no initialization. The init function is provided as an example to show how an initialization function can be mapped to a process.

Configuration Code

Unlike the application code, the configuration code is target specific. The configuration code is written to a file called `config.c` and contains the hardware architecture, software architecture, and the mapping to the application code. TABLE 3-2 lists the `config.c` file and provides comment.

TABLE 3-2 `config.c` File and Comments

`#include <stdio.h>`
`#include "teja_hardware_architecture.h"`
`#include "teja_software_architecture.h"`
`#include "teja_mapping.h"`
`#include "csp/sun/teja_cmt.h"`
`extern teja_architecture_t`
`create_cmt1board_architecture(`
`  teja_architecture_t container, const char *name);`
	Teja configuration APIs are declared. This example targets generic PCs and so includes `teja_cmt.h` from the Sun CMT chip support package. The package has a function to create the CMT1 board architecture. That function is declared as external

`int`
`hwarch(void)`
`{`
`  teja_architecture_t top;`
`  teja_architecture_t pc;`
`  teja_architecture_t cmt1_chip;`
`  top = teja_architecture_create(`
`  NULL, "top",`
`  TEJA_ARCHITECTURE_TYPE_USER_DEFINED);`
`  pc = create_cmt1board_architecture (top, "pc");`
`  cmt1_chip = teja_lookup_architecture (pc, "cmt1_chip");`
`  teja_architecture_set_property (cmt1_chip, "bsp_dir", BSP_DIR);`
`  return 0;`
`}`
	A user-defined hardware architecture called `top` is created as a container for the PC architecture

TABLE 3-2 `config.c` File and Comments *(Continued)*

```
int
swarch(void)
{
  teja_os_t os;
  teja_process_t process;
  teja_thread_t tick, tock;
  teja_channel_t channel;
  teja_memory_pool_t tick_memory_pool;
  const char* processors[3] = {"top.pc.cmt1_chip.strand0",
       "top.pc.cmt1_chip.strand1",
       NULL};
  const char* srcsets[2] = {"ticktock_srcs", NULL};
  teja_thread_t producers[2], consumers[2];
  os = teja_os_create(processors, "os", TEJA_OS_TYPE_RAW);
  process = teja_process_create(os, "ticktock", srcsets);
  tick = teja_thread_create(process, "tick_thread");
  tock = teja_thread_create(process, "tock_thread");
 teja_thread_set_property(tick, TEJA_PROPERTY_THREAD_ASSIGN_TO_PROCESSOR,
                          "top.pc.cmt1_chip.strand0");
 teja_thread_set_property(tock, TEJA_PROPERTY_THREAD_ASSIGN_TO_PROCESSOR,
                          "top.pc.cmt1_chip.strand1");
  producers[0] = tick; producers[1] = NULL;
  consumers[0] = tock; consumers[1] = NULL;
  channel = teja_channel_declare
    ("ticktock_channel",
     TEJA_GENERIC_CHANNEL_SHARED_MEMORY_OS_BASED,
     producers,
     consumers);
  tick_memory_pool = teja_memory_pool_declare
    ("tick_memory_pool",
     TEJA_GENERIC_MEMORY_POOL_SHARED_MEMORY_OS_BASED,
     100,
     32,
     producers,
     consumers,
     "top.pc.dram_mem");
  return 0;
}
```

The software architecture consists of the raw OS running on the CMT with the `ticktock` process running on that. The `tick` and `tock` threads are mapped respectively to `strand0` and `strand1` of the CMT architecture. The `ticktock_channel` and the `tick_memory_pool` have `tick` as the producer and `tock` as the consumer.

```
int
map(void)
{
  teja_map_function_to_thread("ticker", "tick_thread");
  teja_map_function_to_thread("tocker", "tock_thread");
  teja_map_initialization_function_to_process(
    "init", "ticktock");
  return 0;
}
```

The `ticker` function is mapped to the `tick` thread and the `tocker` function is mapped to `tock_thread`. The application code has no variables to be mapped. The `init` function is mapped to the target process.

Build Process

▼ To Create the Binary Image

1. **Create the shared library** `config.so` **by compiling the** `config.c` **file and the Teja-supplied** `cmt1_board.c` **chip support file.**

2. **Compile the** `ticktock.c` **file using** `tejacc` **to generate the application code in the code directory.**

   The following `makefile` shows how this is done.

```
TEJA_INSTALL_DIR=/opt/SUNWndps/tools
BSP_DIR=/opt/SUNWndps/bsp/Niagara1

all: config.so ticktock

%.o:%.c
cc -g -c -xcode=pic13 -xarch=v9
             -DTEJA_RAW_CMT -DBSP_DIR='$(BSP_DIR)'
             -I$(TEJA_INSTALL_DIR)/include $< -o $@

config.so: config.o cmt1_board.o
ld -G -o config.so config.o cmt1_board.o
             $(TEJA_INSTALL_DIR)/bin/libtejahwarchapi.so
             $(TEJA_INSTALL_DIR)/bin/libtejaswarchapi.so
             $(TEJA_INSTALL_DIR)/bin/libtejamapapi.so

cmt1_board.o: $(TEJA_INSTALL_DIR)/src/csp/sun/sparc64/cmt1_board.c
cc -g -c -xcode=pic13 -xarch=v9
             -DTEJA_RAW_CMT -DBSP_DIR='$(BSP_DIR)'
             -I$(TEJA_INSTALL_DIR)/include $< -o $@

ticktock: ticktock.c
$(TEJA_INSTALL_DIR)/bin/tejacc.sh
  -Dprintf=teja_synchronized_printf
           -I$(BSP_DIR)/include
  -hwarch config.so,hwarch
  -swarch config.so,swarch
  -map config.so,map
  -srcset ticktock_srcs ticktock.c

clean:
rm -rf config.so *.o code
```

3. **Run the `gmake` command in the `code/`*process_name*`/` generated source directory to create the application binary image.**

Execution

▼ To Execute the Binary Image

- **Copy the binary image to the `tftpboot` directory of the `tftp` server.**

  The CMT machine is reset, and the sytem is booted. See "Building and Booting Reference Applications" on page 7. When the application starts, the following countdown is printed to the console.

```
init
tick started.
tock started.
9...
8...
7...
6...
5...
4...
3...
2...
1...
SHUTDOWN.  Exiting tick thread ...
BLAST OFF!!!
SHUTDOWN.  Exiting tock thread ...
```

CHAPTER 4

CMT Debugger

This chapter summarizes using the debugger. Topics include:

CMT Debugger Overview

The CMT Debugger runs on the target and enables users to:

- Set, clear, and display breakpoints
- Set and display memory
- Display registers
- Display stack trace
- Manage thread focus
- Step to the next assembly instruction

The debugger is not symbolic. Symbol resolution is performed separately using a host-based tool called `dbg_helper.pl`. See "Resolving Symbols" on page 48.

All the debugger commands use `gdb` style syntax.

Debugging Configuration Code

As seen in "tejacc Compiler Configuration" on page 12, `tejacc` gets information about hardware architecture, software architecture, and mapping by executing the configuration code compiled into dynamic libraries.

The code is written in C and might contain errors causing `tejacc` to crash. Upon crashing, you are presented with a Java™ Hotspot exception, as `tejacc` is internally implemented in Java. The information reported in the exception requires knowledgeable interpretation.

An alternative version of `tejacc.sh` called `tejacc_dbg.sh` is provided to assist debugging configuration code. This program runs `tejacc` inside the default host debugger (`dbx` for Solaris hosts), stopping the execution immediately after the configuration libraries have been loaded. You then can continue execution to reach the instruction that causes the problem and verify its location. Alternatively, you can set breakpoints on the configuration functions, step through code, or use any other functionality provided by the host debugger.

To use `tejacc_dbg.sh`, replace the invocation of `tejacc.sh` in the `makefile` with `tejacc_dbg.sh`.

Entering the Debugger

The application program calls the debugger when any of the following conditions occur:

- At start time – If the application was compiled without the `-O` option, the application calls the debugger at start time. Applications compiled with the `-O` option start normally.
- At a breakpoint – If the application was compiled without the `-O` option and while running encounters a breakpoint, the application calls the debugger. Applications compiled with the `-O` option cannot set breakpoints.
- In a crash – If the application crashes, it calls the debugger. The debugger is called regardless of whether the application was compiled with or without the `-O` option.
- Typing Ctrl-C – If the application calls the `teja_debugger_check_ctrl_c()` function and you type the Ctrl-C key sequence, the debugger is also called. The debugger is called regardless of whether the application was compiled with or without the `-O` option.

Note – A call to the debugger stops all threads.

Note – The `teja_check_ctrl_c()` function must be executed periodically by at least one of the threads in order for Ctrl-C functionality to work. If the thread calling the `teja_check_ctrl_c()` function crashes or goes into a deadlock, the Ctrl-C key sequence stops working.

Debugger Commands

Displaying Help

`help` *command* or
`h` *command*

Description

Displays help for a *command*. If the *command* variable is absent, a general help page is displayed.

Example

```
dbg>help
        break <address> - set breakpoint
                    not available for all instructions (see docs)
        b <address>     - set breakpoint
                    not available for all instructions (see docs)
        bt n            - display stack trace
        delete breakpoint <bpid>       - clear breakpoint
        d breakpoint <bpid>    - clear breakpoint
        info            - display info help
        i               - display info help
        help [cmd]      - display help
        h [cmd]         - display help
        ? [cmd]         - display help
        cont            - resume execution
        c               - resume execution
        step            - step to next Assembly instruction
                    not available for all instructions (see docs)
        s               - step to next Assembly instruction
                    not available for all instructions (see docs)
        x/nfu <address> - display memory:
                          n (count)
                          u = {b|h|w|g} (unit)
                          f = {x|d|u|o|t|a|f|s|i} (format)
        thread <thdid> - switch thread focus
        w/u addr value  - set memory
                          u = {b|h|w|g} (unit)
```

Managing Breakpoints

Setting breakpoints is only supported in non-optimized mode and means the application must be built without the `-O` option to `tejacc`.

`break` *address* Command or `b` *address* Command

Description

Sets a breakpoint, where *address* is the hexadecimal address at which to break. The breakpoint is set only in regions of code that are characterized by sequential execution and not affected by control flow changes. The easiest way to set a proper breakpoint is to use the `dbg_helper` script. See "Resolving Symbols" on page 48.

Example

```
dbg>break 50b188
Breakpoint set at 0x50b188
```

`info break` Command or `i break` Command

Description

Displays a list of active breakpoints.

Example

```
dbg>info break
breakpoint [1] set at 0x50b188
```

In this example, only one breakpoint exists. It has an ID of `1`. When more than one breakpoint is set, each breakpoint receives a consecutive ID.

`delete breakpoint` *ID* Command or `d breakpoint` *ID* Command

Description

Deletes a breakpoint, where *ID* is the ID of the breakpoint.

Example:

```
dbg>delete breakpoint [1]
```

Managing Program Execution

`cont` Command or `c` Command

Description

Continues execution of the application.

Example

```
dbg>cont
```

`step` Command or `s` Command

Description

Steps to the next assembly instruction within the application.

Example

```
dbg>step
```

Note – The `step` command can only be used in regions of code that are characterized by sequential execution and not affected by control flow changes.

Displaying and Setting Memory

`x`/*nfu address* Command

Description

Displays memory contents where:

- *n* – Number of memory units to display.
- *f* – The display format. The only supported value is `x`, for hexadecimal format.
- *u* – The size of the unit. Supported values are:
  - `b` – byte
  - `h` – 2-byte half-word
  - `w` – 4-byte word
  - `g` – 8-byte long word
- *address* – The starting address in hexadecimal.

Example:

```
dbg>x/8xw 10000000
count = 8; format = HEX; unitsize = 4
[10000000] : 00000100 000000cd 00000001 00000114 00000100 000000ce
00000001 00518a44
```

w/*u* *address* *value* Command

Description

Sets memory where:

- *u* – The size of the unit. Supported values are:
  - b – byte
  - h – 2-byte half-word
  - w – 4-byte word
  - g – 8-byte long word
- *address* – The starting address in hexadecimal.
- *value* – The value to write in hexadecimal.

Example

```
dbg>w/w 10000000 00518a44
```

Managing Threads

info threads Command or i threads Command

Description

Displays a list of the active threads. The thread that has the focus is shown with an F symbol. Similarly, if a thread has crashed, it is shown with an F symbol.

Example

```
dbg>info threads
  : generatorthread: Teja thread id 0, strand id 0
F : classifierthread: Teja thread id 1, strand id 1
```

`thread` *ID* Command

Description

Changes the thread focus to the thread with the Teja thread ID of *ID*.

Example

```
dbg>thread 0
Thread focus changed to 0
```

In the previous example, the focus (`F`) was on `classifierthread`, with Teja ID of `1`. In this example, the focus has been moved to `generatorthread`.

Displaying Registers

`info reg` Command or `i reg` Command

Description

Displays the register contents for the thread in focus.

Example

```
dbg>info reg
Registers of strand 0:
G registers:
g[0] :  0000000000000000 0000000000000000 0000000000500000 0000000000000000
g[4] :  0000000000000000 0000000000615fa0 0000000000000000 0000000000000000
I registers:
i[0] :  000000000000006e ffffffffef1fe8d4 0000000000520c30 0000000010e01bc8
i[4] :  0000000000000000 0000000000000000 0000000010e00d91 000000000051458c
O registers:
o[0] :  000000000000006e 0000000000520c30 0000000010e01bc8 0000000000000000
o[4] :  0000000000600000 0000000000000061 0000000010e00cd1 0000000000514a18
L registers:
l[0] :  000000000000006e 0000000010e0172c 000000000051e8f0 ffffffffef1fe8d4
l[4] :  0000000000520c30 0000000000000000 0000000000000000 0000000000000000
gl      :  0000000000000001
tl      :  0000000000000001
tt      :  000000000000007c
tpc     :  0000000000508c88
tnpc    :  0000000000508c8c
tstate  : 0000009914001600
pstate  : 0000000000000014
tick    :  000001884f873558
tba     :  0000000000500000
asi     :  0000000000000014
```

Displaying Stack Trace

`bt` *frame_count* Command

Description

Displays the stack trace for the thread in focus for *frame_count* number of frames.

Example

```
dbg>bt 4
frame 1, sp 0x10e03580, call instruction at 0x50e888:
l[0] :  0000000000000001 00000000111606a8 0000000011160600 0000000000000000
l[4] :  00000000006170d8 0000000000001000 0000000000010000 0000000000000150
i[0] :  0000000000000800 0000000010e036f0 0000000010e036e8 0000000010e036e4
i[4] :  0000000000002000 0000000019ae8ec8 0000000010e02e31 000000000050e888
frame 2, sp 0x10e03630, call instruction at 0x50fcc4:
l[0] :  0000000000000001 00000000111606a8 0000000011160600 0000000000000000
l[4] :  00000000006170d8 0000000000001000 0000000000010000 0000000000000150
i[0] :  0000000000000800 0000000000000001 0000000019d8c148 0000000000000800
i[4] :  0000000019d8c140 0000000019d8c000 0000000010e02f01 000000000050fcc4
frame 3, sp 0x10e03700, call instruction at 0x50fbd8:
l[0] :  0000000000000001 00000000111606a8 0000000011160600 0000000000000000
l[4] :  00000000006170d8 0000000000001000 0000000000010000 0000000000000150
i[0] :  00000000111000e0 0000000000000015 0000000000010000 0000000011160580
i[4] :  0000000000002000 0000000019ae8ec8 0000000010e02fd1 000000000050fbd8
frame 4, sp 0x10e037d0, call instruction at 0x50e104:
l[0] :  fffffffffffffd8 fffffffffffffba 0000000000000003 0000000000000000
l[4] :  00000000006170d8 0000000000617000 0000000000000617 0000000000000400
i[0] :  00000000111000e0 000000000000792d 000000000000792d 0000000011100180
i[4] :  000000000000792d 0000000000000000 0000000010e03081 000000000050e104
```

Resolving Symbols

The `dbg_helper.pl` script is used to resolve symbols to set breakpoints in the correct places. The script is located in the Teja installation, under the `/bin` directory.

`-h` Option

Description

Displays help information.

`-f` *function_name* Option

Description

Prints a debugger command to set a breakpoint at the given *function_name*. This option does not work for static functions. To set a breakpoint inside of a static function, use the `-l` *file_name*`:`*line_number* option.

Example

```
$ dbghelper.pl -f classifier ./main
   b 50b17c
```

`-g` *global_variable* Option

Description

Prints a debugger command to display the contents of the given *global_variable*. The size of the memory displayed is fixed and does not consider the actual size of the *global_variable*. You might need to increase the size of the memory.

Example

```
$ dbghelper.pl -g stats ./main
   x/1wx 13000640
```

`-l` *file_name*`:`*line_number* Option

Description

Prints a debugger command to set a breakpoint at the provided *file_name*:*line_number*. The *file_name* and *line_number* refer to your source code.

Example

```
$ dbghelper.pl -l src/classifier.c:57 ./main
   b 50b188
```

CHAPTER 5

Teja Profiler

This chapter discusses the Teja Profiler used in the Netra Data Plane software. Topics include:

Teja Profiler Introduction

Teja Profiler is a set of API calls that help you collect various critical data during the execution of an application. You can profile one or more areas of your application such as CPU utilization, I/O wait times, and so on. Information gathered using the profiler helps you decide where to direct performance-tuning efforts. The profiler uses special counters and resources available in the system hardware to collect critical information about the application.

As with instrumentation-based profiling, there is a slight overhead for collecting data during the application run. Teja Profiler uses as little overhead as possible so that the presented data is very close to the actual application run without the profiler API in place.

How the Profiler Works

You enable the Teja Profiler with the `-pg` command-line option. You can insert the API calls at desired places to start collecting profiling data. Teja Profiler configures and sets the hardware resources to capture the requested data. At the same time, Teja Profiler reserves and sets up the memory buffer where the data will be stored. You can insert calls to update the profiler data at any further location in the application. With this setup, the profiler reads the current values of the data and stores the values in memory.

There is an option to store additional user data in the memory along with each update capture. Storing this data helps you analyze the application in the context of different application-specific data.

You can also obtain the current profiler data in the application and use the data as desired. With the assistance of other communication mechanisms you can send the data to the host or other parts of the application.

By demarking the portions that are being profiled, you can dump the collected data to the console. The data is presented as a comma-delimited table that can be further processed for report generation.

To minimize the amount of memory space needed for the profile capture, the profiler uses a circular buffer mechanism to store the data. In a circular buffer, the start and the end data is preserved, yet the intermediate data is overwritten when the buffer becomes full.

Groups and Events

The profiling data is captured into different groups based on the significance of the data. For example, with the CPU performance group, events such as completed instruction cycles, data cache misses, and secondary cache misses are captured. In the memory performance group, events such as memory queue and memory cycles are captured. Refer to the Profiler section of the *Netra Data Plane Software Suite Reference Manual* for the different groups and different events that are captured and measured on the target.

Dump Output

The profiler dump output consists of one line per profiler record. Each line most commonly has a format of nine comma-delimited fields. The fields contain values in hexadecimal. If a record is prefixed with a `-1`, that indicates that the buffer allocated for the profiler records has overrun. When a buffer overrun occurs, you should increase the value of the `profiler_buffer_size` property as described in the configuration section of the *Netra Data Plane Software Suite 1.1 Reference Manual*, and run the application again.

TABLE 5-1 describes the fields of the profiler records:

TABLE 5-1 Profiler Record Fields

Field	Description
CPU ID	The number representing the CPU ID where the current profiler call was made.
Caller ID	The number representing the source location of the `teja_profiler` call. The `records/profiler_call_locations.txt` file lists all of the IDs and their corresponding source locations.
Call Type	The type of `teja_profiler` call. The values listed are defined in the `teja_profiler.h` file.
Completed Cycles	The running total of completed clock cycles so far. You can use this value to calculate the time between two entries.
Program Counter	The value of the program counter when the current profiler call was invoked.
Group Type	The group number of the events started or being measured.

TABLE 5-1 Profiler Record Fields *(Continued)*

Field	Description
Event Values	The value of the events. This value can be one or more columns depending on the target CSP. The target-dependent values are described in the profiler section in the *Netra Data Plane Software Suite 1.1 Reference Manual*. The order of the events are the same as the location of the bit set in the event bit mask, passed to `teja_profiler_start`, starting from left to right. For the entry that represents `teja_profiler_start`, the values represent the event types. There are two events per record (group) in the dump output: • `event_hi` – represents the higher bit set in the event mask • `event_lo` – represents the lower bit set in the event mask Overflow values consist of: • `0x0` – no overflow • `0x1` – overflow of the event_lo • `0x2` – overflow of the event_hi • `0x3` – overflow of both `event_hi` and `event_lo`
Overflow	The overflow information of one or more events being measured. The value is target-dependent and is explained in the *Netra Data Plane Software Suite 1.1 Reference Manual*.
User Data	The values of the user-defined data. Zero or more columns, depending on the number of counters allocated and recorded by the user.

Refer to "Dump Output Example" on page 56 for an example of dump output.

Profiler Examples

Profiler API

CODE EXAMPLE 5-1 provides an example of profiler API output.

CODE EXAMPLE 5-1 Sample Profiler API Output

```
main()
{
  /* ...user code... */
 teja_profiler_start(TEJA_PROFILER_CMT_CPU, TEJA_PROFILER_CMT_CPU_IC_MISS);
  /*   ...user code... */
  while (packet) {
    /*  ...user code... */
    teja_profiler_update(TEJA_PROFILER_CMT_CPU, num_pkt);
    if (num_pkt == 100)
      teja_profiler_dump(generator_thread);
teja_profiler_stop(TEJA_PROFILER_CMT_CPU);
  }
}
```

Profiler Configuration

You can change the profiler configuration in the software architecture. The following example shows the three profiler properties that can be changed per process.

```
teja_process_set_property(main_process, "profiler_log_table_size","4096");
```

`main_process` is the process object that was created using the `teja_process_create` call. The property values are applied to all threads mapped to the process specified using `main_process`.

Dump Output Example

The following is an example of dump output.

```
TEJA_PROFILE_DUMP_START,ver1.1
CPUID,ID,Type,Cycles,PC,Grp,Evt_Hi,Evt_Lo,Overflow,User Data
0,2be4,1,29371aa3d0,51171c,1,100,4
0,2bf6,1,294bbbd464,51189c,2,2,1
0,2c0c,1,29629416a0,511a08,4,2,1
0,2c22,1,29761be17c,511b7c,8,2,1
0,2c38,1,2988fbbf60,511ce8,10,2,1
0,2c4e,1,299c3ca170,511e5c,20,2,1
0,30e6,2,2d20448f60,512904,1,36c2ba96,ce,0,0,114ee88
0,30fe,2,2d37b98aec,512acc,2,9,9,0,0
TEJA_PROFILE_DUMP_END
```

The string, `ver1.1`, is the dump format version. The string is used as an identifier of the output format. The string helps scripts written to process the output validate the format before processing further.

In the first record, call type `1` represents `teja_profiler_start`. The values `100` and `4` seen in the `event_hi` and `event_lo` columns are the types of events in group `1` being measured. In the record with ID `30e6`, call type `2` represents `teja_profiler_update`, so the values `36c2ba96` and `ce` are the values of the event types `100` and `1` respectively.

Cycle counts are in increasing order so the difference between two of them provides the exact number of cycle counts between two profiler API calls. The difference divided by the processor frequency calculates the actual time between two calls.

IDs `2be4` and `2bf6` represent the source location of the profiler API call. The `records/profiler_call_locations.txt` file lists a table that maps IDs and actual source locations.

CHAPTER 6

Interprocess Communication Software

This chapter describes the interprocess communication (IPC) software. Topics include:

IPC Introduction

The interprocess communication (IPC) mechanism provides a means to communicate between processes that run in a domain under the NDPS Lightweight Runtime Environment (LWRTE) and processes in a domain with a control plane operating system. This chapter describes the generic APIs that are available to use the IPC mechanism, as well as the specific APIs and configuration interfaces used to operate in an LDoms environment with LWRTE and Solaris software.

Programming Interfaces

The API described in this section is available on all operating environments that support IPC communications with LWRTE domain. The tnipc.h header located in the src/common/include directory of the SUNWndps package defines the interface and must be included in source files using the API. The header file defines a number of IPC protocol types. User-defined protocols must not be in conflict with these predefined types.

ipc_connect

Description

This function registers a consumer with an IPC channel. The opaque handle that is returned by a successful call to this function must be passed to access the channel using any of the other interface functions.

```
ipc_handle_t
ipc_connect(uint16_t channel, uint16_t ipc_proto)
```

Parameters:

channel:	ID of channel
ipc_proto:	Protocol type of IPC messages that are expected

Return values:

NULL in case of failure
IPC handle otherwise. This handle needs to be passed to the tx/rx/free functions.

ipc_register_callbacks

Description

This function registers callback functions for the consumer of an IPC channel. When a message is received by the IPC framework, it strips the IPC header from the message and calls the rx_hdlr function with the content of the message. In future releases, the evt handler may be used to convey link events to the consumer.

```
int
ipc_handle_t ipc_hdl,
```

```
event_handler_ft evt_hdler,
rx_handler_ft rx_hdler,
caddr_t arg)
```

Parameters:

`ipc_hdl`:	Handle for IPC channel, obtained from `ipc_register_callbacks`().
`evt_hdlr`:	Function to handle link events.
`rx_hdlr`:	Function to handle received messages.
`arg`:	Opaque argument that the framework will pass back to the handler functions.

Return values:

`IPC_SUCCESS`	
`EFAULT`	invalid handle

ipc_tx

Description

This function transmits messages over IPC. The message is described by the `mblk` passed to the function. To make the function as efficient as possible, the function makes some nonstandard assumptions about the messages:

- There are 8 bytes of headroom in the data buffer before the messages.
- Messages are contained in a single data buffer.

As the memory containing the message is not freed inside the function, the caller must deal with memory management accordingly.

```
int
ipc_tx(mblk_t *mp, ipc_handle_t ipc_hdl)
```

Parameters:

`mp`:	Pointer to message block describing the messages.
`ipc_hdl`:	Handle for IPC channel, obtained from `ipc_register_callbacks`().

Return values

`IPC_SUCCESS`	
`EIO`	The write to the underlying media failed.

ipc_rx

Description

At this time, the only way to receive messages is through the callback function. In LWRTE, the callback function is called when the polling context finds a message on the channel. In Solaris user space, the callback is hidden in the framework, it makes the message available to be read by the read() system call.

ipc_free

Description

The IPC framework allocates memory for messages that are received using its available memory pools. The consumer of an IPC message must call this function to return the memory to that pool.

```
void
ipc_free(mblk_t *mp, ipc_handle_t ipc_hdl)
```

Parameters

mp	Pointer to message block describing message to be freed.
ipc_hdl	Handle for IPC channel, obtained from ipc_register_callbacks().

Using IPC in the LWRTE Domain

In the LWRTE domain, the interfaces described in the "Programming Interfaces" on page 58 are used to communicate with other domains using IPC. Before this infrastructure can be utilized, you must initialize it. Once it is initialized, because there are no interrupts, you must ensure that every channel is polled periodically. This section describes the API for these tasks.

To use this function, the lwrte_ipc_if.h header file, which is located in the lib/ipc/include directory of the SUNWndps package, must be included where needed.

tnipc_init

Description

This function must be called in the initialization routine. This function must be called after the LDoms framework has been initialized, that is, `mach_descrip_init`(), `lwrte_cnex_init`(), and `lwrte_init_ldc`() must be called first.

```
int
tnipc_init()
```

Return values

`0`	Success
`EFAULT`	Too many channels in machine description
`ENOENT`	Global configuration channel not defined

tnipc_poll

Description

To receive messages or event notifications for any IPC channel, this function must be called periodically. For example, it may be called as part of the main loop in the statistics thread. When a message is received, this ensures that the callback function registered for the channel and IPC type is called.

```
int
tnipc_poll()
```

Return values

This function always returns 0.

Polling through the `tnipc_poll`() API is adequate for most IPC channels carrying low bandwidth control traffic. For higher throughput channels, the polling can be moved to a separate strand, using the following API functions:

- `tnipc_register_local_poll`
- `tnipc_local_poll`
- `tnipc_unregister_local_poll`

tnipc_register_local_poll

Description

This function removes the channel identified by the handle passed to the function from the pool of channels polled by the `tnipc_poll`() function. This function returns an opaque handle that must be passed to the `tnipc_local_poll`() function.

```
ipc_poll_handle_t
tnipc_register_local_poll(ipc_handle_t ipc_hdl)
```

Parameter

`ipc_hdl` The channel handle obtained from the `ipc_connect`() API call.

Return values

NULL invalid input
Opaque handle to be passed to the `tnipc_local_poll`() call.

tnipc_local_poll

Description

This function works the same way as `tnipc_poll`(), except that only the channel identified by the handle is polled. If there is data on the channel, the `rx` callback will be called.

```
int
tnipc_local_poll(ipc_poll_handle_t poll_hdl)
```

Parameter

`poll_hdl` The handle obtained from the `tnipc_register_local_poll`() API call

Return values

This function always returns 0.

tnipc_unregister_local_poll

Description

This function reverses the effect of the tnipc_register_local_poll() call and places the channel identified by the handle back into the common pool polled by tnipc_poll().

```
int
tnipc_unregister_local_poll(ipc_poll_handle_t poll_hdl)
```

Parameter

poll_hdl	The handle obtained from the tnipc_register_local_poll() API call

Return values

This function always returns 0.

Using IPC in the Solaris Domain

In Solaris software, there are two different ways to use the IPC API, from user space and from kernel space.

User Space

To use an IPC channel from the Solaris user space, the character-driver interfaces are used. A program opens the tnsm device, issues an ioctl() call to connect the device to a particular channel, and then uses read() and write() calls to send and receive messages.

Before any of the interfaces can be used, the tnsm driver must be installed and loaded. This is done using the pkgadd system administration command to install the SUNWndpsd package on the Solaris domains that use IPC for communication.

The open(), close(), read(), and write() interfaces are described in their respective man pages.

The open() call on the tnsm driver will create a new instance for the specific client program. Before you can use the read() and write() calls, you must call TNIPC_IOC_CH_CONNECT ioctl. This ioctl takes the channel ID and IPC type to be used for messages by this instance.

Kernel

In the kernel, the interfaces described in "Programming Interfaces" on page 58 are used.

Configuring the Environment for IPC

This section describes the configuration of the environment needed to use the IPC framework. This section covers setup of memory pools for the `LWRTE` application, the LDoms environment, and the IPC channels.

Memory Management

The IPC framework shares its memory pools with the basic LDoms framework. These pools are accessed through `malloc`() and `free`() functions that are implemented in the application. The `ipfwd_ldom` reference application contains an example implementation.

The file `ldc_malloc_config.h` contains definitions of the memory pools and their sizes. `ldc_malloc.c` contains the implementation of the `malloc`() and `free`() routines. These functions have the expected signatures:

- `void *malloc(size_t size)`
- `void free(void *addr)`

In addition to these implementation files, the memory pools must be declared to the NDPS runtime. This is done in the software architecture definition in `ipfwd_swarch.c`.

IPC in the LDoms Environment

In the LDoms environment, the IPC channels use Logical Domain Channels (LDCs) as their transport media. These channels are set up as Virtual Data Plane Channels using the `ldm` command (see the LDoms documentation). These channels are set up between a server and a client, and some basic configuration channels must be defined adhering to the naming convention described in "LDoms Channel Setup" on page 65. Each channel has a server defined in the `LWRTE` domain and a client defined in the link partner domain.

LDoms Channel Setup

There must be a domain that has the right to set up IPC channels in the `LWRTE` domain. This domain can be the primary domain or a guest domain with the client for the configuration service. The administrator must only setup this channel, when the service (`LWRTE`) and the client domain are up (and the `tnsm` driver attached at the client), the special IPC channel with ID 0 is established automatically between the devices. The `tnsmctl` utility can then be used in the configuring domain to set up additional IPC channels (provided that the required virtual data plane channels have been configured.)

- In the `LWRTE` domain, a data plane channel service with the name `primary-gc` must be established using the command
  `ldm add-vdpcs primary-gc` *lwrte-domain-name*.
- In the configuration domain, the respective client with the name `tnsm-gc0` must be established using the command
  `ldm add-vdpcc tnsm-gc0 primary-gc` *config-domain-name*.

To enable IPC communications between the `LWRTE` domain and additional domains, a special configuration channel must be set up between these domains. Again, the channel names must adhere to a naming convention. In the `LWRTE` domain, the service name must begin with the prefix `config-tnsm`, whereas the client name in the other domain must be named `config-tnsm0`. For example, such a channel could be established using the `ldm` commands.

- `ldm add-vdpcs config-tnsm-`*clnt-domain-name lwrte-domain-name*
  in the `LWRTE` domain
- `ldm add-vdpcc config-tnsm0 config-tnsm-`*clnt-domain-name clnt-domain-name*
  in the client domain

Additional channels can be added for data traffic between these domains, there are no naming conventions to follow for these channels, they are configured using the `ldm` commands.

- `ldm add-vdpcs` *service-name lwrte-domain-name*
  in the `LWRTE` domain
- `ldm add-vdpcc` *client-name service-name client-domain-name*
  in the client domain.

IPC Channel Setup

Once the data plane channels are set up by the administrator in the primary domain, the `tnsmctl` utility is used to set up IPC channels from the IPC control domain. *tnsmctl* uses the following syntax:

```
tnsmctl -S -C channel-id -L local-ldc -R remote-ldc -F control-channel-id
```

The parameters to `tnsmctl` are described in TABLE 6-1. All of these parameters need to be present to set up an IPC channel.

TABLE 6-1 `tnsmctl` Parameters

Parameter	Description
-S	Setup IPC channel
-C *channel-id*	Channel ID of the new channel to be set up
-L *local-ldc*	Local LDC ID of the virtual data plane channel to be used for this IPC channel. Local here always means local to the `LWRTE` domain. Obtain this LDC ID using the ldm list-bindings command.
-R *remote-ldc*	Remote LDC ID of the virtual data plane channel to be used for this IPC channel, that is, the LDC ID seen in the client domain. Obtain this LDC ID using the ldm list-bindings command.
-F *control-channel-id*	IPC channel ID of the control channel between the `LWRTE` and the client domain. If the client domain is the control domain, this channel ID is 0. For all other client domains, the control channel must be set up by the administrator. To set up the control channel, use the same ID for both the `-C` and the `-F` options.

Example Environment

The following is a sample environment, complete with all commands needed to set it up in a Sun Fire™ T2000 server.

Domains

TABLE 6-1 describes the four environment domains:

TABLE 6-2 Environment Domains

Domain	Description
`primary`	This domain owns one of the PCI buses and uses the physical disks and networking interfaces to provide virtual I/O to the Solaris guest domains.
`ldg1`	This domain owns the other PCI bus (`bus_b`) with its two network interfaces and runs an LWRTE application.
`ldg2`	This domain runs control plane applications and uses IPC channels to communicate with the LWRTE domain (`ldg1`).
`ldg3`	This domain controls the `LWRTE` domain through the global control channel. The `tnsmctl` utility is used here to set up IPC channels.

The primary as well as the guest domains `ldg2` and `ldg3` run the Solaris 10 Update 3 Operating System (or higher) with the patch level required for LDoms operation. The `SUNWldm` package is installed in the primary domain, the `SUNWndpsd` package is installed in both `ldg2` and `ldg3`.

Assuming 4 GB of memory for each of the domains, and starting with the factory default configuration, the environment can be set up using the following domain commands:

`primary`

```
ldm remove-mau 8 primary
ldm remove-vcpu 28 primary
ldm remove-mem 28G primary
```
(This assumes 32GB of total memory, adjust accordingly.)
```
ldm remove-io bus_b primary
ldm add-vsw mac-addr=you-mac-address net-dev=e1000g0 primary-vsw0
primary
```

```
ldm add-vds primary-vds0 primary
ldm add-vcc port-range=5000-5100 primary-vcc0 primary
ldm add-spconfig 4G4Csplit
```

ldg1 - LWRTE

```
ldm add-domain ldg1
ldm add-vcpu 16 ldg1
ldm add-mem 4G ldg1
ldm add-vnet mac-addr=your-mac-address-2 vnet0 primary-vsw0 ldg1
ldm add-var auto-boot\?=false ldg1
ldm add-io bus_b ldg1
```

ldg2 - Control Plane Application

```
ldm add-domain ldg2
ldm add-vcpu 4 ldg2
ldm add-mem 4G ldg2
ldm add-vnet mac-addr=your-mac-address-3 vnet0 primary-vsw0 ldg2
ldm add-vdsdev your-disk-file vol2@primary-vds0
ldm add-vdisk vdisk1 vol2@primary-vds0 ldg2
ldm add-var auto-boot\?=false ldg2
ldm add-var boot-device=/virtual-devices@100/channel-
devices@200/disk@0 ldg2
```

ldg3 - Solaris Control Domain

```
ldm add-domain ldg3
ldm add-vcpu 4 ldg3
ldm add-mem 4G ldg3
ldm add-vnet mac-addr=your-mac-address-4 vnet0 primary-vsw0 ldg3
ldm add-vdsdev your-disk-file-2 vol3@primary-vds0
ldm add-vdisk vdisk1 vol3@primary-vds0 ldg3
ldm add-var auto-boot\?=false ldg3
ldm add-var boot-device=/virtual-devices@100/channel-
devices@200/disk@0 ldg3
```

The disk files are created using the `mkfile` command. Solaris is installed once the domains are bound and started in a manner described in the LDoms Administrator's Guide.

Virtual Data Plane Channels

While the domains are unbound, the virtual data plane channels are configured in the primary domain as follows:

Global Control Channel

```
ldm add-vdpcs primary-gc ldg1
ldm add-vdpcc tnsm-gc0 primary-gc ldg3
```

Client Control Channel

```
ldm add-vdpcs config-tnsm-ldg2 ldg1
ldm add-vdpcc config-tnsm0 config-tnsm-ldg2 ldg2
```

Data Channel

```
ldm add-vdpcs ldg2-vdpcs0 ldg1
ldm add-vdpcc vdpcc0 ldg2-vdpcs0 ldg2
```

Additional data channels can be added with names picked by you. Once all channels are configured, the domains can be bound and started.

IPC Channels

The IPC channels are configured using the `/opt/SUNWndpsd/bin/tnsmctl` utility in `ldg3`.

Before you can use the utility, the `SUNWndpsd` package must be installed in both `ldg3` and `ldg2`, using the `pkgadd` system administration command. After installing the package, you must add the `tnsm` driver by using the `add_drv` system administration command.

To be able to configure these channels, the output of `ldm ls-bindings` in the primary domain is needed to determine the LDC IDs. As an example, the relevant parts of the output for the configuration channel between `ldg1` and `ldg2` might appear as follows:

For `ldg1`:

```
Vdpcs: config-tnsm-ldg2
               [LDom ldg2, name: config-tnsm0]
               [LDC: 0x6]
```

For ldg2:

```
Vdpcc: config-tnsm0 service:config-tnsm-ldg2 @ ldg1
               [LDC: 0x5]
```

The channel uses the local LDC ID 6 in the LWRTE domain (ldg1) and remote LDC ID 5 in the Solaris domain. Given this information, and choosing channel ID 3 for the control channel, this channel is set up using the following command-line:

```
tnsmctl -S -C 3 -L 6 -R 5 -F 3
```

After the control channel is set up, you can then set up the data channel between ldg1 and ldg2. Assuming local LDC ID 7, remote LDC ID 6, and IPC channel ID 4 (again, the LDC IDs must be determined using ldm ls-bindings), the following command-line sets up the channel:

```
tnsmctl -S -C 4 -L 7 -R 6 -F 3
```

Note that the -C 4 parameter is the ID for the new channel, -F 3 has the channel ID of the control channel set up previously. After the completion of this command, the IPC channel is ready to be used by an application connecting to channel 4 on both sides. An example application using this channel is contained in the SUNWndps package, and described in the following section.

Reference Application

The NDPS package contains an IP forwarding reference application that uses the IPC mechanism. The NDPS package contains an IP forwarding application in LWRTE and a Solaris utility that uses an IPC channel to upload the forwarding tables to the LWRTE domain (see "Forwarding Application" on page 72). NDPS chooses which table to use and where to gather some simple statistics, and displays them in the Solaris domain. The application is designed to operate in the example setup shown in "IPC Channels" on page 69.

Common Header

The common header file `fibtable.h`, located in the `src/common/include` subdirectory, contains the data structures shared between the Solaris and the `LWRTE` domains. In particular, it contains the message formats for communication protocol used between the domains, and the IPC protocol number (201) that it uses. This file also contains the format of the forwarding table entries.

Solaris Utility Code

The code for the Solaris utility is in the `src/solaris` subdirectory and is composed of the single file `fibctl.c`. This file implements a simple `CLI` to control the forwarding application running in the `LWRTE` domain. The application is built using `gmake` in the directory and deployed into a domain that has an IPC channel to the `LWRTE` domain established. The program opens the `tnsm` driver and offers the following commands:

`connect` *Channel_ID*

Connects to the channel with ID *Channel_ID*. The forwarding application is hard coded to use channel ID 4. The IPC type is hard coded on both sides. This command must be issued before any of the other commands.

`use-table` *Table_ID*

Instructs the forwarding application to use the specified table. In the current code, the table ID must be 0 or 1.

`write-table` *Table_ID*

Transmits the table with the indicated ID to the forwarding application. There are two predefined tables in the application.

`stats`

Requests statistics from the forwarding application and displays them.

`read`

Reads an IPC message that has been received from the forwarding application. Currently not used.

`status`

Issues the `TNIPC_IOC_CH_STATUS ioctl`.

`exit / x / quit /q`

Exits the program.

`help`

Contains program help information.

Forwarding Application

There are two components to the code implementing the forwarding application:

- The hardware and software architecture as well as the mapping. These files are located in the `src/config` subdirectory.
- The actual implementation of the packet handling and forwarding algorithm. The files for this implementation are located in the `src/app` subdirectory.

The hardware architecture is identical to the default architecture in all other reference applications.

The software application differs from other applications in that it contains code for the specific number of strands that the target LDoms will have. Also, the memory pools used in the `malloc`() and `free`() implementation for the LDoms and IPC frameworks are declared here.

The mapping file contains a mapping for each strand of the target LDom.

The `rx.c` and `tx.c` files contain simple functions that use the Ethernet driver to receive and transmit a packet, respectively.

`ldc_malloc.c` contains the implementation of the memory allocation algorithm. The corresponding header file, `ldc_malloc_config.h`, contains some configuration for the memory pools used.

user_common.c contains the memory allocation provided for the Ethernet driver, as well as the definition for the queues used to communicate between the strands. The corresponding header file, user_common.h contains function prototypes for the routines used in the application, as well as declarations for the common data structures.

ipfwd.c contains the definition of the functions that are run on the different strands. In this version of the application, all strands start the _main() function. Based on the thread IDs, the _main() function calls the respective functions for rx, tx, forwarding, a thread for IPC, the cli, and statistics gathering.

The main functionality is provided by the following processes:

- The rx_process strand polls one Ethernet interface and places received packets on a queue.
- The ipfwd_process polls the queue of its associated rx interface, calls the IP forwarding algorithm, and places the packet in the outbound queue indicated by the forwarding decision. This process services a single queue inbound, but puts outgoing packets into one of an array of queues.
- The tx_process polls an array of queues (one for each forwarding thread) and transmits any packet on the Ethernet interface.

The IP forwarding algorithm called by the forwarding thread is implemented in ipfwd_lib.c. The lookup algorithm used is a simple linear search through the forwarding table. The destination MAC address is set according to the forwarding entry found, and the TTL is decremented.

ipfwd_config.h contains configuration for the forwarding application, such as the number of strands and memory sizes used.

init.c contains the initialization code for the application. First, the queues are initialized. Initialization of the Ethernet interfaces is left to the rx strands, but the tx strands must wait until that initialization is done before they can proceed. The initialization of the LDoms framework is accomplished using calls to the functions mach_descrip_init(), lwrte_cnex_init(), and lwrte_init_ldc(). After this, the IPC framework is initialized by a call of tnipc_init(). The previous four functions must be called in this specific order. Finally, the data structures for the forwarding (that is, the tables) are initialized.

The forwarding application can be built using the build script located in the main application directory. For this application in an LDoms environment:

- The user_defs.mk file in the same directory must contain the location of the IPC library.
- The make file must contain the line "CLI_MOD = -DLDOMS" to enable LDoms support.

To deploy the application, the image must be copied to a `tftp` server. The image can then be booted using a network boot from either one of the Ethernet ports, or from a virtual network interface. See the `README` file for details. After booting the application, the IPC channels are initialized as described in "Example Environment" on page 67. Once the IPC channels are up, you can use the `fibctl` utility to manipulate the forwarding tables and gather statistics.

APPENDIX **A**

Frequently Asked Questions

This appendix provides frequently asked questions regarding the Netra Data Plane software and how it interacts with the `tejacc` compiler.

Summary

General Questions

What is Teja 4.x and How Does It Differ From an Ordinary C Compiler?

Teja 4.x is an optimizing C compiler (called `tejacc`) and API system for developing scalable, high-performance applications for embedded multiprocessor architectures. `tejacc` operates on a system-level view of the application through three techniques:

- `tejacc` obtains the characteristics of the targeted hardware and software system architecture by executing a user-supplied architecture specification.
- `tejacc` examines multiple sets of source files and their relationship to the target architecture in parallel.
- `tejacc` handles a special class of APIs used in the application code according to the system-level context. See "What Is Context-Sensitive Generation?" on page 92.

The techniques yield superior code validation and optimization, leading to more reliable and higher performance systems.

Where Are the Tutorials?

The ticktock tutorial is in "Tutorial" on page 33.

Where Is the Documentation?

The Teja specific documentation is available from the Teja website:

`http://www.teja.com/library/index.html`

Configuration Questions

What Are the Hardware Architecture, Software Architecture and Mapping Dynamic Libraries for?

These three dynamic libraries are user supplied. The libraries describe the configuration of the hardware (processors, memories, buses), software (OS, processes, threads, communication channels, memory pools, mutexes), and mapping (functions to threads, variables to memory banks). The library code runs in the context of the `tejacc` compiler. The `tejacc` compiler uses this information as a global system view on the entire system (hardware, user code, mapping, connectivity among components) for different purposes:

- Validation – For example, if a thread tries to reach a variable that is mapped to a memory bank that is not reachable by the processor on which the thread runs, the compiler flags this as an error.
- Optimization – See "What Is Context-Sensitive Generation?" on page 92.

The dynamic libraries are run on the host, not on the target.

How Can I Debug the Dynamic Libraries?

Two ways to help debug the dynamic libraries are:

- Add `printf()` calls to the hardware architecture, software architecture, and mapping code. For example:

```
printf("%s:%d\n",__FILE__,__LINE__)
```

- On targets that use `gcc` as the target compiler (not Sun CMT), follow this procedure:

1. **Type:**

```
gdb $teja-install-directory/bin/tejacc
```

2. **Set a breakpoint on the `teja_user_libraries_loaded` function.**
3. **Type `run` followed by the same parameters that were passed to `tejacc`.**

   Control returns immediately after the user dynamic libraries are loaded.
4. **Set a breakpoint on the desired dynamic library function, and type `cont`.**

What Should I Do When the `tejacc` Compiler Crashes?

There might be a bug in the hardware architecture, software architecture, or mapping dynamic libraries. To debug the issue, see "How Can I Debug the Dynamic Libraries?" on page 79.

What If the Hardware Architecture, Software Architecture, or Mapping Dynamic Libraries Crash?

`tejacc` gets information about hardware architecture, software architecture, and mapping by executing the configuration code compiled into dynamic libraries. The code is written in C and might contain errors causing `tejacc` to crash. Upon crashing, you are presented with a Java Hotspot exception, as `tejacc` is internally implemented in Java.

An alternative version of `tejacc.sh` called `tejacc_dbg.sh` is provided to assist debugging configuration code. This program runs `tejacc` inside the default host debugger (`gdb` for Linux/Cygwin hosts, `dbx` for Solaris hosts). The execution automatically stops immediately after the hardware architecture, software architecture, and mapping dynamic libraries have been loaded by `tejacc`.

You can continue the execution and the debugger stops at the instruction causing the crash. Alternatively, you can set breakpoints in the code before continuing or use any other feature provided by the host debugger.

Can I Build Hardware Architecture, Software Architecture, and Mapping in the Same Dynamic Library?

The dynamic libraries can be combined, but the entry points must be different.

Can I Map Multiple Variables With One Function Call?

Use regular expressions to map multiple variables to a memory bank, using the function:

```
teja_mapping_t teja_map_variables_to_process_(const char * var, const char *
process);
```

For example, to map all variables starting with `my_var_` to the OS-based memory bank:

```
teja_map_variables_to_memory ("my_var_.*", TEJA_MEMORY_TYPE_OS_BASED);
```

Building Questions

Where Is the Generated Code?

The generated code is located in the *top-level-application*`/code/`*process* directory, where *top-level-application* is the directory where `make` was invoked and *process* is the process name as defined in the software architecture.

If you are generating with optimization there is an additional directory, `code/`*process*`/.ir`. Optimized generation is a two-step process. The `.ir` directory contains the result of the first step.

Where Is the Executable Image?

The executable image is located in the `code/`*process* directory, where *process* is the process name as defined in the software architecture.

How Can I Compile Multiple Modules on the Same Command-Line?

`tejacc` is a global compiler, and all C files must be provided on the same command-line in order for it to perform global validation and optimization. To compile an application that requires multiple modules, use the `srcset` CLI option. The syntax for this option is:

```
-srcset srcset-name srcset-specific-options source-files
```

where:

- *srcset-name* – Name defined in the software architecture
- *srcset-specific-options* – Options (for example, `-D` or `-I`) that apply only to this source set.
- *source-files* – List of files that are contained in this source set.

How Can I Pass Different CLI Options to Different Modules on the `tejacc` Command-Line?

See "How Can I Compile Multiple Modules on the Same Command-Line?" on page 81.

How Can I Change the Behavior of the Generated `makefile` Without Modifying It?

You can use the `user_defs.mk` file in the top level application directory to overwrite any parameter that the generated `makefile` sets. The substituted value for the parameter is used for the compilation.

You can specify any file other than the default `user_defs.mk` file using the `"external_makefile"` property of the process. For example:

```
teja_process_set_property(<process_obj>, "external_makefile", "new-filename-with-or-without-path")
```

If the path is not specified, the top-level application directory is assumed.

Note – There is no warning or error if the file does not exist, the compilation continues with the generated Makefile parameters.

You can also specify the `user_defs.mk` file as a value to the `EXTERNAL_DEFINES` parameter during the compilation of the generated code. For example:

```
gmake EXTERNAL_DEFINES=../../user_defs.mk
```

The `user_defs.mk` file takes precedence over the value you specify in the software architecture if you use both methods.

How Do I Compile the RLP and IPfwd Applications to Run on the Netra CP3060?

For `ipfwd`, execute `build_1g_1060` under `src/apps/rlp`.

For RLP, you need to modify `apps/rlp/src/app/rlp.c` and `apps/rlp/src/app/rlp_config.h` to reflect the exact mapping.

Add the following to `TEJACC_CFLAGS` in the application `makefile`:

```
-DDEVID_1060
```

How Can I Change the Behavior of the Generated Makefile Without Modifying It?

You can create an auxiliary file that modifies the behavior of the generated `makefile` and then invoke the generated Makefile with the `EXTERNAL_MAKEFILE` variable set to this file name, Or use the "`external_makefile`" property in the software architecture (both mechanisms are explained below). This causes the generated `makefile` to include the file after setting up all the parameters but before invoking any compilation command. You can then overwrite any parameter that the generated Makefile is setting and the new value for that parameter will be in effect for the compilation.

You can specify a file name using the "`external_makefile`" property of the process. To set the new value for the property, invoke the following: `teja_process_set_property`(*process_obj*,"`external_makefile`","*new file name with or without path*")

If the path is not specified, the top level application directory will be assumed. The path can be relative to the top level application directory or an absolute value. There will not be any warning or error if the file does not exist, the compilation will continue with the generated Makefile parameters. If you prefer, you can also specify this external defines filename as a value to the `EXTERNAL_DEFINES` parameter during the compilation of the generated code. For example, `gmake EXTERNAL_DEFINES=../../user_defs.mk`. This value will take precedence over the value specified in the software architecture if both of the approaches are used.

An example of `user_defs.mk` is `USR_CFLAGS=-xO3`.

The generated Makefile can be invoked as:

```
make EXTERNAL_DEFINES=user_defs.mk
```

This has the effect of adding the `-xO3` flag to the compilation lines.

Late-Binding Questions

Note – Refer to "Overview of the Late-Binding API" on page 22 for more information on the Late-Binding API.

What Is the Late-Binding API?

The late-binding API is the Teja equivalent of OS system calls. However, OS calls are fixed in precompiled libraries, and late-binding API calls are generated based on contextual information. This situation ensures that the late-binding API calls are small and optimized. See "What Is Context-Sensitive Generation?" on page 92.

The late-binding API addresses the following services:

- Memory allocation by memory pools
- Communication through channels and queues
- Synchronization from mutex
- Waiting select-like on timeout and channels with `teja_wait()`.

What Is a Teja Memory Pool?

A memory pool is a portion of contiguous memory that is preallocated at system startup. The memory pool is subdivided into equal-sized nodes and allocated. You declare memory pools in the software architecture using `teja_memory_pool_declare()`. Memory pools enable you to choose size implementation type, producers, consumers, and so on. In the application code, you can write data to the channel using `teja_channel_send()` and read from the channel using `teja_wait()`. The `send` and `wait` primitives are Late-Binding API calls (see "What Is the Late-Binding API?" on page 84), so they benefit from context-sensitive generation.

What Is a Teja Channel?

A channel is a pipe-like mechanism to send data from one thread to another. Channels are declared in the software architecture using `teja_channel_declare()`, which enables you to choose the size and number of nodes, implementation type, and so on. In the application code, you can get nodes

from or put nodes in the memory pool, using `teja_memory_pool_get_node()` and `teja_memory_pool_put_node`. The allocation mechanism is more efficient than `malloc()` and `free()`. The `get_node` and `put_node` primitives are Late-Binding API calls, so they benefit from context-sensitive generation.

How Do I Access a Late-Binding Object From Application Code?

Use the `teja_`*late-binding-object-type*`_declare` call to declare all late-binding objects (memory pool, channel, mutex, queue) in the software architecture. The first parameter of this call is a string containing the name of the object. In the application code, the late-binding objects are accessed as a C preprocessor symbolic interpretation of the object name. The name is no longer a string. `tejacc` makes these symbols available to the application by processing the software architecture dynamic library.

Can I Define a Symbol in the Software Architecture and Use It in My Application Code?

The following function in the software architecture can define a C preprocessor symbol used in application code:

```
int teja_process_add_preprocessor_symbol (teja_process_t process, const char *
symbol, const char * value);
```

where:

- *process* – Process in which the symbol is defined.
- *symbol* – String containing the symbol name.
- *value* – String containing the symbol value.

Note – In the application, the symbol is accessed as a C preprocessor symbol, not as a string.

API and Application Questions

Note – Refer to the *Netra Data Plane Software Suite 1.1 Reference Manual* for detailed description of the API functions.

How Do I Synchronize a Critical Region?

Use the mutex API, which is composed of:

- `teja_mutex_declare()`
- `teja_mutex_lock()`
- `teja_mutex_unlock()`
- `teja_mutex_trylock()`

How Do I Send Data From a Thread to Another Thread?

Use the Channel API or the Queue API.

The Channel API is composed of:

- `teja_channel_declare()`
- `teja_channel_is_connection_open()`
- `teja_channel_make_connection()`
- `teja_channel_break_connection()`
- `teja_channel_send()`
- `teja_wait()`

The Queue API is composed of:

- `teja_queue_declare()`
- `teja_queue_enqueue()`
- `teja_queue_dequeue()`
- `teja_queue_is_empty()`
- `teja_queue_get_size()`

How Do I Allocate Memory?

Use the Memory Pool API, which is composed of:

- `teja_memory_pool_declare()`
- `teja_memory_pool_get_node()`
- `teja_memory_pool_put_node()`
- `teja_memory_pool_get_node_from_index()`
- `teja_memory_pool_get_index_from_node()`

What Changes Are Required for the RLP Application Under LDoms?

The RLP application requires some minimal changes to run under the LDoms enviroment.

1. **Update the `src/app/rlp.c` file to reflect the total number of strands given to the LWRTE domain.**
2. **Change the `src/app/rlp_config.h` file so that the `stat` thread is configured to run on the last strand (CPU number) available for your LDom partition.**

   The `NUM_STRANDS` macro requires this change from the default value of strand 31.
3. **Change the `src/config/rlp_map.c` and `src/config/rlp_swarch.c` files to reflect the new architecture of your LWRTE domain under LDoms.**

What Changes Are Required for the `ipfwd` Application Under LDoms?

The `ipfwd` application requires some minimal changes to run under the LDoms enviroment.

1. **In `apps/ipfwd./src/config/ipfwd_map.c`, do the following:**

   Comment out `teja_map_function_to_thread("_main", "main_thd16")` and `main_thd > 16`.
2. **In a`pps/ipfwd.10g/src/config/ipfwd_swarch.c`, do the following:**

   a. **Comment out**
   `processors[16] = "app.cmt1board.cmt1_chip.strand16";`
   and similar lines greater than 16.

b. Commnet out

```
thd[15] = teja_thread_create(init, "main_thd16");
```

and similar lines greater than 16.

c. Assign `stat_thd` to 15 by entering the following:

```
thd[15] = teja_thread_create(init, "stat_thd");
```

d. Comment out

```
thd[31] = teja_thread_create(init, "stat_thd");
```

3. **In `apps/ipfwd.10g/src/app/ipfwd_config.h`, change `NUM_STRANDS` to `16`.**

Note – Strand number of 16 is only used in this example. You can use any multiple of 4, from 4 through 28.

When Should I Use Queues Instead of Channels?

Generally, queues are more efficient than channels. Consider the following guidelines when deciding between queues or channels:

- Fast Queue functions have less code and overhead. Fast Queue functions are poll-driven, and so are more efficient for passing high-rate packet streams.
- Channels can accommodate variable data size and enables you to perform event-driven communication. Data is copied into the channel at the sender and copied out of the channel at the receiver.
- Channels enables you to send an event value to the receiver that distinguishes the type of received data. This capability is good for classifier applications and events that do not arrive regularly.
- The decision to use a queue instead of a channel depends on the application model. For example, if an `ipfwd` application does not require classification, Fast Queue is more efficient.

Why Is It Not Necessary to Block Interface or Queue Reads?

If a queue is used by one producer and one consumer, there is no need to block during the queue read. For example, in the `ipfwd` application, each queue has only one producer and consumer, and does not need to block. See FIGURE A-1.

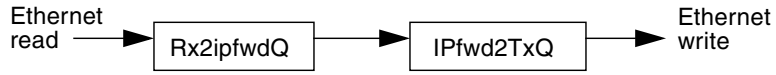

FIGURE A-1 Example for the `ipfwd` Application

Note – If the Teja Queue API is used instead of Fast Queue, then locks are generated implicitly during compile time.

It is not necessary to block Ethernet interface reads, as there is only one thread reading from or writing to a particular interface port or DMA channel at any given time.

Can Multiple Strands on the Same Queue Take Advantage of the Extra CPU Cycles If the Strands Are Not Being Used?

A strand is not being used or consuming the pipeline only when the strand is parked. Even when a strand is calling `teja_wait()`, the CPU consumes cycles because the strand does a busy wait. If the strand performs busy polls, the polls can be optimized so that other strands on the same CPU core utilize the CPU. This optimization is accomplished by executing instructions that release the pipeline to other strands until the instruction completes.

Consider IP-forwarding type applications. When the packet receiving stream approaches line rate, it is better to let the strand perform busy poll for arriving packets. At less than the line rate, the polling mechanism is optimized by inserting large instructions between polls. Under this methodology, the pipeline releases and enables other strands to utilize unused CPU cycles.

Why Does the Application Choose the Role for the Strand From the Code Instead of the Software Architecture API?

When the role is determined from the code, the application (for example, `ipfwd.c`) can be made more adaptable to the number of flows and physical interfaces without modifying any mapping files. However, this is your preference and in some situations, the Software Architecture API can provide a better role for a strand.

Is It Possible to Park a Strand Under LDoms Without Explicitly Specifying So in the Code?

Methods of parking strands are no different in an LDoms environment. Un-utilized strands are automatically parked. If a strand is assigned to a logical domain but is not used, then that strand should be parked. Strands that are not assigned to the LWRTE logical domain are not visible to that domain and cannot be parked.

You must assign complete cores to LWRTE. Otherwise, you have no control over the resources consumed by other domains on the core.

What Is `bss_mem`?

For example:

```
(ipfwd_map.c) (teja_map_variables_to_memory(".*", "app.cmt1board.bss_mem");
```

`bss_mem` is a location where all global and static variables are stored.

Note – The sum of `BSS` and the code size must not exceed 5 Mbytes of memory.

What Is the Significance of `bss_mem` Placement in the Code Listing?

When the example of "What Is `bss_mem`?" on page 90 is inserted into the code, all subsequent variables using `.*_dram` are superseded. To clarify, all variables suffixed with `_dram` are mapped to the DRAM memory region. All other variables are mapped to the `BSS`.

How Are `app.cmt1board.heap_mem0` and Similar Heaps Affected?

The heap region is used by `teja_malloc()`. Every time `teja_malloc()` is called, the heap space is reduced.

Can You Clarify BSS, Code, Heap, and DRAM Memory Allocation?

FIGURE A-1 illustrates the allocation of memory for BSS, code, heap, and DRAM.

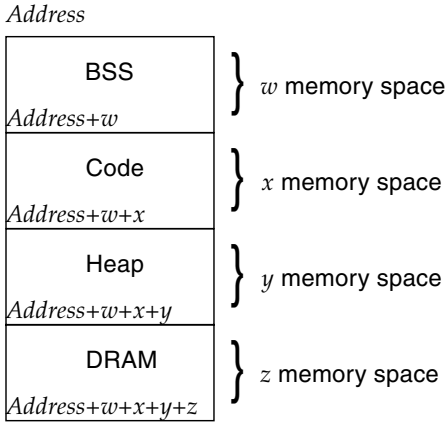

FIGURE A-2 Memory Allocation Stack

where:

- **BSS** – Global and static variables.
- **Code** – Code segment.
- **Heap** – Region for `teja_malloc()`.
- **DRAM** – Used for memory pools. For example, DMA buffers, descriptors, queue data, user application memory, and so on.

Why Are So Many Warnings Displayed When Compiling the `ipfwd` Code?

Some of the warnings are marginal warnings that are accepted by a regular C compiler, but not the `tejacc` compiler.

What Is LWIP Lib?

The Light Weight Internet Protocol Library (LWIP lib) consists of essential functions for implementing a basic User Datagram Protocol (UDP) or Transport Control Protocol/Internet Protocol (TCP/IP) stack.

Does the `eth_*` API Support Virtual Ethernet Devices?

The `eth_*` API supports only physical Ethernet devices at this time. IPC is designed to run on top of Logical Domain Communication (LDC, a HyperVisor protocol) and on the `eth_*` API.

Optimization Questions

How Do I Enable Optimization?

TABLE A-1 describes the options for `tejacc` to enable optimization:

TABLE A-1 Optimization Options for `tejacc`

Option for `tejacc`	Description
`-O`	Enables all optimizations.
`-fcontext-sensitive-generation`	Enables context sensitive generation only.

What Is Context-Sensitive Generation?

Context-sensitive generation is the ability of the `tejacc` compiler to generate minimal and optimized system calls based on global context information provided from:

- Hardware architecture
- Software architecture
- Mapping
- Function parameters
- User guidelines

In the traditional model, the operating system is completely separated from the compiler and the operating system calls are fixed in precompiled libraries. In the `tejacc` compiler, each system call is generated based on the context.

For example, if a shared memory communication channel is declared in the software architecture as having only one producer and one consumer, the `tejacc` compiler can generate that channel as a mutex-free circular buffer. On a traditional operating system the mutex would have to be included because the usage of the function call was not known when the library was built. See "Overview of the Late-Binding API" on page 22 for more information on the Late-Binding API.

What Is Global Inlining?

Functions marked with the `inline` keyword or with the `-finline` command-line option get inlined throughout the entire application, even across files.

Legacy Code Integration Questions

How Can I Reuse Legacy C Code in a Teja Application?

You can port preexisting applications to the Teja environment. There are two methods to integrate legacy application C code with newly compiled Teja application code:

- "Linking Legacy Code to Teja Code" on page 93
- "Changing Legacy Source Code" on page 94

Linking Legacy Code to Teja Code

By linking legacy code the to Teja code as libraries, the legacy code is not compiled and changes are minimized. The legacy library is also linked to the Teja generated code, so those libraries must be available on the target system, where performance is not an important factor.

For example, to port a UNIX legacy application to Teja running on UNIX is simple, because all of the UNIX libraries are available. However, porting a UNIX application to Teja running on bare hardware might require additional effort, because the bare hardware system does not have all the necessary OS libraries. In this situation, you must provide the missing functions.

Changing Legacy Source Code

Introducing calls to the Teja API in the legacy source code enables context-sensitive and late-binding optimizations to be activated in the legacy code. This method provides higher performance than the linking method.

Heavy memory allocation operations such as `malloc` and `free` are substituted with Teja preallocated memory pools, generated in a context-sensitive manner. The same advantage applies to mutexes, queues, communication channels, and functions such as `select()`, which are substituted with `teja_wait()`.

Note – It is not necessary to substitute all legacy calls with Teja calls as only performance-critical parts of legacy code need to be ported to Teja. Error handling and exception code can remain unchanged.

How Can I Reuse Legacy C++ Code in a Teja Application?

Note – See "How Can I Reuse Legacy C Code in a Teja Application?" on page 93.

C++ code can be integrated with a Teja application by two methods:

- "Mixing C and C++ Code" on page 95
- "Translating C++ Code to C Code" on page 95

Mixing C and C++ Code

Teja generates C code, so the final program is in C. Mixing C++ and Teja code is similar to mixing C++ and C code. This topic has been discussed extensively in C and C++ literature and forums. Basically, declare the C++ functions you call from Teja to have C linkage. For example:

```
#include <iostream>
extern "C" int print(int i, double d)
{
   std::cout << "i = " << i << ", d = " << d;
}
```

Compile the C++ code natively with the C++ compiler and link it to the generated Teja code. The Teja code can call the C++ functions with C linkage.

For detailed discussions of advanced topics such as overloading, templates, classes, and exceptions, refer to these URLs:

- http://developers.sun.com/sunstudio/articles/mixing.html
- http://www.parashift.com/c++-faq-lite/mixing-c-and-cpp.html

Translating C++ Code to C Code

The third-party packages at the following web sites can be used to translate code from C++ to C. Sun has not verified the functionality of these software programs:

- http://www.comeaucomputing.com
- http://www.desy.de/user/projects/C++/products/solbourne.html
- http://javashoplm.sun.com/ECom/docs/Welcome.jsp?StoreId=8&PartDetailId=GCC2C-1.1-MP-G-F&TransactionId=Try

---

Note – This latter URL requires registration with the Sun Download Center.

Sun CMT Specific Questions

Is There a Maximum Allowed Size for Text + BSS in My Program?

The limit is 5 Mbyte. If the application exceeds this limit, the generated `makefile` indicates so with a static check.

How Is Memory Organized in the Sun CMT Hardware Architecture?

TABLE A-2 lists the default memory setup in Sun CMT hardware architecture:

TABLE A-2 Default Memory Setup

Memory Address Space	Description
`0x00000000 - 0x11000000`	Reserved for system use.
`0x11000000 - 0x13000000`	Private heap memory for each strand. On CMT systems, there are 32 strands and each strand receives 1/32th of the memory from 0x11000000 to 0x13000000: the first strand has its heap from 0x11000000 to 0x11100000, the second one has its heap from 0x1110000 to 0x11200000, and so on. Heap memory is used by `teja_malloc`().
`0x13000000 - 0x100000000`	Shared DRAM. Variables that are mapped to DRAM are generated in the static memory map.

These values are changed in the memory bank properties of the hardware architecture. For example, to move the end of DRAM to `0x110000000`, add the following code to your hardware architecture:

```
teja_memory_t mem; char * new_value = "0x110000000"; ... mem =
teja_lookup_memory (board, "dram_mem"); teja_memory_set_property
(mem, TEJA_PROPERTY_MEMORY_SIZE, new_value);
```

How Do I Increase the Size of the DRAM `membank`?

You can increase the size of DRAM as explained in "How Is Memory Organized in the Sun CMT Hardware Architecture?" on page 96.

Address Resolution Protocol (ARP) Questions

How Do I Enable ARP in RLP?

To enable ARP in RLP, you need to make the following changes:

1. **Modify `rlp_config.h` to give IP addresses to the network ports.**

   For example,

   a. **Assign an IP address to the network ports of the system, running `lwrte`.**

```
 # define    IP_BY_PORT(port)     \
((port == 0)? __GET_IP(192, 12, 1, 2): \
(port == 1)? __GET_IP(192, 12, 2, 2): \
(port == 2)? __GET_IP(192, 12, 3, 2): \
(port == 3)? __GET_IP(192, 12, 4, 2): \
(0))
```

   b. **Tell the RLP application, the remote IP address to which its going to send IP packets.**

```
 # define DEST_IP_BY_PORT(port) \
((port == 0)? __GET_IP(192, 12, 1, 1): \
(port == 1)? __GET_IP(192, 12, 2, 1): \
(port == 2)? __GET_IP(192, 12, 3, 1): \
(port == 3)? __GET_IP(192, 12, 4, 1): \
(0))
```

c. **Assign** `netmask` **to each port, to define a subnet.**

```
# define NETMASK_BY_PORT(port) (0xffffff00)
```

2. **Compile the RLP application with** `ARP=on`

```
$ gmake clean
$ gmake ARP=on
```

APPENDIX **B**

Tuning

This appendix provides guidelines for diagnosing and tuning network applications running under the Lightweight Runtime Environment (LWRTE) on UltraSPARC® T1 processor multithreading systems.

Topics in this appendix include:

Netra Data Plane Software Introduction

The UltraSPARC T1 CMT systems deliver a strand-rich environment with performance and power efficiency that are unmatched by other processors. From a programming point of view, the UltraSPARC T1 processor's strand-rich environment can be thought of as symmetric multiprocessing on a chip.

The LWRTE provides an ANSI C development environment for creating and scheduling application threads to run on individual strands on the UltraSPARC T1 processor. With the combination of the UltraSPARC T1 processor and LWRTE, developers have an ideal platform to create applications for the fast path and the bearer-data plane space.

UltraSPARC T1 Processor Overview

The Sun UltraSPARC T1 processor employs chip multithreading, or CMT, which combines chip multiprocessing (CMP) and hardware multithreading (MT) to create a SPARC® V9 processor with up to eight 4-way multithreaded cores for up to 32 simultaneous threads. To feed the thread-rich cores, a high-bandwidth, low-latency memory hierarchy with two levels of on-chip cache and on-chip memory controllers is available. FIGURE B-1 shows the UltraSPARC T1 architecture.

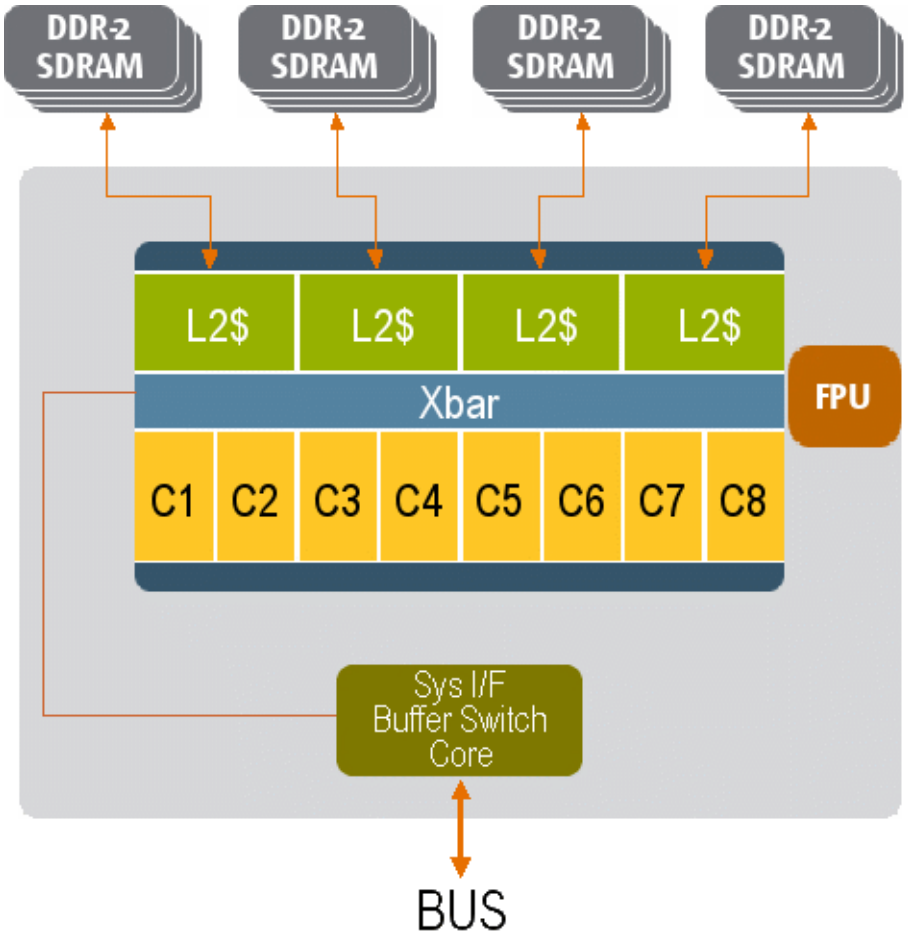

FIGURE B-1 UltraSPARC T1 Architecture

The processing engine is organized as eight multithreaded cores, with each core executing up to four strands concurrently. Each core has a single pipeline and can dispatch at most 1 instruction per cycle. The maximum instruction processing rate is 1 instruction per cycle per core or 8 instructions per cycle for the entire eight core chip. This document distinguishes between a hardware thread (*strand)*, and a software thread (*lightweight process (LWP)*) in Solaris.

A strand is the hardware state (registers) for a software thread. This distinction is important because the strand scheduling is not under the control of software. For example, an OS can schedule software threads on to and off of a strand. But once a software thread is mapped to a strand, the hardware controls when the thread executes. Due to the fine-grained multithreading, on each cycle a different hardware strand is scheduled on the pipeline in cyclical order. Stalled strands are switched out and their slot in the pipeline given to the next strand automatically. Thus, the maximum throughput of 1 strand is 1 instruction per cycle if all other strands are stalled or parked. In general, the throughput is lower than the theoretical maximums.

The memory system consists of two levels of on-chip caching and on-chip memory controllers. Each core has level 1 instruction and data caches and TLBs. The instruction cache is 16 Kbyte, the data cache is 8 Kbyte, and the TLBs are 64 entries each. The level 2 cache is a 3 Mbyte unified instruction, and it is 12-way set associative and 4-way banked. The level 2 cache is shared across all eight cores. All cores are connected through a crossbar switch to the level 2 cache.

Four on-chip DDR2 memory controllers provide low-latency, high-memory bandwidth of up to 25 Gbyte per sec. Each core has a modular arithmetic unit for modular multiplication and exponentiation to accelerate SSL processing. A single floating-point unit (FPU) is shared by all cores, so this software is not optimal for floating-point intensive applications. TABLE B-1 summarizes the key performance limits and latencies.

TABLE B-1 Key Performance Limits and Latencies

Feeds		Speeds
Processor instruction execution bandwidth		9.6 G instructions per sec (peak @ 1.2 GHz)
Memory		
	L1 hit latency	~ 2 cycles
	L2 hit latency	~ 23 cycles
	L2 miss latency	~ 90 ns
	Bandwidth	17 GBps (25 GBps peak)
I/O bandwidth		~ 2 GBps (JBus limitation)

Identifying Performance Issues

The key performance metric is the measure of throughput, usually expressed as either packets processed per second, or network bandwidth achieved in bits or bytes per second. In UltraSPARC T1 systems, the I/O limitation of 2 Gbyte per second puts an upper bound on the throughput metric. FIGURE B-2 shows the packet forwarding rate limited by this I/O bottleneck.

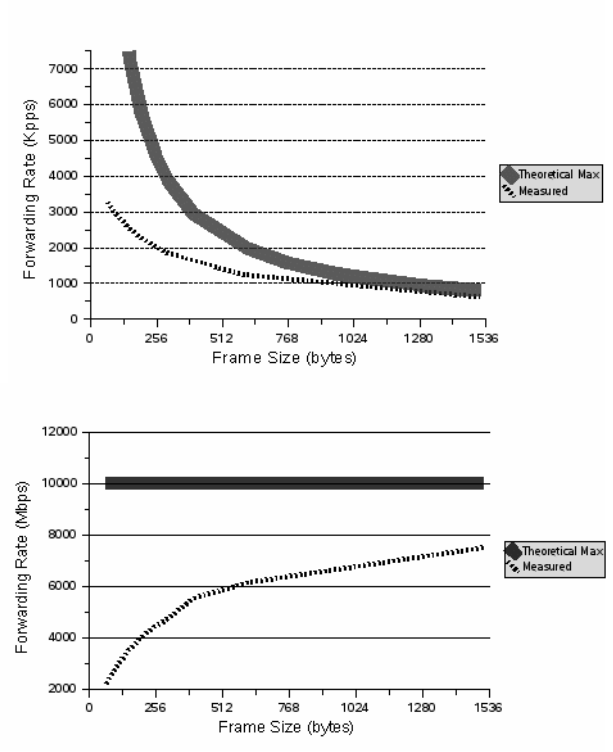

FIGURE B-2 Forwarding Packet Rate Limited by I/O Throughput

The theoretical max represents the throughput of 10 Gbyte per second. The measured results show that the achievable forwarding throughput is a function of packet size. For 64-byte packets, the measured throughput is 2.2 Gbyte per second or 3300 kilo packets per second.

In diagnosing performance issues, there are three main areas: I/O bottlenecks, instruction processing bandwidth, and memory bandwidth. In general, the UltraSPARC T1 systems have more than enough memory bandwidth to support the network traffic allowed by the JBus I/O limitation. There is nothing that can be done about the I/O bottleneck, so this document focuses on instruction processing limits.

For UltraSPARC T1 systems, the network interfaces are 1 Gbit and the interface is mapped to a single strand. In the simplest case, one strand is responsible for all packet processing from the corresponding interface. At a 1 Gbit line rate, 64-byte packets arrive at 1.44 Mpps (million packets per second) or one packet every 672 ns. To maintain this line rate, the processor must process the packet within 672 ns. On average, that is 202 instructions per packet. FIGURE B-3 shows the average maximum number of instructions the processor can execute per packet while maintaining line rate.

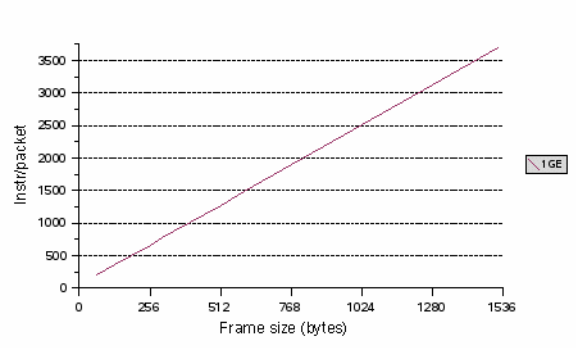

FIGURE B-3 Instructions per Packet Versus Frame Size

The inter-arrival time increases with packet size, so that more processing can be accomplished.

Profiling Application Performance

Profiling consists of instrumenting your application to extract performance information that can be used to analyze, diagnose, and tune your application. LWRTE provides an interface to assist you to obtain this information from your application. In general, profiling information consists of hardware performance counters and a few user-defined counters. This section defines the profiling information and how to obtain it.

Profiling is a disruptive activity that can have a significant performance effect. Take care to minimize profiling code and also to measure the effects of the profiling code. This can be done by measuring performance with and without the profiling code. One of the most disruptive parts of profiling is printing the profiling data to the console. To reduce the effects of prints, try to aggregate profiling statistics for many periods before printing, and print only in a designated strand.

The hardware counters are for the CPU, DRAM controllers, and JBus are described in TABLE B-2, TABLE B-3, and TABLE B-4 respectively.

TABLE B-2 CPU Counters

Event Name	Description
instr_cnt	Number of completed instructions. Annulled, mispredicted, or trapped instructions are not counted.*
SB_full	Number of store buffer full cycles.\
FP_instr_cnt	Number of completed floating-point instructions. d Annulled or trapped instruction are not counted.
IC_miss	Number of instruction cache (L1) misses.
DC_miss	Number of data cache (L1) misses for loads (store misses are not included because the cache is write-through nonallocating).
ITLB_miss	Number of instruction TLB miss trap taken (includes real_translation misses).
DTLB_miss	Number of data TLB miss trap taken (includes real_translation misses).
L2_imiss	Number of secondary cache (L2) misses due to instruction cache requests.
L2_dmiss_ld	Number of secondary cache (L2) misses due to data cache load requests.\

* Tcc instructions that are cancelled due to encountering a higher-priority trap are still counted.

\ SB_full increments every cycle a strand (virtual processor) is stalled due to a full store buffer, regardless of whether other strands are able to keep the processor busy. The overflow trap for SB_full is not precise to the instruction following the event that occurs when ovfl is set. The trap might occur on the instruction following the event or the following two instructions.

d Only floating-point instructions that execute in the shared FPU are counted. The following instructions are executed in the shared FPU: FADDS, FADDD, FSUBS, FSUBD, FMULS, FMULD, FDIVS, FDIVD, FSMULD, FSTOX, FDTOX, FXTOS, FXTOD, FITOS, FDTOS, FITOD, FSTOD, FSTOI, FDTOI, FCMPS, FCMPD, FCMPES, FCMPED.

\ L2 misses because stores cannot be counted by the performance instrumentation logic.

TABLE B-3 DRAM Performance Counters

Counter Name	Description
mem_reads	Number of read transactions.
mem_writes	Number of write transactions.
bank_busy_stalls	Number of bank busy stalls (when transactions are pending).
rd_queue_latency	Read queue latency (incremented by number of read transactions in the queue each cycle).
wr_queue_latency	Write queue latency (incremented by number of write transactions in the queue each cycle).
rw_queue_latency	Read + write queue latency (incremented by number of write transactions in the queue each cycle).
wr_buf_hits	Writeback buffer hits (incremented by 1 each time a read is deferred due to conflicts with pending writes).

TABLE B-4 JBus Performance Counters

Counter Name	Description
jbus_cycles	JBus cycles (time).
dma_reads	DMA read transactions (inbound).
dma_read_latency	Total DMA read latency.
dma_writes	DMA write transactions.
dma_write8	DMA WR8 sub transactions.
ordering_waits	Ordering waits (JBI to L2 queues blocked each cycle).
pio_reads	PIO read transactions (outbound).
pio_read_latency	Total PIO read latency.
pio_writes	PIO write transactions.
aok_dok_off_cycles	AOK or DOK off cycles seen.
aok_off_cycles	AOK_OFF cycles seen.
dok_off_cycles	DOK_OFF cycles seen.

Each strand has its own set of CPU counters that only tracks its own events and can only be accessed by that strand. Only two CPU counters are 32 bits wide each. To prevent overflows, the measurement period should not exceed 6 seconds. In general, keep the measurement period between 1 and 5 seconds. When taking measurements, ensure that the application's behavior is in a steady state. To check this, measure the

event a few times to see that it does not vary by more than a few percent between measurements. To measure all nine CPU counters, eight measurements are required. The application's behavior should be consistent over the entire collection period. To profile each strand on a 32-thread application, each thread must have code to read and set the counters. Sample code is provided in CODE EXAMPLE B-1. You must compile your own aggregate statistics across multiple strands or a core.

The JBus and DRAM controller counters are less useful. Since these resources are shared across all strands, only one thread should gather these counters.

The key user-defined statistic is the count of packets processed by the thread. Another statistic that can be important is a measure of idle time, which is the number of times the thread polled for a packet and did not find any packets to process.

The following example shows how to measure idle time. Assume that the workload looks like the following:

```
while(1)
   If( work_to_do ) {
    Do work
    Increment work_count
   } else {
    Increment idle_loop_count
   }
}
```

User-defined counters count the number of times through the loop no work was done. Measure the time of the idle loop by running idle loop alone (`idle_loop_time`). Then run real workload, counting the number of idle loops (`idle_loop_count`)

```
Idle_time = idle_loop_count * idle_loop_time
```

Profiling Metrics

You can calculate the following metrics after collecting the appropriate hardware counter data using the LWRTE profiling infrastructure. Use the metrics to quantify performance effects and help in optimizing the application performance.

- Instructions per cycle (IPC)

  Calculate this metric by dividing instruction count by the total number of ticks during a time period when the thread is in a stable state. You can also calculate the IPC for a specific section of code. The highest number possible is 1 IPC, which is the maximum throughput of 1 core of the UltraSPARC T1 processor.

- Cycles per instructions (CPI)

  This metric is the inverse of IPC. This metric is useful for estimating the effect of various stalls in the CPU.

- Instruction cache misses per instruction (`IC_miss` per instruction)

  Multiplying this number with the L1 cache miss latency helps estimate the cost, in cycles, of instruction cache misses. Compare this to the overall CPI to see if this is the cause of a performance bottleneck.

- L2 instruction cache misses per instruction (`L2_imiss` per instruction)

  This metric indicates the number of instructions that miss in the L2 cache, and enables you to calculate the contribution of instruction misses to overall CPI.

- Data cache misses per instruction (`DC_miss` per instruction)

  Data cache miss rate in combination with the L2 cache miss rate quantifies the effect of memory accesses. Multiplying this metric with data cache miss latency provides an indication of its effect (contribution) on CPI.

- L2 cache misses per instruction (`L2_miss` per instruction)

  Similar to data cache miss rate, this metric has higher cost in term of cycles of contribution to overall CPI. The metric also enables you to estimate the memory bandwidth requirements.

Optimization Techniques

Code Optimization

Writing efficient code and using the appropriate compiler option is the primary step in obtaining optimal performance for an application. Sun Studio 11 compilers provide many optimization flags to tune your application. Refer to the *Sun Studio 11: C Users Guide* for the complete list of optimization flags available. See "Reference Documentation" on page xiii. The following list describes some of the important optimization flags that might help optimize an application developed with LWRTE.

- Inlining

  Use the `inline` keyword declaration before a function to ensure that the compiler inlines that particular function. Inlining reduces the path length, and is especially useful for functions that are called repeatedly.

- Optimization level

  The `-xO[12345]` option optimizes the object code differently based on the number (level). Generally, the higher the level of optimization, the better the runtime performance. However, higher optimization levels can result in longer compilation time and larger executable files. Use a level of `-xO3` for most cases.

- `-xtarget=UltraT1`

  This option indicates that the target hardware for the application is an UltraSPARC T1 CPU and enables the compiler to select the correct instruction latencies for that processor.

- `-xprefetch` and `-xprefetch_level`

  Useful options if cache misses seem to be slowing down the application.

Pipelining

The thread-rich UltraSPARC T1 processor and the LWRTE programming environment enable you to easily pipeline the application to achieve greater throughput and higher hardware utilization. Pipelining involves splitting a function into multiple functions and assigning each to a separate strand, either on the same processor core or on a different core. You can program the split functions to communicate through Teja fast queues or channels.

One approach is to find the function with the most clock cycles per instruction (CPI) and then split that function into multiple functions. The goal is to reduce the overall CPI of the CPU execution pipeline. Splitting a large slow function into smaller pieces and assigning those pieces to different hardware strands is one way to improve the CPI of some subfunctions, effectively separating the slow and fast sections of the processing. When slow and fast functions are assigned to different strands, the CMT processor uses the execution pipelines more efficiently and improves the overall processing rate.

FIGURE B-4 shows how to split and map an application using fast queues and CMT processor to three strands.

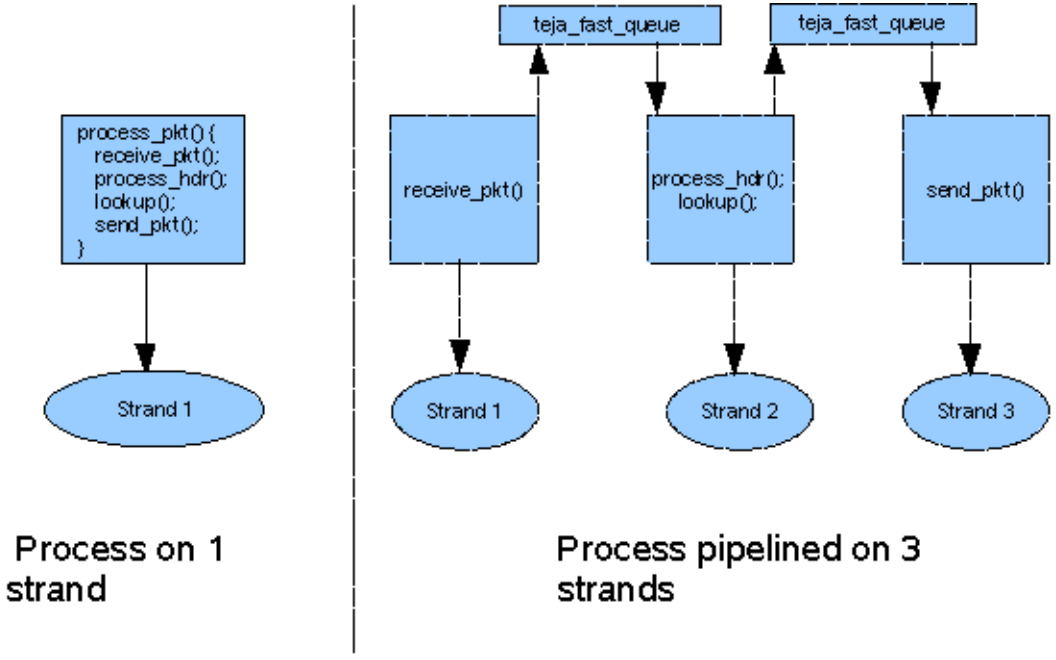

FIGURE B-4 Example of Pipelining

FIGURE B-5 shows how pipelining improves the throughput.

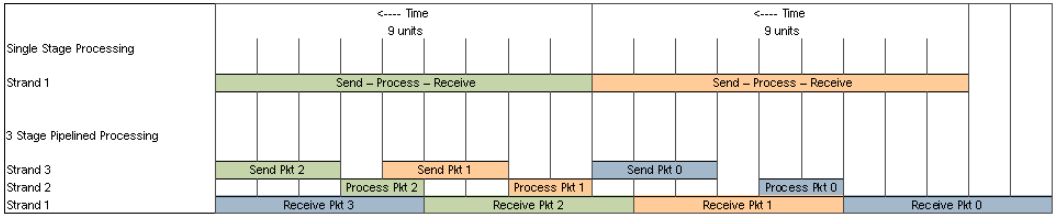

FIGURE B-5 Pipelining Effect on Throughput

In this example, a single-strand application takes nine units of time to complete processing of a packet. The same application split into three functions and mapped to three different strands takes longer to complete the same processing, but is able to process more packets in the same time.

Parallelization

The other advantage of a thread-rich CMT processor is the ability to easily parallelize an application. If a particular software process is very compute-intensive compared to other processes in the application, you can allocate multiple strands to this processing. Each strand executes the same code but works on different data sets. For example, since encryption is a heavy operation, the application shown in FIGURE B-6 is allocated three strands for encryption.

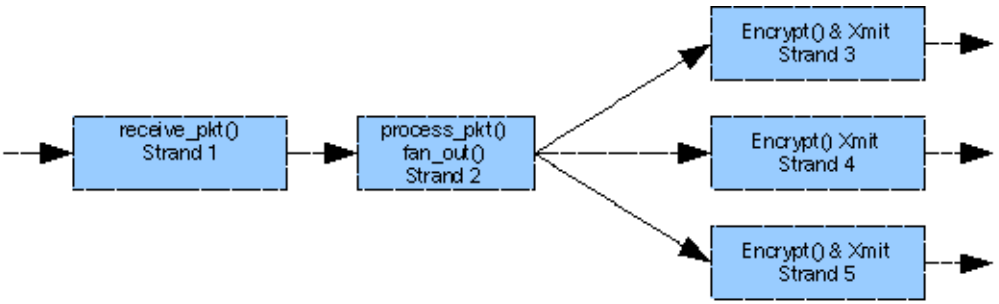

FIGURE B-6 Parallelizing Encryption Using Multiple Strands

The process strand uses some well-defined logic to fan out encryption processing to the three encryption strands.

Packet processing applications that perform identical processing repeatedly on different packets easily lend themselves to this type of parallelization. Any networking protocol that is compute-bound can be allocated on multiple strands to improve throughput.

Mapping

Four strands share an execution pipeline in the UltraSPARC T1 processor. There are eight such execution pipes, one for each core. Determining how to map threads (LWRTE functions) to strands is crucial to achieving the best throughput. The primary goal of performance optimization is to keep the execution pipeline as busy as possible, which means trying to achieve an IPC of 1 for each processor core.

Profiling each thread helps quantify the relative processing speed of each thread and provide an indication of the reasons behind the differences. The general approach is to assign fast threads (high IPC) with slow threads on the same core. On the other hand, if instruction cache miss is a dominant factor for a particular function, then you would want to assign multiple instances of the same function on the same core.

On UltraSPARC T1 processors you must assign any threads that have floating-point instructions to different strands if floating-point instructions are the performance bottleneck.

Parking Idle Strands

Often a workload does not have processing to run on every strand. For example, a workload has five 1 Gbit ports with each port requiring four threads for processing. This workload employs 20 strands for processing, leaving 12 strands unused or idle. You might run other applications on these idle strands but currently are testing only part of the application. LWRTE provides the options to park or to run `while(1)` loops on idle strands (that is, strands not participating in the processing).

Parking a strand means that there is nothing running on it and, therefore, the strand does not consume any of the processor's resources. Parking the idle strands produces the best result because the idle strands do not interfere with the working strands. The downside of parking strands is that there is currently no interface to activate a parked strand. In addition, activating a parked strand requires sending an interrupt to the parked strand, which might take hundreds of cycles before the strand is able to run the intended task.

If you want to run other processing on the idle strands, then parking these strands might result in optimistic performance measurements. When the final application is executed, the performance might be significantly lower than that measured with parked strands. In this case, running with a `while(1)` loop on the idle strands might be a more representative case.

The `while(1)` loop is an isolated branch. The `while(1)` loop executing on a strand takes execution resources that might be needed by the working strands on the same core to attain the required performance. `while(1)` loops only affect strands on the same core, they do not have an effect on strands on other cores. The `while(1)` loop often consumes more core pipeline resources than your application. Thus, if your working strands are compute-bound, running `while(1)` loops on all the idle strands is close to a worst case. In contrast, parking all the idle strands is the best case. To understand the range of expected performance, run your application with both parked and `while(1)` loops on the idle strands.

Slow Down Polling

As explained in "Parking Idle Strands" on page 111, strands executing on the same core can have both beneficial and detrimental effects on performance due to common resources. The `while(1)` loop is a large consumer of resources, often consuming more resources than a strand doing useful work. Polling is very common in LWRTE threads and, as seen with the `while(1)` loop, might waste valuable

resources needed by the other strands on the core to achieve performance. One way to alleviate the waste by polling is to slow down the polling loop by executing a long latency instruction. This situation causes the strand to stall, making its resources available for use by the other strands on the core. LWRTE exports interfaces to slowing down the polling that include:

- Access the memory location using a little endian load (ASI_PRIMARY_LITTLE). This option always goes to L2 and takes about 30 cycles.
- Meaningless CAS, which takes about 39 cycles.
- Meaningless PIO.
- ASI register read.
- Floating-point instructions.

The method you select depends on your application. For instance, if your application is using the floating-point unit, you might not want a useless floating-point instruction to slow down polling because that might stall useful floating-point instructions. Likewise, if your application is memory bound, using a memory instruction to slow polling might add memory latency to other memory instructions.

Tuning Troubleshooting

What Is a Compute-Bound Versus a Memory-Bound Thread?

A thread is compute-bound if its performance is dependent on the rate the processor can execute instructions. A memory-bound thread is dependent on the caching and memory latency. As a rough guideline for the UltraSPARC T1 processor, the CPI for a compute-bound thread is less than five and for a memory-bound thread is considerably higher than five.

Can't Reach Line Rate for Packets Smaller Than 300 Bytes

Single-thread receives, processes, and transmits packets can only achieve line rate for 300 byte packets or larger.

Goal: Want to get line rate for 250 byte packets.

Solution: Need to optimize single-thread performance. Try compiler optimization, different flags `-O2`, `-O3`, `-O4`, `-O5`, fast function inlining. Change code to optimize hot sections of code, you might need to do profiling.

Goal: Want to get to line rate for 64-byte packets.

Solution: Parallelize or pipeline. To get from 300 to 64-byte packets running at line rate is probably too much for just optimizing single-thread performance.

Can't Scale Throughput to Multiple Ports

When you increase the number of ports the results don't scale. For example, with a line rate of 400 byte packets with two interfaces, when you increase to three interfaces, you get only 90% of line rate.

Solution: If the problem is in parallelizing, determine if there are conflicts for shared resources, or synchronization and communication issues. Are there any lock contention or shared data structures? Is there a significant increase in CPI, cache misses, or store buffer full cycles? Are you using the shared resources such as the modular arithmetic unit or floating-point unit? Is the application at the I/O throughput bottleneck? Is the application at the processing bottleneck?

If there is a conflict for pipeline resources, optimizing single-thread performance would use fewer resources and improve overall throughput and scaling. In this situation, distribute the threads across the cores in a more optimal fashion or park unused strands.

How Do I Achieve Line Rate for 64-byte Packets?

The goal is to achieve line rate processing on 64-byte packets for a single 1 Gigabit Ethernet port. The current application requires 575 instructions per packet executing on 1 strand.

Solution: A 64-byte packet size has 202 instructions per packet. So optimizing your code will not be sufficient. You must parallelize or pipeline. In parallelization, the task is executed in multiple threads, each thread doing the identical task. In pipelining, the task is split up into smaller subtasks, each running on a different thread, that are sequentially executed. You can use a combination of parallelization and pipelining.

In parallelization, parallelize the task *N* ways, to increase the instructions per packet *N* times. For example, execute the task on three threads, and each thread can now have 606 instructions per packet (202 x 3) and still maintain 1 Gbit line rate for 64-byte packets. If the task requires 575 instructions per packet, run the code on 3 threads (606 instruction per packet), to achieve 1 Gbit line rate for 64-byte packets.

Parallelizing maximizes the throughput by duplicating the application on multiple strands. However, some applications cannot be parallelized or depend too much upon synchronization when executed in parallel. For example, the UltraSPARC T1 network driver is difficult to parallelize.

In pipelining, you can increase the amount of processing done on each packet by partitioning the task into smaller subtasks that are then run sequentially on different strands. Unlike parallelization, there are not more instructions per packet on a given strand. Using the example from the previous paragraph, split the task into three subtasks, each executing up to 202 instructions of the task. In both the parallel and pipelined cases, the overall throughput is similar at three packets every 575 instructions. Similar to parallelization, not all applications can easily be pipelined and there is overhead in passing information between the pipe stages. For optimal throughput, the subtasks need to execute in approximately the same time, which is often difficult to do.

When Should I Consider Thread Placement?

Thread placement refers to the mapping of threads onto strands. Thread placement can improve performance if the workload is instruction-processing bound. Thread placement is useful in cases where there are significant sharing or conflicts in the L1 caches, or when the compute-bound threads are grouped on a core. In the case of conflicts in the L1 caches, put the threads that conflict on different cores. In the case of sharing in the L1 caches, put the threads that share on the same core. In the case of compute-bound threads fighting for resources, put these threads on different cores. Another method would be to place high CPI threads together with low CPI threads on the same core.

Other shared resources that might benefit from thread placement include TLBs and modular arithmetic units. There are separate instruction and data TLBs per core. TLBs are similar to the L1 caches in that there can be both sharing and conflicts. There is only one modular arithmetic unit per core, so placing threads using this unit on different cores might be beneficial.

Example Exercise

This section uses the reference application RLP to analyze the performance of two versions of an application. The versions of the application are functionally equivalent but are implemented differently. The profiling information helps to make decisions regarding pipelining and parallelizing portions of the code. The information also enables efficient allocation of different software threads to strands and cores.

Application Configuration

The RLP reference application has three basic components:

- PDSN
- ATIF
- RLP

The PDSN and ATIF each have receive (RX) and transmit (TX) components. A Netra T2000 system with four in-ports and four out-ports was configured the four instances of the RLP application. FIGURE B-7 describes the architecture.

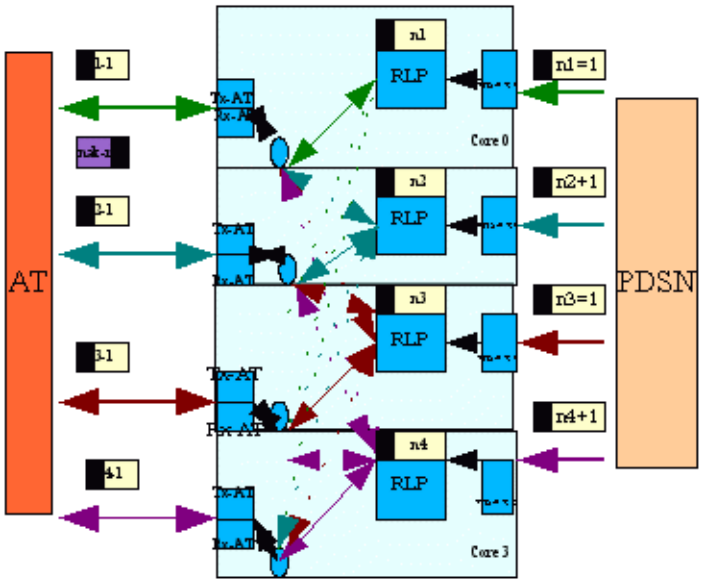

FIGURE B-7 RLP Application Setup

In the application, the flow of packets from PDSN to AT is the forward path. The RLP component performs the main processing. The PDSN receives packets (PDSN_RX) and forwards the packets to the RLP strand. After processing the packet header, the RLP strand forwards the packet to the AT strand for transmission (ATIF_TX). Summarizing:

- -> PDSN_RX -> RLP -> ATIF_TX -> (forward path)
- <- PDSN_TX <- RLP <- ATIF_RX <- (reverse path)

The example focuses on the forward path performance only.

Configuration 1

In configuration 1, the PDSN, ATIF, and RLP functionality is assigned to different threads as shown in TABLE B-5.

TABLE B-5 Configuration 1

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Strand 0	PDSN_RXTX_0	PDSN_RXTX_2	while(1)	while(1)	while(1)	RLP_0	while(1)	while(1)
Strand 1	ATIF_RXTX_0	ATIF_RXTX_3	while(1)	while(1)	while(1)	RLP_1	while(1)	while(1)
Strand 2	PDSN_RXTX_1	PDSN_RXTX_4	while(1)	while(1)	while(1)	RLP_2	while(1)	Profile thread
Strand 3	ATIF_RXTX_1	ATIF_RXTX_4	while(1)	while(1)	while(1)	RLP_3	while(1)	Stat thread

Configuration 2

In configuration 2, the PDSN and ATIF functionality is split into separate RX and TX functions and assigned to different strands as shown in TABLE B-6.

TABLE B-6 Configuration 2

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Strand 0	PSDN_RX_0	PSDN_RX_1	PSDN_RX_2	PSDN_RX_3	while(1)	while(1)	PSDN_TX_1	while(1)
Strand 1	RLP_0	RLP_1	RLP_2	RLP_3	while(1)	while(1)	PSDN_TX_2	while(1)
Strand 2	ATIF_RX_0	ATIF_RX_1	ATIF_RX_2	ATIF_RX_3	while(1)	while(1)	PSDN_TX_3	Profile thread
Strand 3	ATIF_TX_0	ATIF_TX_1	ATIF_TX_2	ATIF_TX_3	while(1)	PSDN_TX_0	while(1)	Stat thread

Using the Profiling API

It is important to understand hardware counter data collected from the strands that have been assigned some functionality. The strands assigned `while(1)` loops take up CPU resources but are not analyzed in this study. This study analyzes overall thread performance by sampling hardware counter data. After the application has reached a steady state, the hardware counters are sampled at predetermined intervals. Sampling reduces the performance perturbations of profiling and averages out small differences in the hardware counter data collected. In both versions of the application, the profiling affected performance by about 5-7% in overall throughput. The goal is to have the application in a steady state with profiling on.

The analysis uses the Teja Profiling API (refer to the *Netra Data Plane Software Suite 1.1 Reference Manual*) and creates a simple function that collects hardware counter data for all the available counters per strand. The function is called from a relevant section of the application. The hardware counter data is related to application performance as the number of packets processed by the application-defined counter that is passed to the Teja API. To reduce the performance impact of profiling, the profiling API is not called for each packet processed. For the RLP application and Netra T2000 hardware combination, the API is called every five seconds, otherwise the counters overflow.

The pseudo-code in CODE EXAMPLE B-1 shows the functions that were created to collect the hardware counter data.

CODE EXAMPLE B-1 Sample Code to Cycle Through UltraSPARC T1 Processor Hardware Counters

```
#ifdef TEJA_PROFILE
/* some global vars */
int event[MAX_CPUS];
uint64_t start_profile_value[MAX_CPUS]; /* when  to start collection hw counter data */
uint64_t update_interval_value[MAX_CPUS]; /* when to move to the next counter */
int number_profile_samples[MAX_CPUS]; /* number of samples to be taken before dumping */
int dump_enable[MAX_CPUS]; /* 0 = Dump Disabled 1 = Dump enabled */
int samples_collected[MAX_CPUS]; /* running count of samples collected */
/* set up control values for collection all CPU hardware counter */
inline void init_profiler(uint64_t start_val, uint64_t interval, int num_samples){
  int cpuid = teja_get_cpu_number();
  event[cpuid] = 1;
  number_profile_samples[cpuid] = num_samples;
  start_profile_value[cpuid] = start_val;
  update_interval_value[cpuid] = interval;
  dump_enable[cpuid] = 0;
  samples_collected[cpuid] = 0;
}
/* pass the value to be compared against for control */
/* this can be time/packet count */
inline void collect_profile(uint64_t user_value){
  int ret;
  int cpuid = teja_get_cpu_number();
  if (user_value ==  start_profile_value[cpuid] ) {
    ret = teja_profiler_start(TEJA_PROFILER_CMT_CPU, event[cpuid] );
    if (ret == -1)
      printf("Error Starting Profile \n");
  }
  if ((user_value % update_interval_value[cpuid] )==0) {
    ret = teja_profiler_update(TEJA_PROFILER_CMT_CPU, user_value);
    if (ret == -1)
      printf("Error Updating Profile \n");
    event[cpuid] = event[cpuid] * 2 ;
    if (event[cpuid]==256){
      event[cpuid] = 1;
      samples_collected[cpuid]++;
      if (samples_collected[cpuid] == number_profile_samples[cpuid] ){
        dump_enable[cpuid] = 1;
        /* there is a race here but the side effect is benign as Teja should print*/
        /* appropriate records when things get over-written */
        samples_collected[cpuid] = 0;
      }
    }
    /* 256 is 2^8 8 is number of HW counter in N1 */
    ret = teja_profiler_start(TEJA_PROFILER_CMT_CPU, event[cpuid] );
    if (ret == -1)
      printf ("Error Starting  Profiler\n");
  }
}
inline void
dump_hw_profile(){
  int cpuid;
  for (cpuid = 0 ; cpuid < MAX_CPUS ; cpuid++){
    if (dump_enable[cpuid] == 1){
      teja_profiler_dump(cpuid);
      dump_enable[cpuid] = 0;
    }
  }
}
#endif
```

The code uses the `teja_profiling_api` to create a simple set of functions for collecting hardware counter data. The code is just one example of API usage, but it is a very good starting point for performance analysis of a LWRTE application.

Each strand that does useful work is annotated with a call to the `collect_profile()` function and is passed the number of packets that have been processed. The location in the code where the call is made is important. In this application, the call is made in the active section of the code where a packet returned is not `null`. The `init_profiler()` function call sets up the starting point, an interval, and number of samples to be collected. The `dump_hw_profile()` function is called in the statistics strand and prints the data to the console.

Profiling Data

The API calls `teja_profile_start` and `teja_profiler_update` to set up and collect a specific pair of hardware counters. The call to `teja_profile_dump` outputs the collected statistics to the console. These function calls are in bold in CODE EXAMPLE B-1. For a detailed description of these API functions refer to the *Netra Data Plane Software Suite 1.1 Reference Manual*.

A sample output based on the code in CODE EXAMPLE B-1 is shown in CODE EXAMPLE B-2.

CODE EXAMPLE B-2 Sample Profile Output

```
PROFILE_DUMP_START,ver,1.1
CPUID,ID,Type,Cycles,PC,Grp,Evt_Hi,Evt_Lo,Overflow,User Data
4,6043,2,30051d74250,512598,1,3a372e12,dc22fb0,0,30c1b080
4,1fad3,1,30051d74c70,525968,1,100,2
4,6043,2,3021dd890b0,512598,1,3a3215c1,0,0,30e03500
4,1fad3,1,3021dd89abc,525968,1,100,4
4,6043,2,303e9d9e3e0,512598,1,3a2ee368,15561,0,30feb980
4,1fad3,1,303e9d9ee4c,525968,1,100,8
4,6043,2,305b5db43b0,512598,1,3a2ef375,29d8db7,0,311d3e00
4,1fad3,1,305b5db4db0,525968,1,100,10
4,6043,2,30781dc9ae0,512598,1,3a2f5793,0,0,313bc280
4,1fad3,1,30781dca544,525968,1,100,20
4,6043,2,3094dddeb10,512598,1,3a303d12,0,0,315a4700
4,1fad3,1,3094dddf51c,525968,1,100,40
4,6043,2,30b19df3258,512598,1,3a2ebfbf,6774,0,3178cb80
4,1fad3,1,30b19df3ccc,525968,1,100,80
4,6043,2,30ce5e08248,512598,1,3a2eb2aa,8c9c8f,0,31975000
4,1fad3,1,30ce5e08e24,525968,1,100,1
4,6043,2,30eb1e1e37c,512598,1,3a2f090e,dbbe5ae,0,31b5d480
4,1fad3,1,30eb1e1eea0,525968,1,100,2
4,6043,2,3107de334a8,512598,1,3a2f958f,0,0,31d45900
4,1fad3,1,3107de33f9c,525968,1,100,4
4,6043,2,31249e48ba8,512598,1,3a2fe948,1564a,0,31f2dd80
PROFILE_DUMP_END
```

All the numbers in the output are hexadecimal. This format can be imported into a spreadsheet or parsed with a script to calculate the metrics discussed in "Profiling Metrics" on page 106. The output in CODE EXAMPLE B-2 shows two types of records that correspond to `teja_profile_start` and `teja_profile_update` calls.

- An example of a `teja_profile_start` record:

```
4,1fad3,1,30051d74c70,525968,1,100,2
```

This record is formatted as: CPUID, ID, Call Type, Tick Counter, Program Counter, Group Type, Hardware counter 1 code, and Hardware counter 2 code. There is one such record for every call to `teja_profiler_start` indicated by a `1` in the Call Type (third) field.

- An example of a `teja_profile_update` record:

```
4,6043,2,31249e48ba8,512598,1,3a2fe948,1564a,0,31f2dd80
```

This record is formatted as: CPUID, ID, Call Type, Tick Counter, Program Counter, Group Type, Counter Value 1, Counter Value 2, Overflow Indicator and user defined data. There is one such record for every call to `teja_profile_update` indicated by a `2` in the Call Type field.

Metrics

The data from the output is processed using a spreadsheet to calculate the metrics per strand as presented in TABLE B-7.

TABLE B-7 Metrics

Metrics	Description
Instructions per packet	Average path length to process 1 packet
Instructions per cycle	Strand's instruction processing rate
Packet rate (Kpps)	Packet processing rate
`SB_full` per 1000 instructions `FP_instr_cnt` per 1000 instructions `IC_miss` per 1000 instructions `DC_miss` per 1000 instructions `ITLB_miss` per 1000 instructions `DTLB_miss` per 1000 instructions `L2_imiss` per 1000 instructions `L2_dmiss_ld` per 1000 instructions	The hardware counter rates per 1000 instructions allows comparison rates from different strands.

These metrics in TABLE B-7 provide insight into the performance of each strand and of each core.

Results

Configuration 1

Configuration 1 sustained 224 kpps (kilo packets per second) on each of the four flows or 65% of 1 Gbps line rate for a 342 byte packet. Only three cores of the UltraSPARC T1 processor were used to achieve this throughput. See FIGURE B-8.

65% Line		342 Byte Packet			4in-4out							
					HW Counter / 1000 Instruction							
Strands		Instruction /Packet	Instructions/Cycle	Packet Rate(Kpps)	SB_full	FP_instr_cnt	IC_miss	DC_miss	ITLB_miss	DTLB_miss	L2_imiss	L2_dmiss_ld
0	PDSN_RXTX	588.60	0.11	224.84	279.62	0.00	0.86	76.69	0.00	0.00	0.03	16.00
1	ATIF_RXTX	426.67	0.08	224.84	5.22	0.00	0.75	82.19	0.00	0.00	0.02	8.63
2	PDSN_RXTX	591.95	0.11	224.84	277.09	0.00	0.89	73.67	0.00	0.00	0.03	16.55
3	ATIF_RXTX	433.76	0.08	224.84	5.50	0.00	0.77	83.45	0.00	0.00	0.02	8.78
4	PDSN_RXTX	412.04	0.08	224.84	2.77	0.00	0.41	88.04	0.00	0.00	0.03	15.21
5	ATIF_RXTX	482.02	0.09	224.84	24.01	0.00	0.59	77.00	0.00	0.00	0.02	8.51
6	PDSN_RXTX	588.60	0.11	224.84	277.67	0.00	1.29	81.49	0.00	0.00	0.03	16.07
7	ATIF_RXTX	436.00	0.08	224.84	7.68	0.00	0.71	79.90	0.00	0.00	0.02	9.03
8	While(1)											
9	While(1)											
10	While(1)											
11	While(1)											
12	While(1)											
13	While(1)											
14	While(1)											
15	While(1)											
16	While(1)											
17	While(1)											
18	While(1)											
19	While(1)											
20	RLP	1180.86	0.22	224.84	46.10	0.00	0.01	21.54	0.00	0.00	0.01	5.58
21	RLP	1182.38	0.22	224.84	46.26	0.00	0.01	21.67	0.00	0.00	0.01	5.58
22	RLP	1449.37	0.27	224.84	42.15	0.00	0.01	11.24	0.00	0.00	0.01	0.79
23	RLP	1182.21	0.22	224.84	47.95	0.00	0.01	21.96	0.00	0.00	0.01	5.58
24	While(1)											
25	While(1)											
26	While(1)											
27	While(1)											
28	While(1)											
29	While(1)											
30	Profile Thread											
31	Stat Thread											

FIGURE B-8 Results From Configuration 1

Configuration 2

Configuration 2 sustained 310 kpps (kilo packets per second) on each of the four flows or 90% of 1 Gbps line rate for a 342 byte packet. Four cores of the UltraSPARC T1 processor were used to achieve this throughput. The `Polling` notation implies that the `ATIF_RX` thread was allocated to a strand, but no packets were handled by that thread during the test. See FIGURE B-9.

90% Line Rate				342 Byte Packet			4in-4out					
					HW Counter / 1000 Instruction							
Strands		Instruction /Packet	Instruction s/Cycle	Packet Rate(Kpps)	SB_full	FP_inst r_cnt	IC_miss	DC_miss	ITLB_ miss	DTLB_ miss	L2_imis s	L2_dmis s_ld
0	PDSN_RX	452.24	0.12	310.98	193.79	0	0.28	62	0	0	0.03	16.67
1	RLP	986.36	0.26	310.98	203.52	0	0.05	14.34	0	0	0.01	6.47
2	ATIF_RX	Polling										
3	ATIF_TX	1368.48	0.35	310.98	1.72	0	0.04	5.68	0	0	0	0.89
4	PDSN_RX	452.77	0.12	310.98	194.67	0	0.28	61.13	0	0	0.03	17.25
5	RLP	990.63	0.26	310.98	202.71	0	0.05	17.12	0	0	0.01	6.44
6	ATIF_RX	Polling										
7	ATIF_TX	1366.77	0.35	310.98	1.73	0	0.04	6.91	0	0	0	0.88
8	PDSN_RX	296.25	0.08	310.98	0.53	0	0.21	66.32	0	0	0.02	17.89
9	RLP	1320.54	0.34	310.98	206.43	0	0.03	4.98	0	0	0	0.75
10	ATIF_RX	Polling										
11	ATIF_TX	1355.82	0.35	310.98	1.75	0	0.03	6.49	0	0	0	0.89
12	PDSN_RX	452.02	0.12	310.98	193.69	0	0.28	60.94	0	0	0.03	17.31
13	RLP	994.12	0.26	310.98	202.82	0	0.05	16.33	0	0	0.01	6.42
14	ATIF_RX	Polling										
15	ATIF_TX	1371.24	0.36	310.98	1.75	0	0.04	6.35	0	0	0	0.88
16	While(1)											
17	While(1)											
18	While(1)											
19	While(1)											
20	While(1)											
21	While(1)											
22	While(1)											
23	PDSN_TX	Polling										
24	PDSN_TX	Polling										
25	PDSN_TX	Polling										
26	PDSN_TX	Polling										
27	While(1)											
28	While(1)											
29	While(1)											
30	Profile Thread											
31	Stat Thread											

FIGURE B-9 Results From Configuration 2

Analysis

When comparing the processed hardware counter information it is necessary to co-relate that data with the collection method. The counter information was sampled over the steady-state run of the application. There are other methods to collect hardware counter data that enable you to optimize a particular section of the application.

Comparing the Instruction per Cycle columns from the two tables shows that `RXTX` threads in configuration 1 are slower than the split `RX` and `TX` threads in configuration 2. The focus is on the forward path processing. Consider the following:

- For configuration 1 – `PDSN_RXTX -> RLP -> ATIF_RXTX`
- For configuration 2 – `PDSN_RX -> RLP -> ATIF_TX`

The main bottleneck in configuration 1 is the combined `ATIF_RXTX` thread that runs at the slowest rate, taking about 12 cycles per instruction. In configuration 2, `ATIF_RX` is moved to another strand and the bottleneck in the forward path (that does not need `ATIF_RX`) is removed, allowing `ATIF_TX` to run at a considerably faster 2.82 cycles per instruction. Also in configuration 2, using another strand speeded up the slowest section of pipelined processing. To speed up this configuration even more would require optimizing `PDSN_RX`, which is now the slowest part of the pipeline taking up 8.53 cycles per instruction. This optimization can be accomplished by optimizing code to reduce the number of instructions per packet or by splitting up this thread using more strands.

To explain the high CPI of the `ATIF_RXTX` strand in configuration 1, note that there are 82 `DC_misses` (dcache misses) per 1000 instructions as compared to just six misses in the `ATIF_TX` of configuration 2. You can estimate the effect of these misses by calculating the number of cycles these misses add to overall processing. Use information from TABLE B-1 to calculate the worst case effect of the data cache and L2 cache misses. The results for these calculations are shown in TABLE B-8 for configuration 1 and in TABLE B-9 for configuration 2.

TABLE B-8 Effect of Dcache and L2 Cache Misses on CPI – Configuration 1

	CPI	Cycle per Dcache Miss	Dcache Miss Effective %	Cycles per L2 Miss	L2 Miss Effective %
`PDSN_RXTX`	9.07	1.76	19.45	1.73	19.05
`ATIF_RXTX`	12.51	1.89	15.11	0.93	7.46
`PDSN_RXTX`	9.02	9.02	9.02	9.02	9.02
`ATIF_RXTX`	1.69	1.69	1.69	1.69	1.69

TABLE B-9 Effect of Dcache and L2 Cache Misses on CPI – Configuration 2

	CPI	Cycle per Dcache Miss	Dcache Miss Effective %	Cycles per L2 Miss	L2 Miss Effective %
PDSN_RX	8.53	1.43	16.71	1.8	21.1
RLP	3.91	0.33	8.43	0.7	17.86
ATIF_RX					
ATIF_TX	2.82	0.13	4.63	0.1	3.39

The highlighted rows show that the CPI contribution of dcache and L2 cache misses in configuration 1 is much higher than configuration 2, making the ATIF_RXTX strand much slower.

There are other effects involved here besides those outlined in the preceding tables. The move to put the RLP on the same core as PDSN_RX and ATIF_TX causes constructive sharing in the level 1 instruction and data caches as seen in the DC_misses per 1000 instructions for RLP strand. Another effect is that the slower processing rate of configuration 1 causes the RLP strand to spin on null more often, increasing the number of instruction per packet metric and slowing down processing. Other experiments have shown that threads that poll or do the while(1) loop take away processing bandwidth from other more useful threads.

In conclusion, configuration 2 achieves a higher throughput because the ATIF processing was split to RX and TX and each was mapped to a different strand, effectively parallelizing the ATIF thread. Configuration 2 used more strands, but was able to achieve much higher throughput.

Other Uses

The same teja_profiling_api can be used in another way to evaluate and understand the performance of an application. Besides the sampling method outlined in the preceding section, you can use the API to profile specific sections of the code. This type of profiling enables you to make decisions regarding pipelining and reorganizing memory structures in the application.

Index