Netra™ Data Plane Software Suite 2.0 User's Guide

Sun Microsystems, Inc.
www.sun.com

Part No. 820-3362-10
April 2008, Revision A

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

Please
Recycle

Contents

Figures

Tables

Preface

This user's guide provides information regarding the operation and use of the Netra™ Data Plane Software Suite 2.0. This document is written for software engineers, developers, programmers, and users who have advanced experience with low-level programming.

Netra Data Plane Software is also referred to in this document as Netra DPS.

How This Document Is Organized

Chapter 1 is an introduction to Netra Data Plane Software Suite 2.0, and provides installation and theoretical information.

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

Chapter 3 discusses the profiler used in the Netra Data Plane software.

Chapter 4 describes the Netra DPS native debugger and GNU debugger (GDB).

Chapter 5 describes interprocess communication (IPC) support.

Chapter 6 describes the remote command-line-interface (CLI).

Chapter 7 describes the the Eclipse-based Teja Advance Development Environment.

Chapter 8 describes the basic functions of the Receive Packet Classifier and the Netra DPS software interface.

Chapter 9 describes reference applications including IP packet forwarding and radio link protocol (RLP) applications..

Chapter 10 describes provides guidelines for diagnosing and tuning network applications running under the Lightweight Runtime Environment (LWRTE).

Appendix A is a tutorial to `tejacc` programming.

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

Glossary

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 following table lists the documentation for this product. The online documentation is available at:

http://docs.sun.com/app/docs/prod/netra.dp

Application	Title	Part Number	Format	Location
Operation	*Netra Data Plane Software Suite 2.0 User's Guide*	820-3362-10	PDF	online
Reference	*Netra Data Plane Software Suite 2.0 Reference Manual*	820-3363-10	PDF	online
Last-minute information	*Netra Data Plane Software Suite 2.0 Release Notes*	820-3364-10	PDF	online
Documentation Location	*Netra Data Plane Software Suite 2.0 Getting Started Guide*	820-3365-10	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
- *CoolThreads — CMT Application Tuning and UltraSPARC T2 Server Resources*
  http://www.sun.com/servers/coolthreads/tnb/t2.jsp
- *Sun Studio 12: C User's Guide*
  http://docs.sun.com/app/docs/doc/819-5265
- *Netra Data Plane Software Suite 2.0 Reference Manual* (`tejacc` 4.0 for Sun CMT reference manual)

  Available in the Netra Data Plane Software Suite 2.0 release package.
- *UltraSPARC T1 Supplement to UltraSPARC Architecture 2005 Specification*
  http://opensparc-t1.sunsource.net/index.html
- *UltraSPARC Architecture 2007 Specification and OpenSPARC T2 Implementation-Supplement*
  http://www.opensparc.net/opensparc-t2
- *Logical Domains (LDoms): Sun SPARC CMT Virtualization Technology*
  http://www.sun.com/servers/coolthreads/ldoms/index.xml
- *Eclipse: An Open Development Platform*: http://www.eclipse.org/
- *GDB: The GNU Project Debugger*: http://sourceware.org/gdb/
- Information and documentation for the Logical Domains (LDoms) virtualization technology: http://www.sun.com/ldoms

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 2.0 User's Guide, part number 820-3362-10

CHAPTER 1

Netra Data Plane Software Suite Overview

This chapter is an introduction to the Netra Data Plane Software Suite 2.0, and provides installation and theoretical information. Topics include:

Product Description

The Netra Data Plane Software (Netra DPS) Suite 2.0 is a complete board software package solution. The software provides an optimized rapid development and runtime environment on top of multistrand partitioning firmware for Sun CMT platforms. The software enables a scalable framework for fast-path network processing. Netra DPS 2.0 includes 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 2.0 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 a superior code validation and optimization, enabling more reliable and higher performance systems.

Supported Systems

Netra DPS 2.0 supports the following Sun UltraSPARC® T1 and UltraSPARC T2 platforms:

- Sun Fire™ T1000 system (OEM variants)
- Sun Fire T2000 system (OEM variants)
- Sun SPARC® Enterprise T5210 system (OEM variants)
- Sun SPARC Enterprise T5220 system (OEM variants)
- Netra T2000 system
- Netra T5220 system
- Netra ATCA CP3060 system
- Netra ATCA CP3260 system

Software Installation

The Netra DPS Suite 2.0 is distributed for SPARC platforms.

- `Netra_Data_Plane_Software_Suite_2.0.zip` contains the `SUNWndps` and `SUNWndpsd` packages.
- `Netra_Data_Plane_Software_Suite_Crypto_2.0.zip` contains the `SUNWndpsc` package.

The `SUNWndps` and `SUNWndpsc` packages are installed in the development server. The `SUNWndpsd` package is installed on the target deployment system.

TABLE 1-1 describes the contents of the `SUNWndps` and `SUNWndpsc` packages:

TABLE 1-1 `SUNWndps` and `SUNWndpsc` Package Contents

Directory	Contents
`/opt/SUNWndps/bsp`	Contains header files and low-level Sun UltraSPARC T1 and Sun UltraSPARC T2 platform initialization and management code.
`/opt/SUNWndps/lib`	Contains system-level libraries, such as CLI, IPC, and LDoms/LDC (Logical Domain Channel).
`/opt/SUNWndps/src`	Contains `ipfwd`, `remotecli`, `rlp`, and `PacketClassifier` reference applications, and network device driver interface header definitions.
`/opt/SUNWndps/tools`	Contains the compiler and runtime system.
`/opt/SUNWndpsc/lib`	Contains the Netra DPS Sun UltraSPARC T2 cryptograpghy driver.
`/opt/SUNWndpsc/src`	Contains Netra DPS Crypto API, IPsec reference application source, and libraries.

TABLE 1-2 describes the contents of the SUNWndpsd package:

TABLE 1-2 SUNWndpsd Package Contents

Directory	Contents
/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

To support Netra Data Plane Software Suite 2.0, use the appropriate firmware installed. See *Netra Data Plane Software Suite 2.0 Release Notes* for the latest information in using the correct combination of firmware and software.

▼ To Check Your OpenBoot PROM Firmware Version

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

  See the following four examples for each system supported:

```
ok showhost
Netra CP3260, No Keyboard
Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
OpenBoot 4.27.8, 16256 MB memory available, Serial #93062640.
Ethernet address 0:14:4f:8c:5:f0, Host ID: 858c05f0.

sc> showhost
 Sun System Firmware 7.0.7.c 2007/11/26 07:18

Host flash versions:
   Hypervisor 1.5.4 2007/10/29 20:27
   OBP 4.27.8 2007/11/15 07:09
   POST 4.27.7 2007/10/24 08:5
```

```
ok showhost
SPARC Enterprise T5120, No Keyboard Copyright 2007 Sun Microsystems, Inc.
All rights reserved.
OpenBoot 4.27.0, 32640 MB memory available, Serial #75404926.
Ethernet address 0:14:4f:7e:96:7e, Host ID: 847e967e.
```

```
ok showhost
Sun Fire T2000, No Keyboard
Copyright 2007 Sun Microsystems, Inc. All rights reserved.
OpenBoot 4.27.0, 8064 MB memory available, Serial #64545116.
Ethernet address 0:3:ba:d8:e1:5c, Host ID: 83d8e15c.
```

```
ok showhost
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 showhost
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.
```

Package Dependencies

The package software has the following dependencies:

- The `SUNWndps` package depends on Sun Studio 12, Java™ version 1.6.0 and `gmake`. You must install these packages before applications are built.
- The `SUNWndpsc` crypto package requires the `SUNWndps` base package.
- 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 – The `SUNWndpsd` software package located in the `Netra_Data_Plane_Software_Suite_2.0.zip` file is not installed on the development system. See "Interprocess Communication Software" on page 79 for details on using this package in the LDoms environment.

Note – If you have previously installed an older version of the Netra Data Plane Software Suite 2.0, remove it before installing the new version. See "To Remove the Software" on page 8.

▼ To Install the Software Into the Default Directory

1. **After downloading the Netra Data Plane Software Suite 2.0 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_2.0.zip
```

3. **Install the `SUNWndps` package. Type:**

```
# /usr/sbin/pkgadd . SUNWndps
```

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

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

Use `Netra_Data_Plane_Software_Suite_Crypto_2.0.zip` for `crypto` drivers. For information on Netra DPS regarding the `crypto` package, see Support Services at: `http://www.sun.com/service/online/`

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

1. **After downloading the Netra Data Plane Software Suite 2.0 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_2.0.zip
```

3. **Add the `SUNWndps` package to *your_directory*. Type:**

```
# pkgadd -d 'pwd' -R your_directory SUNWndps
```

The software is installed in *your_directory*.

4. **Open the *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= your_directory/opt/SUNWndps/tools
```

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

▼ To Remove the Software

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

```
# /usr/sbin/pkgrm SUNWndps SUNWndpsc
```

The Netra Data Plane Software Suite 2.0 is removed.

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

Building and Booting Reference Applications

You need to add the compiler path to your `.cshrc` file before continuing with build instructions.

`.cshrc` File and Required Compiler Path

All the application build scripts are C shell scripts, which do not inherit the environment from where they are invoked. These scripts use the compiler whose path is defined in your `.cshrc` file.

`SUNWndps 2.0` requires Sun Studio 12. Ensure that the correct `PATH` is set and Sun Studio 12 binaries are used for Netra DPS application compilation.

The Netra DPS application build scripts use `csh`, therefore, the user `.cshrc` file must contain the correct path setting for Sun Studio 12. If the user path points to an older cc compiler, the build script exits with a message such as the following:

```
$ pwd
/opt/SUNWndps/src/apps/rlp

$ ./build_10g_niu
cc is a tracked alias for /opt/SUNWspro/bin/cc
cc version 5.8 is less than 5.9
Please install Sun Studio 12
```

Building Reference Application Instructions

The instructions for building reference applications are located in the application directories.

TABLE 1-3 lists the directories and instructional file.

TABLE 1-3 Reference Application Instruction Files

Reference Applications	Building Instruction Location
`ipfwd`	`/SUNWndps/src/apps/ipfwd/README`
`remotecli`	`/SUNWndps/src/apps/remotecli/README.remotecli`
`udp`	`/SUNWndps/src/apps/udp/README`
`ipsec`	`/SUNWndpsc/src/apps/ipsec-gw-nxge/README`
`Teja(R) Tutorial`	`/SUNWndps/tools/examples/PacketClassifier/README`

The application image is booted over the network. Ensure that 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 1-4 describes the optional parameters.

TABLE 1-4 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.
router_ip	The IP address of a router between the client and server.

TABLE 1-4 Boot Optional Parameters *(Continued)*

Option	Description
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 the end of the list.

▼ To Boot an Application Image

1. **Copy the application image to the `tftpboot` directory of the boot server.**
2. **At 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 Netra DPS, 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 Netra DPS attains maximum performance in the target mapping.

When writing the application, you must do the following:

- Be aware of the multiple threads of the application.
- Protect critical regions of the code by using mutual exclusion primitives.
- Communicate structured data using polled queues or event-driven channels.
- 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 issue is where to break the original processing loop into stages. This choice is dictated by the following 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. The following two methods are available in reusing 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 that code changes are minimized.
- Substitute system and application calls with calls to the Netra DPS 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 33 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`.

`tejacc` Compiler Options

TABLE 1-5 lists options to `tejacc`.

TABLE 1-5 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 1-6 describes some of the configuration options of tejacc.

TABLE 1-6 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 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 Netra DPS Interaction

FIGURE 1-1 shows the interaction of tejacc with the other platform components of Netra DPS.

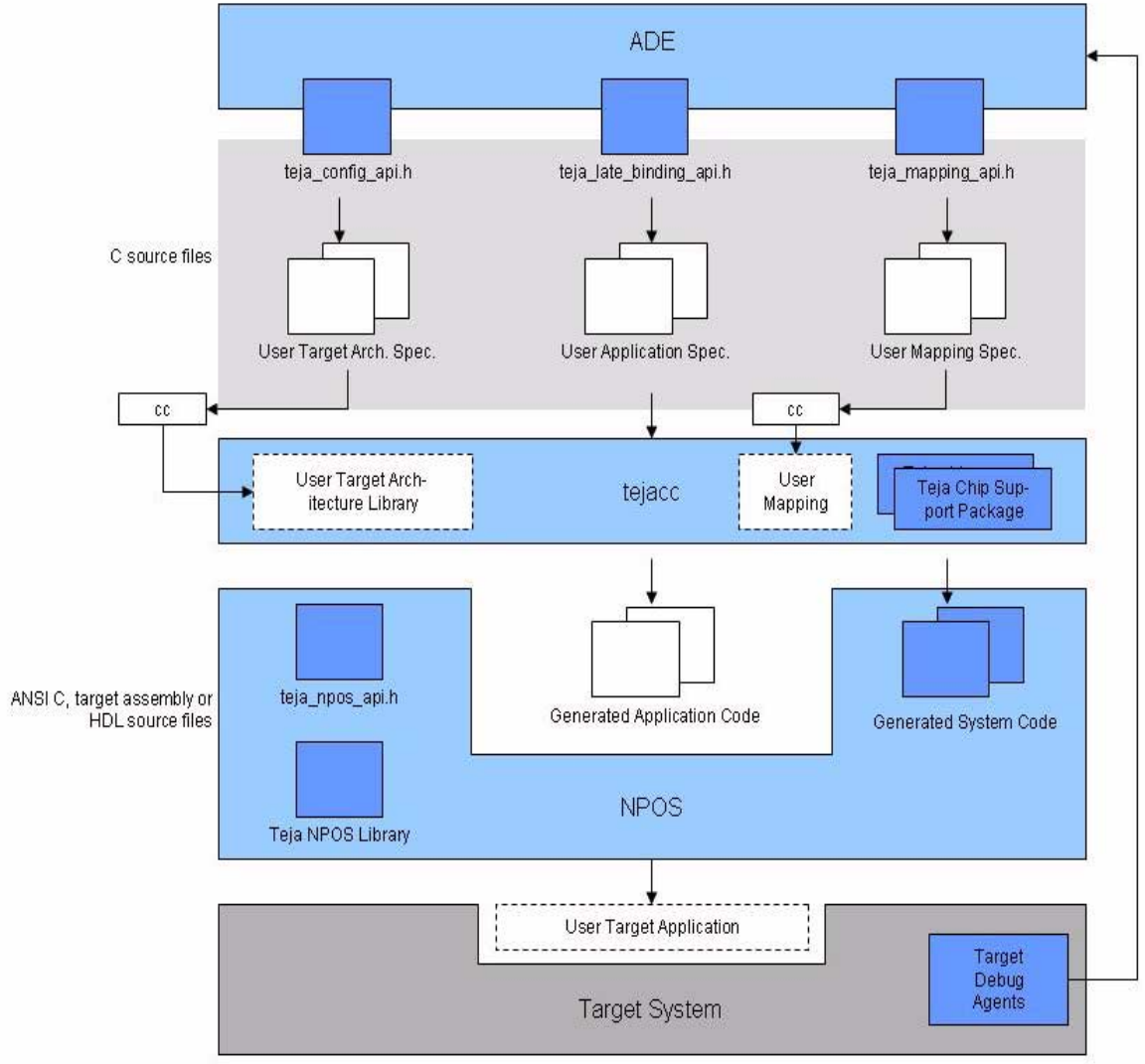


FIGURE 1-1 Teja 4.0 Overview Diagram

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 Netra DPS Runtime 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 Netra DPS Runtime API, and calls to this API pass to the generated code where the calls are either macro expanded (if defined in the Netra DPS Runtime library include file) or linked to the target-specific Netra DPS Runtime library.

Netra DPS also provides a graphical application development environment (ADE) to visualize and manipulate applications. A description of the ADE is not within the scope of this document.

Architecture Elements

Hardware Architecture API 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 Netra DPS 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 1-7 describes the basic hardware architecture elements.

TABLE 1-7 Basic Hardware Architecture Elements

Element	Description
Hardware architecture	A hardware architecture is a container of architecture elements. A hardware architecture has a user-defined name that must be unique in its container, and a type that 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 create a user-defined architecture 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 that provides it a name and type.
Memory	A memory is a target for mapping program variables. A memory is contained in an architecture that provides a name and type.
Hardware object	A hardware object is a logic block that is either known to `tejacc` or is a target for user-defined hardware logic. A hardware object is contained in an architecture that 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 that provides it a name and type, and indicates whether the bus is exported. That is, the bus is visible outside of the containing architecture.

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 2.0 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 the elements properties using the `teja_type_set_property()` and `teja_type_get_property()` functions.

TABLE 1-8 describes advanced hardware architecture elements.

TABLE 1-8 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 architecture specification. A port is also a handle on an architecture for connecting to another port. A port is contained in an architecture that 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, the port 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 2.0 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 2.0 Reference Manual*.

TABLE 1-8 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 that 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 API 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 the Late-Binding API which 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 the following:

- 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 to and from 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 2.0 Reference Manual* for a full list of 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 1-9 describes the Late-Binding elements.

TABLE 1-9 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 the following: • `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 the following: • `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

TABLE 1-9 Late-Binding Elements *(Continued)*

Late-Binding Element	Description
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 `free()` and `malloc()` calls. Sometimes the application needs to match accesses to two memory pools. Given a buffer from one memory pool, obtain the memory pool 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 the following: • `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.
Channel	The Channel API is used to establish connections among threads, to inspect connection states, 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.You can 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 1-10 describes other elements.

TABLE 1-10 Other Elements

Other Element	Description
Operating system	An operating system runs on processors and is a target for running processes. An operating system 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 2.0 Reference Manual*.

User API Overview

This section gives an overview of the Netra DPS API for writing the user application files in the source sets given to `tejacc`. This API is executed on the target and it is composed of three sets of functions:

Late-Binding API Overview

The Late-Binding API is described in "Software Architecture and Late-Binding API Overview" on page 20. This 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. The *Netra Data Plane Software Suite 2.0 Reference Manual* contains API function information.

Netra DPS Runtime API Overview

The Netra DPS Runtime 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, Netra DPS Runtime APIs are not treated by the compiler and are implemented in precompiled libraries.

The memory management functions offer `teja_malloc` and `teja_free` functionality. These functions are computation expensive and should only be used in initialization code or nonrelative critical code. On bare hardware targets, the `teja_free()` function is an empty operation, so only `teja_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 when the registered signal is a valid OS signal code, when the application receives an actual OS signal, the handler is still called synchronously. If a Teja process running multiple threads receives an OS signal, every one of its threads 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 reacts 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 the 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 Netra DPS Runtime API is provided in the *Netra Data Plane Software Suite 2.0 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 the Netra DPS advance development environment (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 2.0 Reference Manual*. The API includes facilities to do the following:

- 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 source files to the target architecture. TABLE 1-11 describes these relationships.

TABLE 1-11 Mapping of Elements

Elements	Mapping
Functions	Mapped to threads with the `teja_map_function_to_thread()` function.
Variables	Mapped to memories or process address spaces with the `teja_map_variable_to_memory()` and `teja_map_variables_to_memory()` functions.
Processors	Initialized with the `teja_map_initialization_function_to_processor()` function.
Mapping-specific properties	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, so options are 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, so options are 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, so options are 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 source set consisting of one or more source files. The source set 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 21. These functions are generated from context 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, including 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 31.

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

   - For channel, mutex, queue, and memory pool functions, ensure that 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 parameters specifying a time quantity 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:

- 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

Profiler

This chapter discusses the Netra DPS profiler used in the Netra Data Plane software. Topics include:

Profiler Introduction

The Netra DPS 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. The 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 profiler with the `-pg` command-line option (`tejacc`). You can insert the API calls at desired places to start collecting profiling data. The profiler configures and sets the hardware resources to capture the requested data. At the same time, the 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 API chapter of the *Netra Data Plane Software Suite 2.0 Reference Manual* for the different groups and different events that are captured and measured on the target.

Profiler Output

The profiler output consists of one line per profiler record. Each line commonly has a format of nine comma-delimited fields. The fields contain values in hexadecimal. If a record is prefixed with a `-1`, 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 API chapter of the *Netra Data Plane Software Suite 2.0 Reference Manual*, and run the application again.

TABLE 3-1 describes the fields of the profiler record:

TABLE 3-1 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 3-1 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 API chapter in the *Netra Data Plane Software Suite 2.0 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 the following: • `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.
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 "Profiler Output Example" on page 40 for an example of dump output.

Profiler Examples

For profiler API function descriptions, refer to the *Netra Data Plane Software Suite 2.0 Reference Manual.*

Profiler API

CODE EXAMPLE 3-1 provides an example of profiler API usage.

CODE EXAMPLE 3-1 Sample Profiler API Usage

```
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 profiler properties that you can change 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`.

Profiler Output Example

The following is an example of the profiler output.

```
TEJA_PROFILE_DUMP_START,ver2.0
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, `ver2.0`, is the profiler 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.

Profiling Application Performance

Profiling consists of instrumenting your application to extract performance information that can be used to analyze, diagnose, and tune your application. Netra DPS 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 for the CPU, DRAM controllers, and JBus are described in TABLE 3-2, TABLE 3-3, and TABLE 3-4 respectively.

TABLE 3-2 Sun UltraSPARC T1 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 3-3 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 3-4 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 behavior is in a steady state. To check this behavior,

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 3-1. 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 where 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 Netra DPS 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 number 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 terms of cycles of contribution to overall CPI. This metric also enables you to estimate the memory bandwidth requirements.

Using the Profiler Script

The profiler script is used to summarize the profiling output generated from the profiler. The profiler script (written in `perl`) converts the raw profiler output to a summarized format that is easy to read and interpret.

Profiler Scripts

Two scripts are available, `profiler.pl` and `profiler_n2.pl`. `profiler.pl` is used for parsing outputs generated from a Sun UltraSPARC T1 (CMT1) processor. `profile_n2.pl` is used for parsing outputs generated from a Sun UltraSPARC T2 (CMT2) processor.

Usage

For Sun UltraSPARC T1 platforms (such as a Sun Fire T2000 system):

```
profiler.pl input_file > output_file
```

For Sun UltraSPARC T2 platforms (such as a Sun SPARC Enterprise T5220 system):

```
profiler_n2.pl input_file > output_file
```

input_file

This file consists of raw profile data generated by the Netra DPS profiler. Typically, this data is captured on the console and saved into a file with `.csv` suffix, indicating that this is a CSV (comma-separated values) file. For example, *input_file*`.csv`

output_file

This file is generated by redirecting the outputs of the `profiler.pl` script to an output file. This file should also be in CSV format. For example, *output_file*`.csv`.

Note – If there is no redirection (that is, the *output_file* is not specified), the output of the script will display on the console.

Raw Profile Data

Raw profile data is the direct output from the profiler.

The following shows an example of the raw profile data output from a Sun UltraSPARC T1 processor:

```
TEJA_PROFILE_DUMP_START,ver1.1
CPUID,ID,Type,Cycles,PC,Grp,Evt_Hi,Evt_Lo,Overflow,User Data
4,18236,1,4cf2eb9ce4,521f08,1,100,1
4,3a2f,2,4d048acb40,5128f0,1,31cffa4,c2a,0,1b7740,3da594c
4,18236,1,4d048ad5c4,521f08,1,100,2
4,3a2f,2,4d162a0db0,5128f0,1,31d274e,0,0,1e8480,3da594c
4,18236,1,4d162a1888,521f08,1,100,4
4,3a2f,2,4d27c951cc,5128f0,1,31d2e36,50e,0,2191c0,3da594c
4,18236,1,4d27c95c28,521f08,1,100,8
4,3a2f,2,4d396893a0,5128f0,1,31d238f,25b863,0,249f00,3da594c
4,18236,1,4d39689dd8,521f08,1,100,10
4,3a2f,2,4d4b07cca0,5128f0,1,31cf8de,0,0,27ac40,3da594c
4,18236,1,4d4b07d708,521f08,1,100,20
4,3a2f,2,4d5ca70e88,5128f0,1,31d183c,0,0,2ab980,3da594c
4,18236,1,4d5ca7194c,521f08,1,100,40
4,3a2f,2,4d6e4654ac,5128f0,1,31d2bd3,1b2,0,2dc6c0,3da594c
4,18236,1,4d6e465ef4,521f08,1,100,80
TEJA_PROFILE_DUMP_END
```

The following shows an example of the raw profile data output from the Sun UltraSPARC T2 processor:

```
TEJA_PROFILE_DUMP_START,ver1.1
CPUID,ID,Type,Cycles,PC,Grp,Evt_Hi,Evt_Lo,Overflow,User Data
2,315,1,d8a403c78c,52cf10,1,12,12
2,21c9,2,d8a403e3b1,514fe8,1,e,e,0,927c0,1d905b
2,4cd,1,d8a403eca2,52cf10,1,22,22
2,21c9,2,d8b8cd3be2,514fe8,1,5e89cc,5e89cc,0,30d40,0
2,4cd,1,d8b8cd3fee,52cf10,1,42,42
2,21c9,2,d8cd9812d0,514fe8,1,0,0,0,30d40,0
2,4cd,1,d8cd98178a,52cf10,1,82,82
2,21c9,2,d8e2636b16,514fe8,1,db21ac,db21ac,0,30d40,0
2,4cd,1,d8e2636f18,52cf10,1,102,102
2,21c9,2,d8f72f1c5c,514fe8,1,46042d,46042d,0,30d40,0
2,4cd,1,d8f72f2058,52cf10,1,202,202
2,21c9,2,d90bfa2d22,514fe8,1,0,0,0,30d40,0
2,4cd,1,d90bfa3181,52cf10,1,402,402
2,21c9,2,d920c5ce6c,514fe8,1,24ea141,24ea141,0,30d40,0
2,4cd,1,d920c5d301,52cf10,1,802,802
2,21c9,2,d93590ffc6,514fe8,1,8fb2c,8fb2c,0,30d40,0
2,4cd,1,d9359103dc,52cf10,1,fd2,fd2
2,21c9,2,d94a5cf7e3,514fe8,1,3f5f51c,3f5f51c,0,30d40,0
2,4cd,1,d94a5cfc19,52cf10,1,13,13
2,21c9,2,d95f283398,514fe8,1,0,0,0,30d40,0
2,4cd,1,d95f28379f,52cf10,1,23,23
2,21c9,2,d973f413a1,514fe8,1,2734a8,2734a8,0,30d40,0
2,4cd,1,d973f417ba,52cf10,1,103,103
2,21c9,2,d988bfbbca,514fe8,1,0,0,0,30d40,0
2,4cd,1,d988bfbfe1,52cf10,1,203,203
2,21c9,2,d99d8be47f,514fe8,1,61aa,61aa,0,30d40,0
2,4cd,1,d99d8be94f,52cf10,1,44,44
2,21c9,2,d9b257ba5a,514fe8,1,0,0,0,30d40,0
2,4cd,1,d9b257be48,52cf10,1,84,84
2,21c9,2,d9c7237ebc,514fe8,1,0,0,0,30d40,0
2,4cd,1,d9c72382f0,52cf10,1,104,104
2,21c9,2,d9dbee7725,514fe8,1,0,0,0,30d40,0
2,4cd,1,d9dbee7b2f,52cf10,1,204,204
2,21c9,2,d9f0b99d84,514fe8,1,0,0,0,30d40,0
2,4cd,1,d9f0b9a1c5,52cf10,1,15,15
2,21c9,2,da05853c14,514fe8,1,0,0,0,30d40,0
2,4cd,1,da05854024,52cf10,1,25,25
2,21c9,2,da1a5067bf,514fe8,1,0,0,0,30d40,0
```

```
2,4cd,1,da1a506bdd,52cf10,1,45,45
2,21c9,2,da2f1c54fd,514fe8,1,300388,300388,0,30d40,0
2,4cd,1,da2f1c5948,52cf10,1,85,85
2,21c9,2,da43e87245,514fe8,1,0,0,0,30d40,0
2,4cd,1,da43e876d0,52cf10,1,105,105
2,21c9,2,da58b3416a,514fe8,1,3d0910,3d0910,0,30d40,0
2,4cd,1,da58b3457e,52cf10,1,205,205
2,21c9,2,da6d7e5a3b,514fe8,1,0,0,0,30d40,0
2,4cd,1,da6d7e5e5d,52cf10,1,16,16
2,21c9,2,da824aa191,514fe8,1,0,0,0,30d40,0
2,4cd,1,da824aa5e5,52cf10,1,26,26
2,21c9,2,da9715c92e,514fe8,1,0,0,0,30d40,0
2,4cd,1,da9715cd85,52cf10,1,46,46
2,21c9,2,daabe167f2,514fe8,1,0,0,0,30d40,0
2,4cd,1,daabe16c18,52cf10,1,86,86
2,21c9,2,dac0ad6c8d,514fe8,1,0,0,0,30d40,0
2,4cd,1,dac0ad7142,52cf10,1,106,106
2,21c9,2,dad5792613,514fe8,1,0,0,0,30d40,0
2,4cd,1,dad5792a2b,52cf10,1,206,206
2,21c9,2,daea449364,514fe8,1,0,0,0,30d40,0
2,4cd,1,daea44979f,52cf10,1,17,17
2,21c9,2,daff0f72f4,514fe8,1,0,0,0,30d40,0
2,4cd,1,daff0f76fd,52cf10,1,27,27
2,21c9,2,db13db2e84,514fe8,1,0,0,0,30d40,0
2,4cd,1,db13db32cc,52cf10,1,47,47
2,21c9,2,db28a68860,514fe8,1,0,0,0,30d40,0
2,4cd,1,db28a68c8d,52cf10,1,87,87
2,21c9,2,db3d7120a0,514fe8,1,0,0,0,30d40,0
2,4cd,1,db3d7125a6,52cf10,1,107,107
2,21c9,2,db523c58b1,514fe8,1,0,0,0,30d40,0
2,4cd,1,db523c5cdf,52cf10,1,207,207
2,21c9,2,db6707bf3f,514fe8,1,0,0,0,30d40,0
2,4cd,1,db6707c3ea,52cf10,1,4b,4b
2,21c9,2,db7bd4202d,514fe8,1,0,0,0,30d40,0
2,4cd,1,db7bd42494,52cf10,1,8b,8b
2,21c9,2,db909fb827,514fe8,1,0,0,0,30d40,0
2,4cd,1,db909fbc6c,52cf10,1,cb,cb
2,21c9,2,dba56a6332,514fe8,1,0,0,0,30d40,0
2,4cd,1,dba56a67dd,52cf10,1,12,12
TEJA_PROFILE_DUMP_END
```

Summarized Profile Data

Summarized profile data is the processed data generated from the `profiler.pl` and the `profile_n2.pl` for the Sun UltraSPARC T1 (CMT1) and (Sun UltraSPARC T2 (CMT2) processors, respectively.

Sun UltraSPARC T1 Processor Profiler Output

For the Sun UltraSPARC T1 processor, the summary displays as in the following example:

```
cpuid , cycle ,  SB_full ,ITLB_miss ,Instr_cnt ,FP_instr_cnt ,DTLB_miss
,IC_miss ,L2_Imiss ,DC_miss ,L2_Dmiss_LD ,userdata1 ,userdata2 ,
4 , 289219777 ,3121, 0, 51104522, 0, 0, 1080, 433, 2471858, 236191, 2600000
,64641356 ,
CPU,StartPC,UpdatePC,Cycles,Instr_cnt,CntrName,Value,UserData.1,UserData.2,
4,0x521f08,0x5128f0,295649212,52240523,FP_instr_cnt,0,400000,64641356,
4,0x521f08,0x5128f0,147824128,26122620,IC_miss,689,600000,64641356,
4,0x521f08,0x5128f0,295647284,52238312,DC_miss,2472263,800000,64641356,
4,0x521f08,0x5128f0,295646420,52234078,ITLB_miss,0,1000000,64641356,
4,0x521f08,0x5128f0,295644896,52241803,DTLB_miss,0,1200000,64641356,
4,0x521f08,0x5128f0,295649084,52246157,L2_Imiss,434,1400000,64641356,
4,0x521f08,0x5128f0,295646316,52250156,L2_Dmiss_LD,236270,1600000,64641356,
4,0x521f08,0x5128f0,295644764,52232100,SB_full,3114,1800000,64641356,
```

TABLE 3-5 describes each field in the top section of the summarized Sun UltraSPARC T1 profile data output:

TABLE 3-5 Sun UltraSPARC T1 Profile Data Output Field Descriptions

Field	Description
`cpuid`	CPU ID found in the first column of the raw profile data. Note: If profiling is done for multiple strands, then multiple rows of summarized data (with different CPU IDs) are shown in the top section.
`cycle`	Average number of clock cycles elapsed per profiling interval.
`SB_full`	Average number of `SB_full` occurrences per profiling interval.
`ITLB_miss`	Average number of `ITLB_miss` occurrences per profiling interval.
`Instr_cnt`	Average number of instructions executed per profiling interval.
`FP_instr_cnt`	Average number of floating point instructions executed per profiling interval.
`DTLB_miss`	Average number of `DTLB_miss` occurrences per profiling interval.

TABLE 3-5 Sun UltraSPARC T1 Profile Data Output Field Descriptions *(Continued)*

Field	Description
`IC_miss`	Average number of `IC_miss` occurrences per profiling interval.
`L2_Imiss`	Average number of `L2_Imiss` occurrences per profiling interval.
`DC_miss`	Average number of `DC_miss` occurrences per profiling interval.
`L2_Dmiss_LD`	Average number of `L2_Dmiss_LD` occurrences per profiling interval.
`UserData.1`	Average number taken from the User Defined Data1 column.
`UserData.2`	Average number taken from the User Defined Data2 column.

Sun UltraSPARC T2 Processor Profiler Output

For the Sun UltraSPARC T2 processor, the summary displays as in the following example:

```
CPUid, cycles, Store_instr,  L2_instr_misses,
ITLB_miss_L2,  CPU_ST_to_PCX,  MA_OP,  MA_Busy,
Completed_branches,  Icache_misses,  Stream_LD_to_PCX,  DES_3DES_OP,
DES_3DES_Busy_cycle,  Sethi_instr,  L2_load_misses,  DTLB_miss_L2,
MMU_LD_to_PCX,  CRC_TCPIP_Cksum_OP,  CRC_MPA_Cksum,  Taken_branches,
Dcache_misses,  Stream_ST_to_PCX,  AES_OP,  AES_Busy_cycle,
Other_instr,  FGU_arithmatic_instr,  ITLB_ref_L2,  CPU_LD_to_PCX,
RC4_OP,  RC4_Busy_cycle,  ITLB_miss,  Atomics,
Load_instr,  DTLB_ref_L2,  CPU_Ifetch_to_PCX,  MD5_SHA1_SHA256_OP,
MD5_SHA1_SHA256_Busy_cycle,  DTLB_miss,  TLB_miss,  All_instr,
Userdata.1,Userdata.2,

17, 347989526,  3185726,  78,
0,  3000015,  0,  0,
5023983,  113,  0,  0,
0,  0,  216952,  0,
0,  0,  0,  6393524,
2050737,  0,  0,  0,
48603479,  0,  0,  2636500,
0,  0,  0,  184283,
13328505,  0,  150,  0,
0,  0,  0,  74964356,
210256,  1032899,

17
347989526
3185726
78
0
3000015
0
0
5023983
113
0
0
0
0
216952
0
0
0
0
```

```
6393524
2050737
0
0
0
48603479
0
0
2636500
0
0
0
184283
13328505
0
150
0
0
0
0
74964356
210256
1032899
```

Note – The data in the second and third sections of the Sun UltraSPARC T2 summary are identical. The format of the first section is the field header. The format in the second section matches the layout of the field header. The format in the third section is in one single column. This layout enables you to easily transfer data to a spreadsheet file column.

TABLE 3-6 describes each field in the top section of the summarized Sun UltraSPARC T2 profile data output:

TABLE 3-6 Sun UltraSPARC T2 Profile Data Output Field Descriptions

Field	Description
`CPUid`	CPU ID found in the first column of the raw profile data. Note: If profiling is done for multiple strands, then multiple rows of summarized data (with different CPU IDs) are shown in the top section.
`cycles`	Average number of clock cycles elapsed per profiling interval.
`Store_instr`	Number of Store Instructions executed per profiling interval.

TABLE 3-6 Sun UltraSPARC T2 Profile Data Output Field Descriptions *(Continued)*

Field	Description
L2_instr_misses	Number of L2 cache instruction misses per profiling interval.
ITLB_miss_L2	Average number of ITLB_miss occurrences per profiling interval.
CPU_ST_to_PCX	Number of CPU stores to PCX (Processor to Cache) per profiling interval.
MA_OP	Number of MA operations executed per profiling interval.
MA_Busy	Number of busy encountered per profiling interval when attempted to execute the MA operation.
Completed_branches	Number of completed branches per profiling interval.
Icache_misses	Number of Instruction Cache misses per profiling interval.
Stream_LD_to_PCX	Number of Loads to PCX (Processor to Cache) per profiling interval
DES_3DES_OP	Number of DES_3DES operations per profiling interval.
DES_3DES_Busy_cycle	Number of busy cycles encountered per profiling interval when attempted to execute the DES_3DES operations.
Sethi_instr	Number of Sethi instructions executed per profiling interval.
L2_load_misses	Number of L2 cache load misses per profiling interval.
DTLB_miss_L2	Number of Data TLB misses for L2 cache per profiling interval.
MMU_LD_to_PCX	Number of MMU load to PCX (Processor to Cache) per profiling interval.
CRC_TCPIP_Cksum_OP	Number of CRC, TCPIP_Cksum operations per profiling interval.
CRC_MPA_Cksum	Number of CRC and MPA checksum operations per profiling interval.
Taken_branches	Number of branches taken per profiling interval.
Dcache_misses	Number of L1 Data Cache misses per profiling interval.
Stream_ST_to_PCX	Number of stream store operations to PCX (Processor to Cache) per profiling interval.
AES_OP	Number of AES operations per profiling interval.
AES_Busy_cycle	Number of busy cycles encountered per profiling interval when attempted to execute the AES operation.
Other_instr	Number of all other instructions executed per profiling interval.
FGU_arithmatic_instr	Number of Floating-point arithmetic instructions executed per profiling interval.
ITLB_ref_L2	Number of Instruction TLB referenced for L2 cache per profiling interval.

TABLE 3-6 Sun UltraSPARC T2 Profile Data Output Field Descriptions *(Continued)*

Field	Description
CPU_LD_to_PCX	Number of CPU load to PCX (Processor to Cache) per profiling interval.
RC4_OP	Number of RC4 operations executed per profiling interval.
RC4_Busy_cycle	Number of busy cycles encountered per profiling interval when attempted to execute the RC4 operation.
ITLB_miss	Number of Instruction TLB misses (for L1 cache) per profiling interval.
Atomics	Number of atomic operations executed per profiling interval.
Load_instr	Number of Load instructions executed per profiling interval.
DTLB_ref_L2	Number of Data TLB referenced for L2 cache per profiling interval.
CPU_Ifetch_to_PCX	Number of CPU Instruction fetches to PCX (Processor to Cache) per profiling interval.
MD5_SHA1_SHA256_OP	Number of MD5_SHA1_SHA256 operations executed per profiling interval.
MD5_SHA1_SHA256_Busy_cycle	Number of busy cycles encountered per profiling interval when attempted to execute the MD5_SHA1_SHA256 operation.
DTLB_miss	Number of Data TLB misses (for L1 cache) per profiling interval.
TLB_miss	Number of TLB (for L1 cache) misses per profiling interval.
All_instr	Total number of instructions executed per profiling interval.
Userdata.1	Average number taken from the User Defined Data1 column.
Userdata.2	Average number taken from the User Defined Data2 column.

Performance Parameters Calculations

You can use the output values of the summarized data to derive various important performance parameters. This section lists performance parameters and the method from which they are derived.

- Key for this section:
- Division: /
- Multiplication: *
- pkts_per_interval = Number of packets per interval (for example, 200000)

  This can be obtained from the Userdata.1 field.
- cpu_frequency = CPU frequency in Hz (for example, 1200000000 for T2000 system)

Sun UltraSPARC T1 Processor

Instructions per Packet:

Average number of instructions executed in a packet.

Formula: value = (`Instr_cnt` / `pkts_per_interval`)

Instructions per Cycle (IPC):

Average number of instructions executed per cycle.

Formula: value = (`Instr_cnt` / cycle)

Packet Rate:

Average number of packets executed per second (in Kilo-packets per second).

Formula: value = ((`pkts_per_interval` / (cycle / `cpu_frequency`)) / 1000)

`SB_full` **per thousand instructions:**

Average number of SB_full occurrences per 1000 instructions executed.

Formula: value = ((`SB_full` / `Instr_cnt`) * 1000)

`FP_instr_cnt` **per thousand instructions:**

Average number of `FP_instr_cnt` occurrences per 1000 instructions executed.

Formula: value = ((`FP_Instr_cnt` / `Instr_cnt`) * 1000)

`IC_miss` **per thousand instructions:**

Average number of `IC_miss` occurrences per 1000 instructions executed.

Formula: value = ((`IC_miss` / `Instr_cnt`) * 1000)

`DC_miss` **per thousand instructions:**

Average number of `DC_miss` occurrences per 1000 instructions executed.

Formula: value = ((`DC_miss` / `Instr_cnt`) * 1000)

`ITLB_miss` **per thousand instructions:**

Average number of `ITLB_miss` occurrences per 1000 instructions executed.

Formula: value = ((`ITLB_miss` / `Instr_cnt`) * 1000)

`DTLB_miss` **per thousand instructions:**

Average number of `DTLB_miss` occurrences per 1000 instructions executed.

Formula: value = ((`DTLB_miss` / `Instr_cnt`) * 1000)

`L2_imiss` **per thousand instructions:**

Average number of `L2_miss` occurrences per 1000 instructions executed.

Formula: value = ((`L2_miss` / `Instr_cnt`) * 1000)

`L2_dmiss_LD` **per thousand instructions:**

Average number of `L2_Dmiss_LD` occurrences per 1000 instructions executed.

Formula: value = ((`L2_miss` / `Instr_cnt`) * 1000)

Sun UltraSPARC T2 Processor

Instruction per Packet:

Average number of instructions executed in a packet.

Formula: value = (`All_instr` / `pkts_per_interval`)

Instructions per Cycle (IPC):

Average number of instructions executed per cycle.

Formula: value = (`All_instr` / cycle)

Note – The Sun UltraSPARC T2 processor has two pipelines in each core. The maximum IPC number of each pipeline is 1. Therefore, the maximum IPC number of each core is 2. Pipeline utilization is this number of each pipeline multiplied by 100%. For example, if the IPC is 0.8, then the pipeline utilization of that pipeline is 80%.

Store Instructions per Packet:

Average number of Store instructions executed per packet.

Formula: value = (`Store_instr` / `pkts_per_interval`)

Load Instructions per Packet:

Average number of Load instructions executed per packet.

Formula: value = (`Load_instr` / `pkts_per_interval`)

L2 Load misses per Packet:

Average number of L2 cache Load misses per packet.

Formula: value = (`L2_load_misses` / `pkts_per_interval`)

Icache misses per 1000 Packets:

Average number of L1 `Icache` misses per 1000 packet.

Formula: value = (`Icache_misses` x 1000) / `pkts_per_interval`)

Dcache misses per Packet:

Average number of L1 `Icache` misses per packet.

Formula: value = (`Dcache_misses` / `pkts_per_interval`)

Packet Rate:

Average number of packets executed per second (in Kilo-packets per second).

Formula: value = ((pkts_per_interval / (cycle / cpu_frequency)) / 1000)

Note – Not all possible parameters are shown here. You can derive any parameter with any formula using the data outputs from the summary.

Note – These formulas can easily be inserted into a spreadsheet program.

▼ To Use a Spreadsheet For Performance Analysis

1. **Open the summary file.**
   For example, an *output_file*`.csv` generated by `profiler.pl`.
2. **Insert formulas into the spreadsheet.**
3. **Save the spreadsheet for future reference.**

CHAPTER 4

Debugger

This chapter describes the Netra DPS native debugger and GNU debugger (GDB). Topics include:

Debugger Introduction

The Netra DPS native debugger is the default debugger and is useful for debugging during development. This debugger also identifies system hangs or crashes in the field deployment. To access the Netra DPS native debugger, press Ctrl-C.

To use the GNU Debugger (GDB), you must have your own source code and the binary. You need to turn on the flag for this application, for example,

```
USR_CFLAGS = -DTEJA_DEBUGGER_MODE=TEJA_DEBUGGER_GDB_MODE
```

See "GNU Project Debugger (GDB) Showcase Application" on page 70 for detailed setup and example information.

Native Debugger

The native debugger runs on the target and enables users to do the following:

- 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 `dbghelper.pl`. See "Resolving Symbols" on page 68.

The native debugger is denoted by `dbg`. See "Native Debugger Commands" on page 61.

Debugging Configuration Code

As seen in "tejacc Compiler Configuration" on page 15, `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 can then 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 native 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 the Ctrl-C function to work. If the thread calling the `teja_check_ctrl_c()` function crashes or goes into a deadlock, the Ctrl-C key sequence stops.

Native 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 nonoptimized mode and means that 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 `dbghelper` script. See "Resolving Symbols" on page 68.

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. The breakpoint 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 – You can only use the `step` command 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 the following:
  - `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 `dbghelper.pl` script is used to resolve symbols to set breakpoints in the correct places. The script is located in *install_dir*`/tools/bin` directory, where *install_dir* is the `SUNWndps` package installation directory. For example, `/opt/SUNWndps/tools/bin/dbghelper.pl`.

`-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
```

GNU Project Debugger (GDB) Showcase Application

GDB, the GNU Project debugger, enables you to debug your program in C source code level. The following sections describe the reference Netra DPS application that gives a showcase of the GDB support in Netra DPS over the Logical Domain Channel (LDC).

Configuring LDoms for GDB Showcase

Configuring LDoms for GDB showcase requires the latest LDoms release. If not installed, download and follow the LDoms latest release instructions. Find the latest release of LDoms at: http://www.sun.com/download/index.jsp?tab=2#L)

Use Logical Domains 1.0.1 or later release from the Sun Download Center.

▼ To Configure LDoms Required to Run the GDB Demo

1. **Add the following on the primary domain:**

```
/opt/SUNWldm/bin/ldm remove-mau 8 primary
/opt/SUNWldm/bin/ldm remove-vcpu 28 primary
/opt/SUNWldm/bin/ldm remove-mem 31G primary

/opt/SUNWldm/bin/ldm add-vdiskserver primary-vds0 primary
/opt/SUNWldm/bin/ldm add-vconscon port-range=5000-5100 primary-vcc0 primary
/opt/SUNWldm/bin/ldm add-vswitch net-dev=e1000g0 primary-vsw0 primary
/opt/SUNWldm/bin/ldm list-config
/opt/SUNWldm/bin/ldm add-config remotecli-config
```

2. **Reboot the system for** `remotecli-config` **to take effect.**

3. **Configure LDoms running Netra DPS as follows:**

```
/opt/SUNWldm/bin/ldm add-domain ldg1
/opt/SUNWldm/bin/ldm add-vcpu 16 ldg1
/opt/SUNWldm/bin/ldm add-mem 4G ldg1
/opt/SUNWldm/bin/ldm add-vnet vnet1 primary-vsw0 ldg1
/opt/SUNWldm/bin/ldm add-vdsdev /export/bootdisk1 vol1@primary-vds0
/opt/SUNWldm/bin/ldm add-vdisk vdisk1 vol1@primary-vds0 ldg1
/opt/SUNWldm/bin/ldm add-vdpcs ndps-cli ldg1
/opt/SUNWldm/bin/ldm bind ldg1
/usr/openwin/bin/xterm -sb -sl 5000 -T "Console: ldg1" -geometry 80x12+98+193 -
bg black -fg white
-e /usr/bin/telnet localhost 5000 &
/opt/SUNWldm/bin/ldm start ldg1
```

4. **Configure LDoms running Solaris as follows:**

```
/opt/SUNWldm/bin/ldm add-domain ldg2
/opt/SUNWldm/bin/ldm add-vcpu 12 ldg2
/opt/SUNWldm/bin/ldm add-mem 8G ldg2
/opt/SUNWldm/bin/ldm add-vnet vnet2 primary-vsw0 ldg2
/opt/SUNWldm/bin/ldm add-vdsdev /export/bootdisk2 vol2@primary-vds0
/opt/SUNWldm/bin/ldm add-vdisk vdisk2 vol2@primary-vds0 ldg2
/opt/SUNWldm/bin/ldm add-vdpcc solaris-cli ndps-cli ldg2
/opt/SUNWldm/bin/ldm bind ldg2
/usr/openwin/bin/xterm -sb -sl 5000 -T "Console: ldg2" -geometry 80x12+98+193 -
bg black -fg cyan -e /usr/bin/telnet localhost 5001 &
/opt/SUNWldm/bin/ldm start ldg2
```

▼ To Compile the GDB Showcase

- **From the** `SUNWndps` **package, compile the application under** `/src/apps/gdb`**. Type:**

```
gmake clean
gmake CMT=N1 (for UltraSPARC T1 based platforms)
gmake CMT=N2  (for UltraSPARC T2 based platforms)
cd code/main
gmake
```

This action generates the binary file called `main` under `src/apps/gdb/code/main`. The required Solaris utility binaries are under `src/apps/gdb/solaris-gw/ldc_so`.

▼ To Configure the Solaris Domain for GDB

Once the LDoms domains are configured and running, do the following steps to configure the gateway for GDB in the Solaris domain.

1. **Copy the following files to your Solaris LDoms domain under** `src/apps/gdb/solaris-gw/ldc_so/`.

   To do this, copy the drivers to the location as shown in this example:

```
cp src/apps/gdb/solaris-gw/ldc_so/remldc.conf /kernel/drv/
cp src/apps/gdb/solaris-gw/ldc_so/remldc /kernel/drv/sparcv9
```

   - For `ldc` driver for GDB: `remldc` `remldc.conf`
   - For `gdb` gateway: `remotegw` `remotegw.xml`

2. **Load the drivers as shown below:**

```
rem_drv remldc
add_drv remldc
```

3. **Copy the configuration utility file to the following locations as shown below:**

```
cp -f src/apps/gdb/solaris-gw/ldc_so/remotegw /usr/bin
chmod ugo+xr /usr/bin/remotegw
echo "remotegw    34980/tcp" >> /etc/services
cp -f src/apps/gdb/solaris-gw/ldc_so/remotegw.xml
/var/svc/manifest/network/remotegw.xml
svccfg import /var/svc/manifest/network/remotegw.xml
svcadm enable svc:/network/remotegw:remotegw
```

▼ To Load the GDB Showcase Binary in the Netra DPS Domain

1. **Verify that you have copied your GDB Netra DPS binary** `main` **into your install server under** `/tftpboot`**.**

2. **Type the following at the Netra DPS domain OpenBoot Prom** `ok` **prompt:**

```
ok boot /virtual-devices@100/channel-devices@200/network@0:,main
```

▼ To Run the GDB Command

See "To Get GDB" on page 74 for getting the GDB program.

- **Once your GDB showcase application is compiled, do the following under any host machine as long as you can access your code base** `src/apps/gdb/code/main`**:.**

```
cd src/apps/gdb/code/main
gdb main
```

`main` is the same binary code that you loaded into your Netra DPS domain. The GDB debugger then displays the following:

```
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=sparc-sun-solaris2.10 --target=sparc64-
elf"...
(gdb)
```

You can acquire the host name or IP address for the Solaris domain by typing:

```
ifconfig -a
do: (assuming your ip is 10.7.56.249)

(gdb) target remote tcp:10.7.56.249:34980
```

This connects to your Netra DPS GDB application. The Netra DPS console then displays:

```
LDC Status = UP
calling set_debug_traps()....
Program started: initial breakpoint reached
```

This indicates that the GDB showcase application reached the initial breakpoint artificially created by the application. You can then use the following GDB commands to investigate your application.

GDB commands include the following:

- `target remote tcp:10.7.56.194:34980` – Connects to remote Netra DPS target
- `info thread` – Display threads
- `thread #` – Switch thread
- `info reg` – Show the register files
- `info break` – Show the breakpoint
- `b #` – Set breakpoint
- `d #` – Clear breakpoint
- `c` – Continue
- `s` – Step
- `x` – Check memory location
- `p` – Display variable
- `list` – Display source code, for example, list `debug_func`
- `bt` – Backtrace
- `detach` – To end the remote communication, allow the Solaris gateway program to exit. See "To Create a GDB Showcase" on page 75 for more details.

For additional GDB information and instructions, see *GDB: The GNU Project Debugger* at `http://sourceware.org/gdb/`.

▼ To Get GDB

1. **Download the GDB source code from:**
   `http://sourceware.org/gdb/download`

GDB version 6.6 or above is required.

2. **Under the `gdb` directory, type the following:**

```
./configure --target=sparc64-elf
```

The `gdb` binary is `gdb/gdb`. Also, you can use `gdb/gdbtui` which is a text-based user interface that enables you to browse the source code while `gdb` is debugging the target in one single screen.

▼ To Create a GDB Showcase

1. **Go to** `src/apps/gdb/code/main`.

```
cd src/apps/gdb/code/main
```

2. **Load the** `src/apps/gdb/code/main/main` **into your Netra DPS domain.**
3. **Configure the Solaris gateway in the Solaris domain (for example,** `10.7.56.194`**).**
4. **Type the following:**

```
gdb main
```

In this example, the following output is displayed:

```
Remote debugging using tcp:10.7.56.194:34980
0x00540df4 in teja_breakpoint ()
Current language:  auto; currently minimal
```

```
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=sparc-sun-solaris2.10 --target=sparc64-
linux-elf"...
(gdb)  target remote tcp:10.7.56.249:34980
Remote debugging using tcp:10.7.56.249:34980
0x0053f34c in teja_breakpoint ()
Current language:  auto; currently minimal
(gdb) info thread
* 2 Thread 2 (stat_thd)  0x0053f34c in teja_breakpoint ()
 1 Thread 1 (main_thd00)  0x0050c5d8 in main_thread ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:97
(gdb) thread 1
[Switching to thread 1 (Thread 1)]#0  0x0050c5d8 in main_thread ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:97
97         while ( count  < (2))
(gdb) list debug_func
58         int i ;
59         int j ;
60      //      char *tmp; File: src/app/remcon_impl.c Line: 40
61         char * tmp ;
62      //      tmp = "0xdeadbeef"; File: src/app/remcon_impl.c Line: 41
63            tmp  = "0xdeadbeef";
64      //      gdbptr = "0xbaddcafe"; File: src/app/remcon_impl.c Line: 42
65            gdbptr  = "0xbaddcafe";
66      //      i = first_time++; File: src/app/remcon_impl.c Line: 43
67            i  =  first_time ++;
(gdb) b 67
Breakpoint 1 at 0x50c510: file src/apps/gdb/code/main/_src_app_remcon_impl.c,
line 67.
(gdb) p first_time
$1 = 18255275
(gdb) c
Continuing.
Can't send signals to this remote system. SIGSTOP not sent.
```

```
Program received signal SIGSTOP, Stopped (signal).
debug_func ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:67
67             i  =  first_time ++;
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGSTOP not sent.

Program received signal SIGSTOP, Stopped (signal).
0x0050c514 in debug_func ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:67
67             i  =  first_time ++;
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGSTOP not sent.
Program received signal SIGSTOP, Stopped (signal).
debug_func ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:67
67             i  =  first_time ++;
(gdb) p first_time
$4 = 18255276
(gdb) info thread
 2 Thread 2 (stat_thd)  0x00508f04 in hv_ldc_rx_get_state ()
* 1 Thread 1 (main_thd00)  debug_func ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:67
(gdb) s
Can't send signals to this remote system.  SIGSTOP not sent.

Program received signal SIGSTOP, Stopped (signal).
0x0050c534 in debug_func ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:67
67             i  =  first_time ++;
(gdb) s
Can't send signals to this remote system.  SIGSTOP not sent.

Program received signal SIGSTOP, Stopped (signal).
debug_func ()
   at src/apps/gdb/code/main/_src_app_remcon_impl.c:69
69             j  =  i  + ( first_time );
(gdb) detach
Ending remote debugging.
```

5. Once debugging is completed, type **detach**.

```
gdb detach
```

Note – Once completed, always type **detach** on `gdb`. Otherwise, the `/usr/bin/remotegw` process is left in the Solaris domain `10.7.56.249`. Then use the `kill` command to kill the `/usr/bin/remotegw` process before you start again.

---

6. **To start the process again, type the following:**

```
./ldm stop the-ndps-domain
./ldm start the-ndps-domain
```

7. **Reload your binary in the Netra DPS domain.**

   If your source changes, you need to quit the `gdb` and re-enter `gdb`.

CHAPTER 5

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 Netra DPS Lightweight Runtime Environment (LWRTE) and processes in a domain with a control plane operating system. This chapter gives an overview of the programming interfaces, shows how to set up an LDoms environment in which the IPC mechanism can be used, and explains the IPC specific portions of the IP forwarding reference application (see Chapter 9, "Reference Applications" on page 127).

Programming Interfaces Overview

Chapter 5, Interprocess Communication API, of the *Netra Data Plane Software Suite 2.0 Reference Manual* contains a detailed description of all APIs needed to use IPC. The common API can be used in an operating system to connect to an IPC channel, and transmit and receive packets. First, the user must connect to the channel and register a function to receive packets. Once the channel is established this way, the `ipc_tx()` function can be used to transmit. The framework calls the registered callback function when a message is received.

In an Netra DPS application, the programmer is responsible for calling the framework initialization routines for the IPC and LDC frameworks before using IPC, and must ensure that polling happens periodically.

In a Solaris domain, the IPC mechanism can be accessed from either user or kernel space. Before any API can be used, you must install the `SUNWndpsd` package using the `pkgadd` command, and you must add the `tnsm` driver to the system using `add_drv`. Refer to the respective man pages for detailed instructions. From the Solaris kernel, the common APIs mentioned above are used for IPC. In user space, the `tnsm` driver is seen as a character driver. The `open`(), `ioctl`(), `read`(), `write`(), and `close`() interfaces are used to connect to a channel, and send and receive messages.

Configuring the Environment for IPC

This section describes the configuration of the environment needed to use the IPC framework. This section also 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 Netra DPS runtime. This declaration 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. Some basic configuration channels must be defined adhering to the naming convention described in "LDoms Channel Setup" on page 81. 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 set up 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. These commands 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.

Names for data plane channel servers and clients cannot be longer than 48 characters. This limit includes the prefixes of configuration channels.

Note – A Solaris domain may only have one configuration channel. In the configuration domain, where the channel client `tnsm-gc0` is present, a channel client with the name `config-tnsm0` must not be configured.

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. This utility is part of the `SUNWndpds` package and is located in the `bin` directory. `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 5-1. All of these parameters need to be present to set up an IPC channel.

TABLE 5-1 `tnsmctl` Parameters

Parameter	Description
`-S`	Set up 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 for UltraSPARC T1 Based Servers

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

Domains

TABLE 5-1 describes the four environment domains.

TABLE 5-2 Environment Domains

Domain	Description
`primary`	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`	Owns the other PCI bus (`bus_b`) with its two network interfaces and runs an LWRTE application.
`ldg2`	Runs control plane applications and uses IPC channels to communicate with the LWRTE domain (`ldg1`).
`ldg3`	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 11/06 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 4GByte 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 32GByte 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 20 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 selected by the system administrator. 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, you must install the `SUNWndpsd` package 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 -e` 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
        NAME                CLIENT                          LDC
        config-tnsm-ldg2    config-tnsm0@ldg2               6
```

For ldg2:

```
VDPCS
        NAME                CLIENT                          LDC
        config-tnsm-ldg2    config-tnsm0@ldg2               6
```

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 -e), 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.

Example Environment for UltraSPARC T2 Based Servers

The example configuration described in "Example Environment for UltraSPARC T1 Based Servers" on page 84 can be used with UltraSPARC T2 based servers with some minor modifications.

- The LWRTE domain (`ldg1`) must still be core aligned.

  The UltraSPARC T2 chip has eight threads per core, so changing the number of `vcpus` in the primary from four to eight aligns the second domain to a core boundary.

- The UltraSPARC T2 chip does not have two PCI buses.

  In the environment in "Example Environment for UltraSPARC T1 Based Servers" on page 84, the primary domain owned one of the PCI buses (`bus_a`), while the Netra DPS Runtime Environment domain owned the other one (`bus_b`). With a UltraSPARC T2 there is only one PCI bus (`pci`) and the Network Interface Unit (`niu`). To set up an environment on such a system, the NIU should be removed from the primary domain and added to the Netra DPS Runtime Environment domain (`ldg1`).

In addition, the IP forwarding and RLP reference applications use forty threads in the UltraSPARC T2 LDoms configurations, and the Netra DPS Runtime Environment domain must be sized accordingly.

Reference Applications

The Netra DPS package contains an IP forwarding reference application that uses the IPC mechanism. The Netra DPS 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 90). Netra DPS chooses which table to use and where to gather some simple statistics, and displays the statistics in the Solaris domain. The application is designed to operate in the example setup shown in "IPC Channels" on page 86.

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, the command header file 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

The code that implements the forwarding application consists of two components:

- 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 architecture 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 LDoms.

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 initialization, the IPC framework is initialized by a call of tnipc_init(). The previousfour functions must be called in this specific order. The data structures are then initialized for the forwarding tables.

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 for UltraSPARC T1 Based Servers" on page 84. Once the IPC channels are up, you can use the fibctl utility to manipulate the forwarding tables and gather statistics.

CHAPTER 6

Remote Command-Line Interface

This chapter describes the remote command-line-interface (CLI). Topics include:

Remote Command-Line Interface Introduction

The remote command-line interface (CLI) provides you remote access to commands for you to configure and gather the Netra DPS runtime system information (for example, platform). The CLI also provides you remote access to the Netra DPS runtime interactive debugger and a core dump facility.

IPC Setup for Remote CLI

To access the CLI remotely, you must have the interprocess communication (IPC) mechanism set up on your system (see Chapter 5 for IPC information). In the same way that the IPC channel with ID 4 was set up to be used by the IP forward reference application, a channel with ID 1 must be set up for the remote CLI. `SUNWndpsd` must be installed on the Solaris system that will host the remote CLI.

Note – The remote CLI communicates over IPC channel number 1 (one), therefore, IPC channel number 1 should *not* be used for any other purpose.

The applications that use the remote command-line interface must have the following:

- The `cli` type is declared as `"remote"` in hardware configuration file `teja_architecture_set_property(cmt1_chip, "cli_type", "remote");`
- On one of the CPU strands, the Fast Path Manager must be running `lwrte_fastpath_manager_process();USR_LIBS` contains common, LDC and IPC libraries
- `USR_LIBS = /opt/SUNWndps/lib/common/lwrtecmn.o /opt/SUNWndps/lib/ldc/lwrteldc.o /opt/SUNWndps/lib/ipc/lwrteipc.o`

Accessing the Remote CLI

Once IPC channel number 1 is set up between the LWRTE and the remote CLI Solaris host system, you are ready to access the remote CLI.

▼ To access the CLI Console

1. **Connect to the Solaris CLI host system.**

   Use `telnet` to the hosting Solaris system at the default port number 30001.

```
# telnet solarisdomain 30001
Trying 192.168.1.6...
Connected to solarisdomain.
Escape character is '^]'.
ndps>
```

2. **Enter `help` at the prompt, as shown in this example, to list options.**

```
ndps> help
        connect                 : connect to NDPS
        disconnect              : disconnect from NDPS Channel
        send break dbg          : jump into debugger
        send break sys          : jump into system cli
        cont                    : quit from debugger
        c                       : quit from debugger
        coredump [-d <dump dir>] <corename>     : dumps lwrte core
              [-d <dump dir>] dump directory (default: "/var/lwrtedump")
                <corename> core dump file name
        quit                    : quit from system cli
        exit                    : quit this program
        help                    : help for this
        console [-f file]       : connects to runtime console
                                  file is the optional log file
ndps>
```

3. **To connect to the remote CLI, type `connect` at the prompt:**

   Type **`disconnect`**, as shown, to close the channel to the remote CLI.

```
ndps> connect
Opening channel 1
IPC channel #: 1
ndps> disconnect
Closing channel 1
```

4. **To close the connection, type `exit` at the prompt.**

```
ndps> exit
the IPC link is DOWN or CLOSED, please type connect to bring it up again!
Connection to sol closed by foreign host.
#
```

Debugging Remotely

Once connected to the Netra DPS runtime, you can access the Netra DPS debugger.

▼ To Access the Netra DPS Debugger

- **Type the `send break dbg` command:**

```
ndps> connect
Opening channel 1
IPC channel #: 1
ndps> send break dbg
enter NDPS debugger...
dbg>
```

Type **`help`** or **`?`** for help options.

Type **`c`** or **`cont`** to quit the debugger program:

```
dbg> c
exit NDPS debugger...
ndps>
```

Coredump Support

Coredump is supported under the Debugger program (see "Debugging Remotely" on page 96). From the `dbg` mode, use the `coredump` command to dump the LWRTE system core. The `coredump` command has the following format:

```
coredump [-d dump_dir] corename
```

dump_dir is the directory where the core is saved on the CLI hosting Solaris system. By default, the core is saved in `/var/lwrtedump`.

corename is the core file name. The next available numeric is appended to this core file name, followed by `.gz`.

```
dbg> coredump core
Using dump directory "/var/lwrtedump"
Total dumped: 74024954 bytes, compressed to: 456741 bytes
finished coredump successfully!
dbg>
```

The preceding core file is created at `/var/lwrtedump/core-1.gz` on the remote CLI host system (`solarisdomain`).

System Configuration

You can collect system information and, if desired, change the configuration from the system (`sys`) mode.

▼ To Go to the `sys` Mode From the Remote CLI

1. **Connect to the remote CLI.**

   See "Accessing the Remote CLI" on page 94.

2. **To connect to `sys` mode, use the `send break sys` command.**

```
ndps> connect
Opening channel 1
IPC channel #: 1
ndps> send break sys
enter NDPS system cli...
sys>
```

3. **Enter `help` for options.**

```
sys> help
        set             - set commands
        clr             - clear commands
        show            - show commands
        help            - help commands
        version         - version command
        quit            - quit sys cli command
sys>
```

4. **To disconnect from `sys` mode, type `quit`.**

```
sys> quit
exit NDPS system cli...
ndps>
```

CHAPTER 7

Eclipse Development Environment

This chapter describes the Eclipse-based Teja Advance Development Environment (ADE) graphical user interface (GUI). Topics include:

ADE Introduction

Eclipse is an open source community where projects are focused on building extensive development platforms, runtimes, and application frameworks. Eclipse includes building, deploying, and managing software across the entire software life cycle.

Eclipse is more than a Java IDE. The Eclipse open source community has over 60 open source projects. These projects can be conceptually organized into seven different categories:

- Enterprise development
- Embedded and device development
- Rich client platform
- Rich internet applications
- Application frameworks
- Application lifecycle management (ALM)
- Service oriented architecture (SOA)

Refer to `http:// www.eclipse.org` for detailed information.

Starting the Eclipse-Based ADE GUI

Start the Eclipse-based ADE GUI by running `bin/eclipse.sh` from a shell terminal window.

▼ To Start the Eclipse-Based ADE GUI

- **Type:**

```
% /opt/SUNWndps/tools/bin/eclipse.sh
```

Creating a Teja Project

To use the Eclipse-based Teja ADE, the user creates a project. A project can be created from scratch or from an already existing Teja application. In the latter case, the project can be created in the same directory as the application or in a different one but linking some files from the original application directory.

▼ To Create a Project in the Same Directory as an Existing Teja Application

The following steps describe how to create a project in the same directory as an existing Teja application using `examples/PacketClassifier` as an example.

1. **From the File menu, select New Project.**
2. **Choose** `Teja/Teja Project` **in the list of possible wizards, then click Next.**
3. **In the Project Name field, type the name of the project.**

   In this example, type **`PacketClassifier`** (the name does not need to match the name of the application).
4. **To create the project in the directory of the application,**

   a. **Deselect** `Use default`**.**

b. Click the Browse button to get to the PacketClassifier directory.

c. Press OK.

Keeping Use default selected would create the project in the workspace. See FIGURE 7-1.

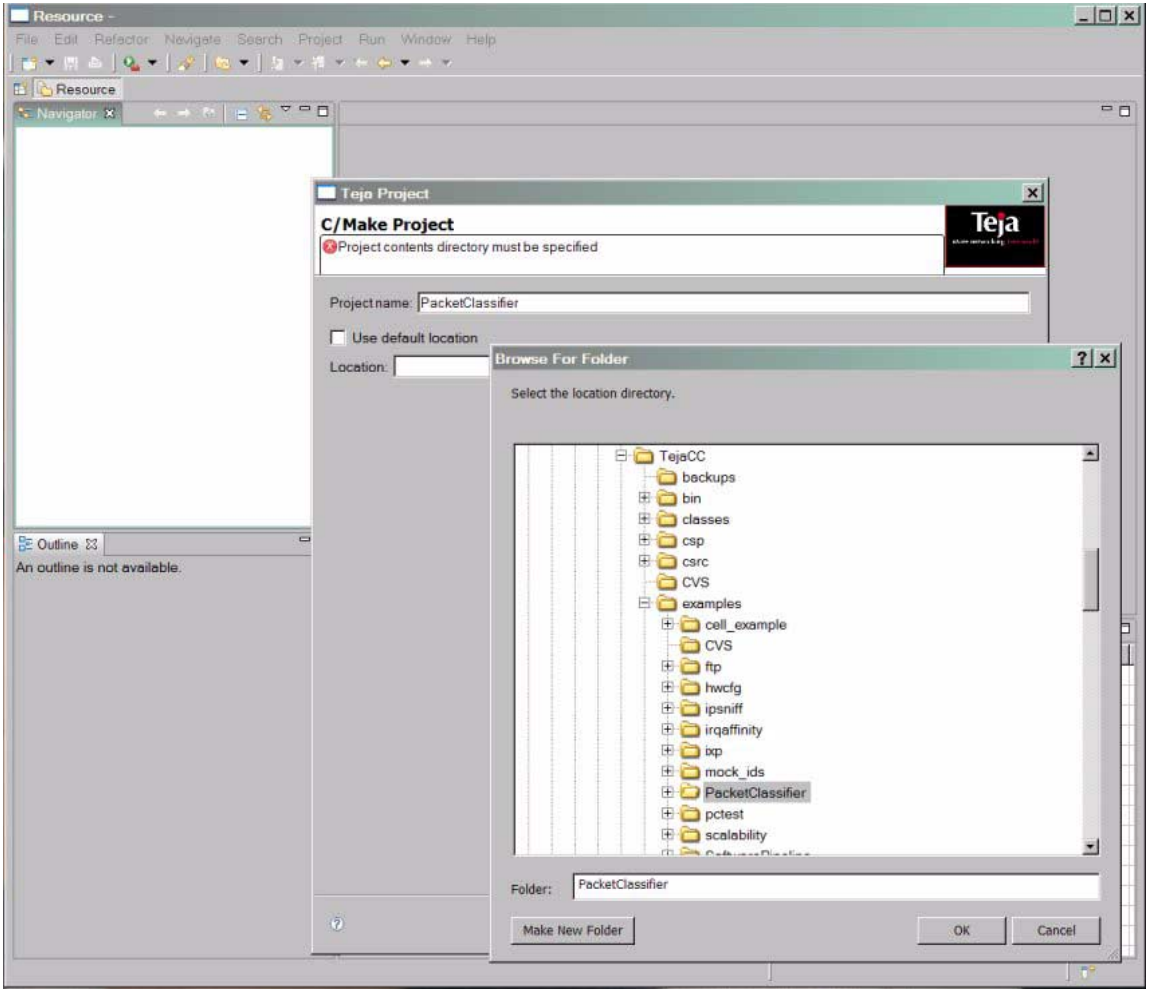


FIGURE 7-1 Eclipse-Based ADE GUI

You don not need to set the C/Make Project Setting tab, which defines the project and Builder settings, at this point.

In the Teja Project Settings tab, the information used to create the product specific graphic files is set. By default, these files have the same name as the project and are contained in the project directory. In the Graphic Files Info section, you can specify a different location and a different name, which will be the same for the three files with extension `tjh`, `tjs` and `tjm`. You can select whether to generate all three graphic files, or a subset, by putting a tag in the list in the General section.

You can populate the Teja Project Setting (FIGURE 7-2) tab in two ways:

- By specifying a configuration file, selecting the `Config file` button in the General section, and providing the name in the Configuration File section. This file is generated by `tejacc` with the name parameters. `tjc` contains all the information on the parameters `tejacc` was invoked (use the `parameters_file` switch to `tejacc.sh` to specify a different file name). This file includes which libraries are correlated to know which architectures that refer to which mapping. With a `config` file it is possible to validate across libraries.
- Providing the libraries and entry function names for hardware and software architectures and mapping. This approach decouples the hardware architecture, software architecture and mapping, allowing for visualizing one even when the other is not available or has bugs. To use this approach, select the Libraries and functions button in the General section and type the required information in the active sections. For each architecture and mapping, you have to provide the path of the shared library and the name of the entry point function.

In both ways, only the selected graphics files will be generated. Press the Finish button and a project is then created.

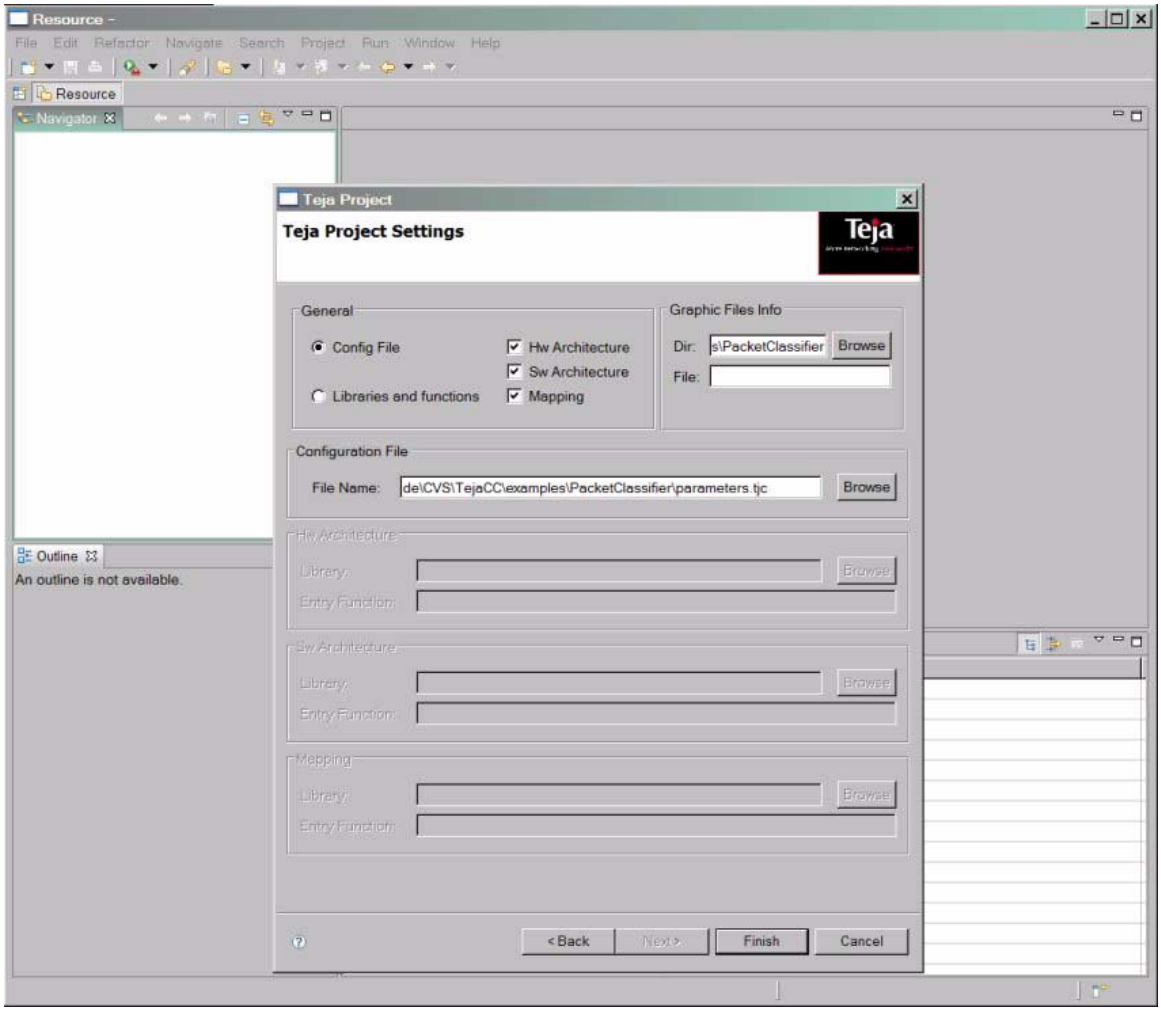


FIGURE 7-2 Teja Project Settings

You can also specify only the project name and directory and press Finish. The project is created without the graphic files, which can be added in a second step.

▼ To Add the Graphic Files to a Project

1. **In the Navigator view, right-click on the directory name inside the project where you would like to have the graphic files.**
2. **Open** `New/Other/Teja/Teja Graphic Files` **and press Next.**

   You get the Teja Project Settings tab. You need to fill this out as described in "To Create a Project in the Same Directory as an Existing Teja Application" on page 100.

Files and Viewers

The Eclipse-based Teja ADE can view three Teja elements: hardware architecture, software architecture, and mapping. To display the Teja element, the viewer uses the graphical information stored in separate files, one for each part of the application. These files are created when the project is created and have the same name as the project but with different extensions, `tjh` for the hardware architecture, `tjs` for the software architecture, and `tjm` for the mapping. These files contain the name of the library and entry function name and some graphical data such as the coordinates of the various objects, orientation, and type of routing.

After a project is created, it is visible in the Navigator tab by expanding and showing all the files and directories of the application, in addition to the graphical files. Double-clicking on these files opens a viewer for the element associated to the files.

Hardware Architecture Viewer

The Hardware Architecture Viewer (FIGURE 7-3) gives a graphical representation of the hardware architecture. Since hardware architectures can contain other hardware architectures, you can navigate the containment by double-clicking on architectures. The Outline tab provides a more straightforward visualization of the containment and the objects that a hardware architecture contains. To open this tab, go to the `Window/Show view/Outline` menu. Click any element in the outline to select the same element in the viewer, possibly changing the architecture shown to the one containing the selected object.

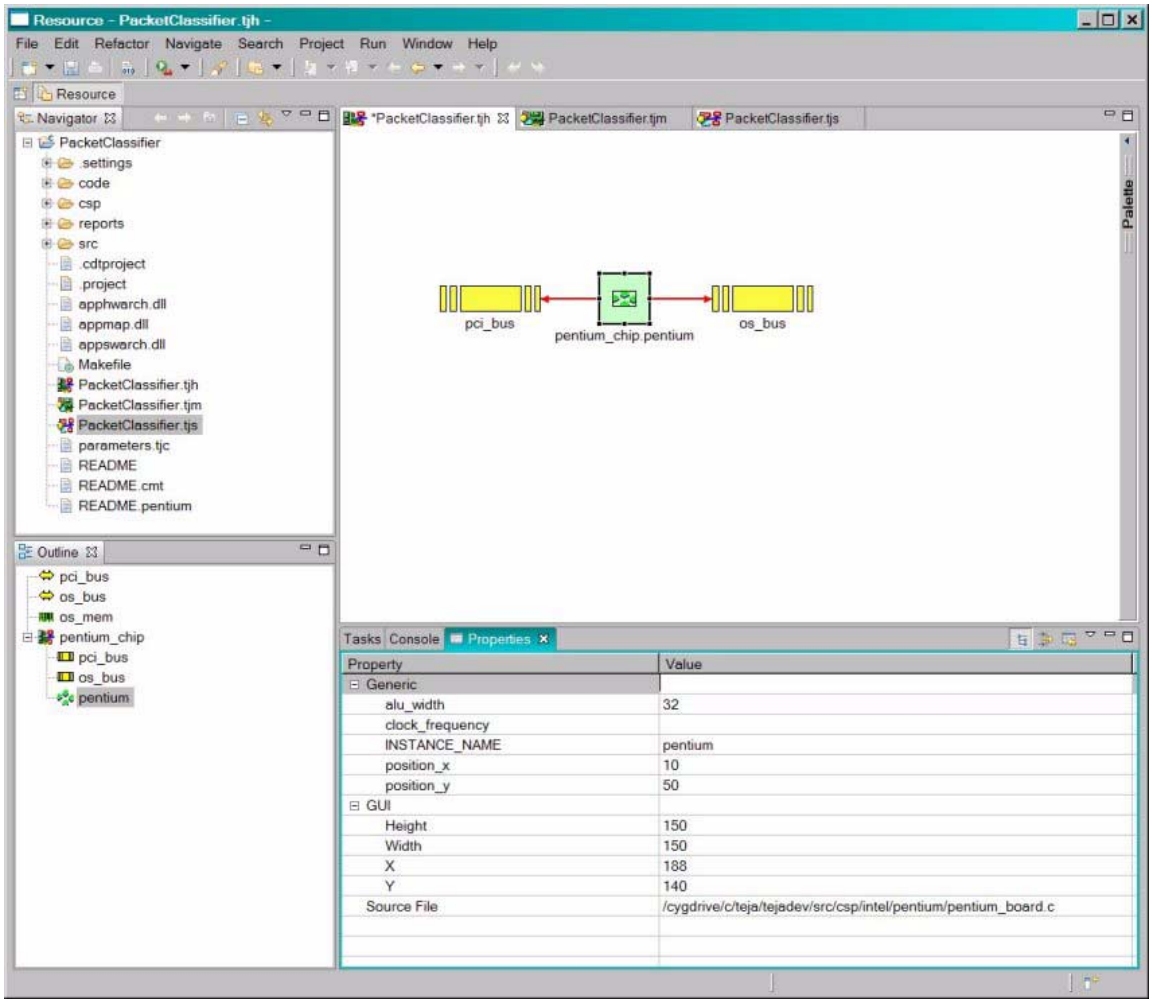

Resource - PacketClassifier.tjh -
File Edit Refactor Navigate Search Project Run Window Help
Resource
Navigator
PacketClassifier
.settings
code
csp
reports
src
.cdtproject
.project
apphwarch.dll
appmap.dll
appswarch.dll
Makefile
PacketClassifier.tjh
PacketClassifier.tjm
PacketClassifier.tjs
parameters.tjc
README
README.cmt
README.pentium
*PacketClassifier.tjh
PacketClassifier.tjm
PacketClassifier.tjs
Palette
pci_bus
os_bus
pentium_chip.pentium
Outline
pci_bus
os_bus
os_mem
pentium_chip
pci_bus
os_bus
pentium
Tasks Console Properties
Property
Value
Generic
alu_width
32
clock_frequency
INSTANCE_NAME
pentium
position_x
10
position_y
50
GUI
Height
150
Width
150
X
188
Y
140
Source File
/cygdrive/c/teja/tejadev/src/csp/intel/pentium/pentium_board.c

FIGURE 7-3 PacketClassifier Hardware Architecture – Inner Hardware

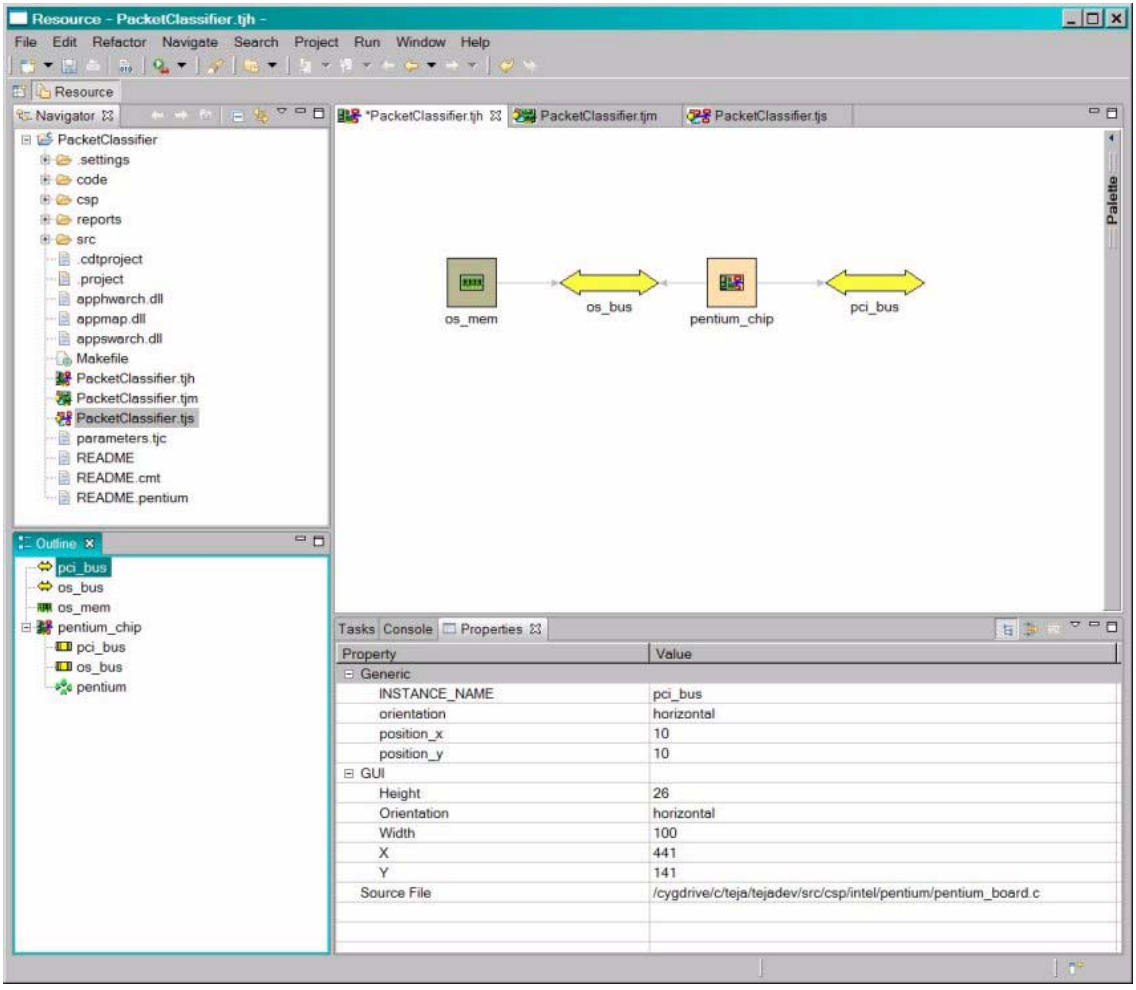


FIGURE 7-4 PacketClassifier Hardware Architecture – Outer Hardware

Netra DPS objects have properties with values of potential interest. The Properties tab displays such properties and their values. To open the Properties tab, go to the `Window/Show view/Others/Properties` menu. Along with the application properties there are also the GUI properties, some of which can be changed. For example, a bus has the GUI property AlignStyle. Clicking on the value and pulling down the menu (there is an arrow on the left) shows the possible values, in this case Horizontal and Vertical. By choosing one value and selecting Enter, the bus alignment change is applied. Another property is Source File which is the name of

the file where the selected object was created. If such a file is opened in the GUI, then clicking the object will indicate in the file the line of code where that object was created.

Software Architecture Viewer

The Software Architecture Viewer (FIGURE 7-5) gives a graphical representation of the software architecture and consists of two tabs. The viewer opens showing the OS view tab, with information of threads, processes, and processors. FIGURE 7-5 shows the OS View.

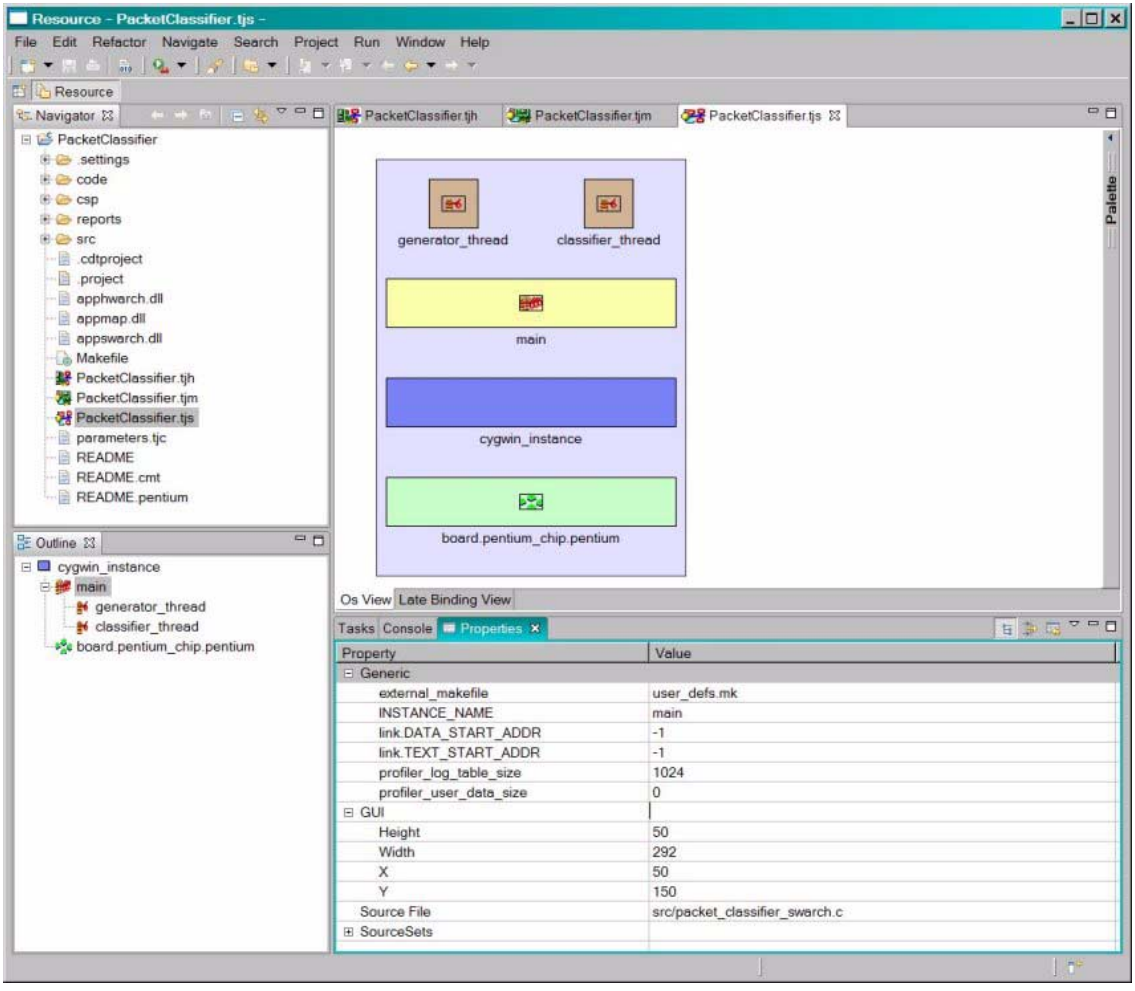


FIGURE 7-5 PacketClassifier Software Architecture – OS View

A second tab, the Late-Binding View (FIGURE 7-6) shows the information of threads, mutexes, channels, queues, and memory pools. When a validation is available, that is, the project was created through a configuration file, the processors displayed in the OS View are actually created in the hardware architecture. The processors are checked for a mismatch in the hardware architecture, and in case of error, the processors display a cross to highlight the problem. The outline and properties views are the same as the ones described for the hardware architecture ("Hardware Architecture Viewer" on page 104.)

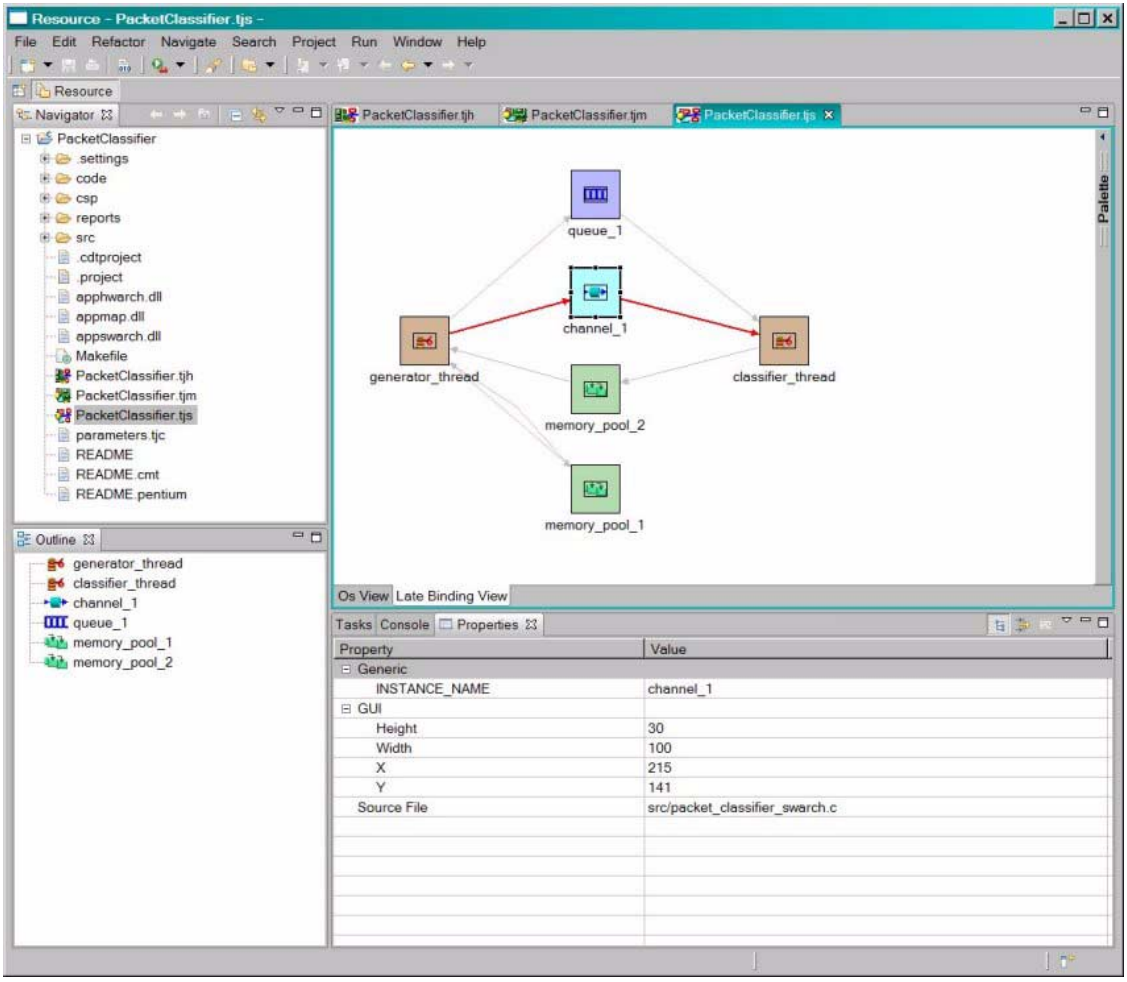


FIGURE 7-6 PacketClassifier Software Architecture - Late-Binding View

Mapping Viewer

The Mapping Viewer (FIGURE 7-7) shows which functions are mapped on which threads and which variables are mapped on which memory banks. In the Type combo box, select an element among Function, Memory Bank, Thread, and Variable.

The Mapping table displays a list of all the elements chosen in the combo. Selecting an element in the Mapping table causes all the elements mapped to it to be shown in the right-side list. For example, if you choose Function, the left side of the Mapping table shows you all the functions that are defined in the application code. When one function is selected, the names of the threads that have that function as an entry point function are shown on the right side of the table.

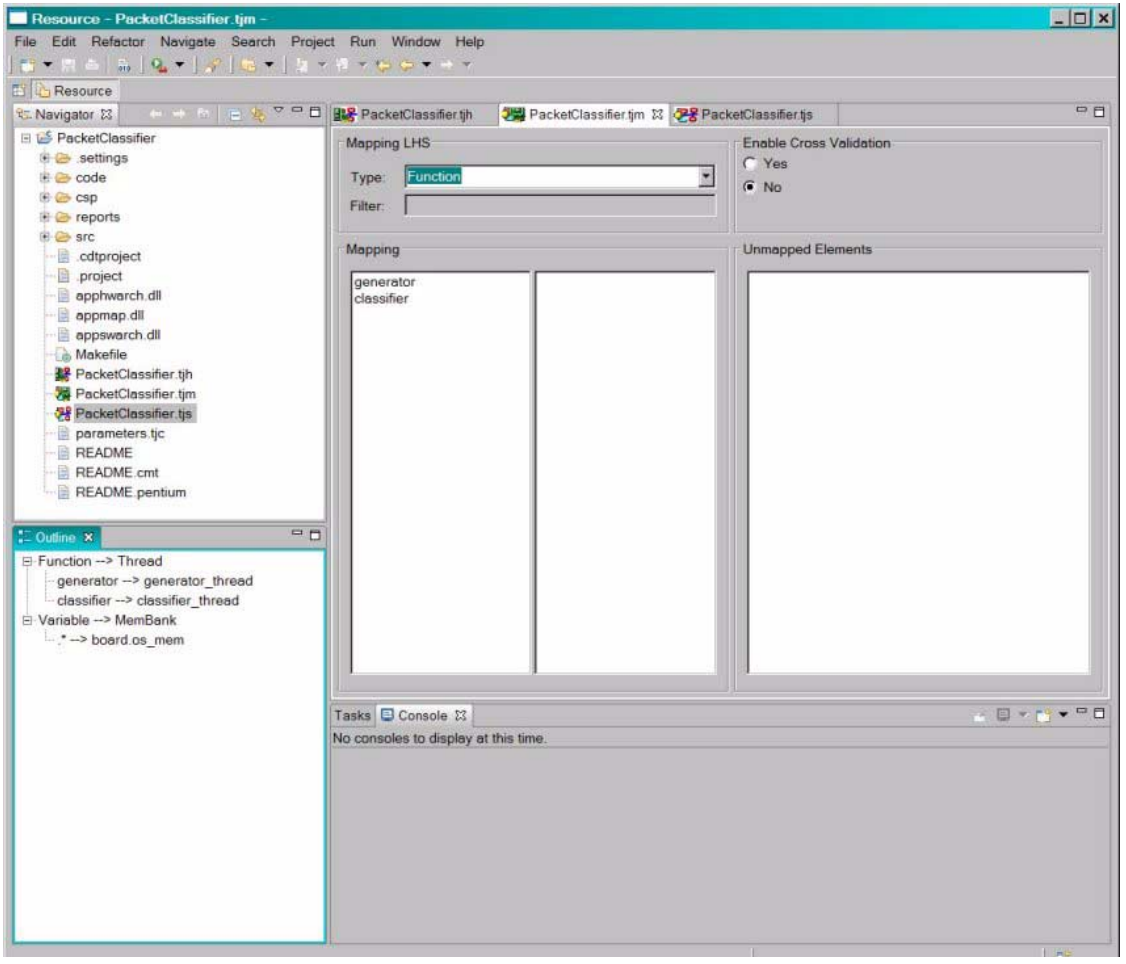


FIGURE 7-7 PacketClassifier Mapping

You can set the Mapping Viewer in one of two modes:

- The *basic* mode shows only information available in the corresponding mapping library without considering the hardware architecture, software architecture or application code. This mode is useful to see what you specified in the mapping, but does not validate that such information is correct when correlated to the rest of the elements. For example, if you map a function `f1()` on a thread `t1`, the basic mode shows no indication of whether `f1()` and `t1` actually exist in the application code and software architecture is provided. Also, if the user maps variables to a memory using the `regexp` variant of the mapping API, the regular expression provided for the variables is shown rather than the matching variables.
- The *extended* mode gathers and correlates the information from all the libraries. The extended mode provides architectural validation, but requires all the libraries to exist. As an example of information currently shown in this mode, if an application has unmapped variables, such variables are shown in the rightmost list when the Type combo is set to Variables. This is an error in the user application since all variables must be mapped.

Build

It is possible to compile the Teja application in the Eclipse-based ADE.

▼ To Compile the Teja Application in the Eclipse-Based ADE

1. **Create the target All.**
2. **Select the project name in the navigator tab and right click on Create Make Target.**
3. **Type in the Target Name and Make Target fields. Click Create.**
4. **To compile, select the project name again and right-click Build Make Target.**
5. **Choose All and click Build.**

   In the Console tab, the compiler output and warnings or errors, if any, are shown.

CHAPTER 8

Receive Packet Classification

This chapter describes the basic functions of the Receive Packet Classification and the Netra DPS software interface. Topics include:

Receive Packet Classification Introduction

The Sun multithreaded 10GbE with Network Interface Unit (NIU) networking hardware consists of a Receive Packet Classifier that performs L2/L3/L4 header parsing, matching and searching functions. Netra DPS provides the software interface to utilize this hardware mechanism.

Classification is needed for the following reasons:

- To spread traffic flows into multiple DMA for load balancing

  This classification spreads traffic flows across multiple CPUs so that each CPU hardware strand shares the load of 10 Gbps processing. By spreading the load across at least eight pipelines, packets are processed at 10Gbps preventing overloading of processing power on a particular processing unit.

- To separate and isolate different traffic types for special treatment

  This classification refers to blocking, re-routing, or to perform special processing to certain traffic types from the incoming traffic stream.

- To sustain high traffic throughput rate

  This classification sustains forwarding of 10Gbps of incoming traffic with a relatively small packet size from the 10Gbps ethernet ingress port to the 10Gbps egress port. Traffic must be spread into multiple DMA channels for processing.

Supported Networking Interfaces

The following network interfaces support classification:

- Sun multithreaded 10GbE, 4GbE and the on-chip 10GbE
- Network Interface Unit (NIU) in UltraSPARC T2.

Sun Multithreaded 10GbE and NIU Receive Packet Classifier

Sun multithreaded PCIE 10GbE, PCIe 4GbE, and 10GbE NIU supports two ways to spread input packets:

- Hashing based on Level 2, Level 3, and Level 4 (L2/L3/L4) headers

  Determines the target DMA channel based on a L2 RDC group and then a hash algorithm applied on the defined values of L2/L3/L4 header fields.

- Flow match based on L2/L3/L4 header

  Determines the target DMA channel based on the values of L2/L3/L4 header fields with the help of hardware lookup tables and TCAM preprogrammed with matching rules.

Hashing Based on Level 2, Level 3, and Level 4 Header Classification

The procedure of hashing includes a hash lookup table based on the hash key. The hash key is created by applying a hash algorithm to a flow key and the flow key is generated from extracting certain fields from Level 2, Level 3, and Level 4 (L2/L3/L4) packet headers.

The header fields in the flow key selections consist of the following individual header fields:

- MAC port number
  - MAC destination address
  - VLAN ID if tagged
- Protocol ID/next header
- IP source address, IP destination address
- Level 4 source and destination port number.
  or a combination of these fields.

Hash Key generation

The hashing algorithm is based on polynomial hashing with CRC-32C. The algorithm is a 32-bit hash value. The last four bits of the value is used to index into a hardware hash table to lookup a DMA channel. In a Netra DPS environment, one RDC table is used. The DMA channel number is one-to-one corresponding to the RDC table entry number, the value of the last four bits, therefore, equals the DMA channel number.

$X^{32} + x^{28} + X^{27} + X^{26} + X^{25} + X^{23} + X^{22} + X^{20} + X^{19} + X^{18} + X^{14} + X^{13} + X^{11} + X^{10} + X^{9} + X^{8} + X^{6} + 1$

Application

You can use hashing for general load spreading and load balancing applications. The traffic load of each DMA channel depends on the value in the header fields used for the hash. Since the target DMA channel is determined by a polynomial, the correlation between the header value and the target DMA channel cannot be easily

determined. How balance of the DMA channels are spread also depends on the value and range of the header fields. Hashing is considered a general purpose load spreading scheme.

Classification Policy

Hashing is enabled by default. The hash policy is determined by setting the FLOW_POLICY to one of the values shown in TABLE 8-1:

TABLE 8-1 Hash Policy Values

Value	Meaning
HASH_IP_ADDR	Hash on IP destination and source addresses
HASH_IP_DA	Hash on IP destination address
HASH_IP_SA	Hash on IP source address
HASH_VLAN_ID	Hash on VLAN ID
HASH_PORTNUM	Hash on physical MAC port number
HASH_L2DA	Hash on L2 destination address
HASH_PROTO	Hash on protocol number
HASH_SRC_PORT	Hash on L4 source port number
HASH_DST_PORT	Hash on L4 destination port number
HASH_ALL	Hash on all of the above fields
TCAM_CLASSIFY	Perform TCAM lookup

The default FLOW_POLICY is set to HASH_ALL, meaning that the hash hardware hash algorithm is applied on all of the above header fields. To disable hash, set FLOW_POLICY to 0 or TCAM_CLASSIFY. When set to 0, no traffic spreading is performed. All traffic ends up at a default DMA channel. When set to TCAM_CLASSIFY, traffic spreading is determined by predefined flow specifications.

Flow Match Based on Level 2, Level 3, and Level 4 Header Classification

Level 2 (L2) Classification

L2 header classification is performed by lookup into the MAC address tables or the VLAN tables. MAC address tables and VLAN tables consist of information on the target DMA channels. L2 header classification is not currently supported.

Level 3 and Level 4 (L3/L4) Classification

L3/L4 header classification relies on the TCAM hardware to determine how traffic flows are distributed. There are multiple TCAM hardware entries (256 in Sun multithreaded 10GbE, 128 in NIU) for specifying flow specification. The CAM lookup table key generation use the concept of classes of packets to assemble a key. With the CAM key, a packet goes through a single CAM lookup table for an associative search. The L3/L4 header classification starts when the header parse identifies the incoming L2/L3 packet type.

The following packet classes are supported in Netra DPS:

- UDP over IPv4
- TCP over IPv4
- SCTP over IPv4
- IPSEC (AH/ESP) over IPv4
- TCP over IPv6
- UDP over IPv6
- SCTP over IPv6A
- IPSEC (AH/ESP) over IPv6

Applications

You can use flow tables and TCAM to direct a particular type of traffic flow (with different traffic classes) into particular DMA channels. Flow tables and TCAM are ideal for use in load balancing applications.

Classification Programming Interface

The interface to the Flow Matching scheme is the `ETH_IOC_SET_CLASSSIFY "IO Control"` command of the Netra DPS ethernet interface. The following shows the calling convention of the interface:

```
eth_ioc(ihdlnet[port], ETH_IOC_SET_CLASSIFY, (void *)&clsfy_ioc);
```

`ihdlnet[]` is an array of device driver handle indexed by the ethernet port number [`port`]. `ETH_IOC_SET_CLASSIFY` is the set classifier command.

The `clsfy_ioc` structure is defined as follows:

```
typedef struct classify_ioc_s {
   uint_t opcode;
   uint_t action;
   flow_spec_t flow_spec;
} classify_ioc_t;
```

opcode

`opcode` specifies what to do about a new traffic flow. TABLE 8-2 shows possible `opcode` values:

TABLE 8-2 `opcode` Values

Value	Meaning
`IOC_ADD_CLASSIFY`	Add a flow.
`IOC_INVALIDATE_CLASSIFY`	Invalidate a flow.

action

`action` specifies what action to take when there is a match. TABLE 8-3 shows possible `action` values:

TABLE 8-3 `action` Values

Value	Meaning
`IOC_FLOW_ACCEPT`	Accept when a match.
`IOC_FLOW_DISCARD`	Discard when a match.

flow_spec

`flow_spec` is the flow specification specifying the characteristics of a flow. The following shows the `flow_spec` structure:

```
typedef struct flow_spec_s {
                    uint_t fs_type;
                    uint_t index;
                    uint_t channel;
                    union {
                              flow_spec_ipv4_t ip4;
                              flow_spec_ipv6_t ip6;
                              flow_spec_l2_t   l2;
                              uint8_t hd[64];
                    } ue, um;
} flow_spec_t;
```

fs_type

TABLE 8-4 shows the possible values of the traffic flow spec types (`fs_type`):

TABLE 8-4 `fs_type` Possible Values

Value	Meaning
`FSPEC_TCPIP4`	TCP over IPv4
`FSPEC_UDPIP4`	UDP over IPv4
`FSPEC_AHIP4`	IPSEC/AH over IPv4
`FSPEC_ESPIP4`	IPSEC/ESP over IPv4
`FSPEC_SCTPIP4`	SCTP over IPv4
`FSPEC_TCPIP6`	TCP over IPv6
`FSPEC_UDPIP6`	UDP over IPv6
`FSPEC_AHIP6`	IPSEC/AH over IPv6
`FSPEC_ESPIP6`	IPSEC/ESP over IPv6
`FSPEC_SCTPIP6`	SCTP over IPv6

index

This is the index into the TCAM entries (for L3/L4 TCAM classification) or index into the MAC or VLAN table (for L2 MAC/VLAN classification).

- For TCAM on Sun multithreaded 10GbE: value range is 0 ~ 255
- For TCAM on NIU: value range is 0 ~ 127

Note – The software application must keep tract of the index number.

---

channel

This is the target DMA channel ranges 0 ~ 15.

ue or um

`ue` is the 5-tuple for IPv4 or 4-tuple for IPv6 structure for L3/L4 TCAM classification. For L2 classification, it is the L2 header structure. `um` is the bit-mask corresponding to the `ue`. Set 1 to bit-mask for `don't care` (not to compare). Set 0 in bit-mask to compare.

hd

This is the entire 64-bit header.

flow_spec_ipv4_t

The following is the IPv4 flow specification structure:

```
typedef struct port_s {
            uint16_t src;
            uint16_t dst;
     } port_t;

     typedef struct spi_port_s {
            uint32_t port;
     } spi_port_t;

     typedef struct flow_spec_ipv4_s {
            uint8_t protocol;
            union {
                    port_t tcp;
                    port_t udp;
                    spi_port_t spi;
            } port;
            uint32_t src;
            uint32_t dst;
     } flow_spec_ipv4_t;
```

flow_spec_ipv6_t

The following is the IPv6 flow specification structure:

```
typedef struct flow_spec_ipv6_s {
        uint8_t protocol;
        union {
                    port_t tcp;
                    port_t udp;
                    spi_port_t spi;
        } port;
        uint8_t src[16];
        uint8_t dst[16];
} flow_spec_ipv6_t;
```

flow_spec_l2_t

This is the L2 header structure as shown below:

```
typedef struct flow_spec_l2_s {
                  uint8_t dst[6];          /* MAC address */
                  uint8_t src[6];          /* MAC address */
                  uint16_t type;           /* Ether type */
                  uint16_t vlantag;        /* VLANID|CFI|PRI */
} flow_spec_l2_t;
```

Examples

▼ To Use Hash Flow

- **Set `FLOW_POLICY` to a desired policy. For example:**

```
gmake .... FLOW_POLICY=HASH_ALL
```

This command tells Sun multithreaded 10GbE with NIU hardware to hash on all L2/L3/L4 header fields.

▼ To Use TCAM Classification

This example shows how a flow table can be established in the application.

1. **Set up an array of flow table entries.**

For example, use entries with the following structure:

```
typedef struct flow_spec_ip4_tab_s {
        int             index;
        uint8_t         protocol;
        uint8_t         protocol_mask;
        uint16_t        src_port;
        uint16_t        src_port_mask;
        uint16_t        dst_port;
        uint16_t        dst_port_mask;
        char            *src_addr;
        char            *src_addr_mask;
        char            *dst_addr;
        char            *dst_addr_mask;
        int             action;
        uint8_t         dma_chan;
} flow_spec_ip4_tab_t;
```

2. Populate the flow table as shown in the below example.

```
flow_spec_ip4_tab_t ip4_flow_tab[] = {
                    {0, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.1", "255.255.255.0",
                                       FLOW_ACCEPT, 0},
                    {1, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.2", "255.255.255.0",
                                       FLOW_ACCEPT, 1},
                    {2, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.3", "255.255.255.0",
                                       FLOW_ACCEPT, 2},
                    {3, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.4", "255.255.255.0",
                                       FLOW_ACCEPT, 3},
                    {4, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.5", "255.255.255.0",
                                       FLOW_ACCEPT, 4},
                    {5, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.6", "255.255.255.0",
                                       FLOW_ACCEPT, 5},
                    {6, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.7", "255.255.255.0",
                                       FLOW_ACCEPT, 6},
                    {7, IPPROTO_UDP, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
                                       "192.30.50.0", "255.255.255.255",
                                       "192.31.50.8", "255.255.255.0",
                                       FLOW_ACCEPT, 7},
                    {-1, 0, 0, 0, 0, 0, 0, "", "", "", "", 0, 0}
};
```

3. **Write a parsing function to parse the entries in the table as shown in the below example.**

```
void
classify_parse_entries(uint_t flow_cfg, uint8_t port,
                    uint8_t chan, flow_spec_ip4_tab_t *fe)
{
          classify_ioc_t  clsfy_ioc;
          int i;
          for (i = 0; fe[i].index != -1; i++) {
                    if (fe[i].dma_chan != chan)
                              continue;
                    clsfy_ioc.opcode = IOC_ADD_CLASSIFY;
                    clsfy_ioc.flow_spec.fs_type = FSPEC_UDPIP4;
                    clsfy_ioc.flow_spec.index = fe[i].index;
                    clsfy_ioc.flow_spec.channel = fe[i].dma_chan;
                    clsfy_ioc.flow_spec.ue.ip4.protocol =
                                                        fe[i].protocol;
                    clsfy_ioc.flow_spec.ue.ip4.port.udp.src =
                                                  fe[i].src_port;
                    clsfy_ioc.flow_spec.ue.ip4.port.udp.dst =
                                                  fe[i].dst_port;
                    clsfy_ioc.flow_spec.ue.ip4.src =
                                                  ips2h(fe[i].src_addr);
                    clsfy_ioc.flow_spec.ue.ip4.dst =
                                                  ips2h(fe[i].dst_addr);
                    clsfy_ioc.flow_spec.um.ip4.protocol =
                                                  ~fe[i].protocol_mask;
                    clsfy_ioc.flow_spec.um.ip4.port.udp.src =
                                                  ~fe[i].src_port_mask;
                    clsfy_ioc.flow_spec.um.ip4.port.udp.dst =
                                                  ~fe[i].dst_port_mask;
                    clsfy_ioc.flow_spec.um.ip4.src =
                                        ~ips2h(fe[i].src_addr_mask);
                    clsfy_ioc.flow_spec.um.ip4.dst =
                                        ~ips2h(fe[i].dst_addr_mask);
                    if (fe[i].action == FLOW_ACCEPT)
                              clsfy_ioc.action = IOC_FLOW_ACCEPT;
                    /* Program the TCAM HW */
                    (void) eth_ioc(ihdlnet[port],
                                                  ETH_IOC_SET_CLASSIFY,
                                                  (void *)&clsfy_ioc);
          }
}
```

4. **During the build, enable TCAM classification and disable hashing. To do this, type:**

```
gmake .... FLOW_POLICY=TCAM_CLASSIFY
```

This command enables Sun multithreaded 10GbE with NIU hardware to enable TCAM classification with matching rules as shown in Step 1 to Step 3.

CHAPTER 9

Reference Applications

This chapter describes reference applications including IP Packet Forwarding and radio link protocol (RLP) applications. Topics include:

IP Packet Forwarding Application

The IP packet forwarding application (`ipfwd`) simulates an IPv4 (Internet Protocol Version 4) forwarding operation. When each packet is received, the program performs a forward table search, rewrites the MAC source and destination address, rewrites the TTL, and recalculates the IP header checksum. The packet is then sent out to a specified port. The application can either simulate an IP packet forwarding lookup table or actually perform an IP packet forwarding lookup through the use of the lookup table. The goal of this application is to provide a framework for users to develop their own application based on their own lookup algorithm.

Source Files

All `ipfwd` source files are located in the following package directory: `/opt/SUNWndps/src/apps/ipfwd`. The contents include:

- Makefiles for two different ethernet hardware devices include:
  - `./Makefile.ipge`
  - `./Makefile.nxge`
- Information files for `ipfwd` and LDoms configuration include:
  - `./README`
  - `./README.config`
- Build scripts for one step build include:
  - `./build_10g`
  - `./build_10g_niu`
  - `./build_1g`
  - `./build_1g_1060`
  - `./build_4g`
- Application files and application configuration files include:
  - `./src`
  - `./src/app`
  - `./src/app/common.h`
  - `./src/app/init.c`
  - `./src/app/ipfwd.c`
  - `./src/app/ipfwd_config.c`
  - `./src/app/ipfwd_config.h`
  - `./src/app/ipfwd_lib.c`
  - `./src/app/lb_objects.h`
  - `./src/app/ldc_malloc.c`
  - `./src/app/ldc_malloc_config.h`
  - `./src/app/radix`
  - `./src/app/radix/ipfwd_radix.h`
  - `./src/app/radix/radix.c`
  - `./src/app/radix/radix.h. /src/app/rx.c`
  - `./src/app/tx.c`
  - `./src/app/user_common.c`
  - `./src/app/user_common.h`

- ./src/common
- ./src/common/include
- ./src/common/include/fibtable.h

- System configuration for the application include:
  - ./src/config
  - ./src/config/ipfwd_hwarch.c
  - ./src/config/ipfwd_map.c
  - ./src/config/ipfwd_swarch.c
- Solaris FIB Control applications for ipfwd LDoms include:
  - ./src/solaris
  - ./src/solaris/Makefile
  - ./src/solaris/fibctl.c

Compiling the ipfwd Application

On a system that has SUNWndps installed, go to the src/app/ipfwd directory and use the build scripts shown in TABLE 9-1 to build the application.

Note – Copy the application files from the /opt/SUNWndps/src/apps/ipfwd directory of the package into your workspace before compiling the application.

Build Scripts

TABLE 9-1 shows the ipfwd application build scripts.

TABLE 9-1 ipfwd Application Build Scripts

Build Script	Usage
build_1g <cmt> [ldoms]	Build ipfwd application to run on e1000g Ethernet interface.
build_1g_1060 [ldoms]	Build ipfwd application to run on e1000g Ethernet interface on the Netra ATCA CP3060 System.

TABLE 9-1 `ipfwd` Application Build Scripts *(Continued)*

Build Script	Usage
`build_4g <cmt> [ldoms]`	Build `ipfwd` application to run on Sun multithreaded 10GbE with NIU QGC (quad 1Gbps `nxge` Ethernet interface)
`build_10g <cmt> [ldoms]`	Build `ipfwd` application to run on Sun multithreaded 10G (dual 10Gbps `nxge` Ethernet interface).
`build_10g_niu [ldoms]`	Build `ipfwd` application to run on NIU (dual 10Gbps UltraSPARC T2 Ethernet Interface) on a CMT2 based system.

Note – This script does not have the <cmt> option because the Netra ATCA CP3060 system is UltraSPARC T1 based only.

Build Script Arguments

< > – Required arguments

[ ] – Optional arguments

Argument Descriptions

`<cmt>`

This argument specifies whether to build the `ipfwd` application to run on the UltraSPARC T1 platform or UltraSPARC T2 platform.

`cmt1` – Build for CMT1 (UltraSPARC T1) architecture

`cmt2` – Build for CMT2 (UltraSPARC T2) architecture

This argument is required for scripts that expect `<cmt>`.

[`ldoms`]

This optional argument specifies whether to build the `ipfwd` application to run on the LDoms environment. When this flag is specified, the `IPFWD LDoms` reference application will be compiled. If this argument is not specified, then the non-LDoms application is compiled.

Build Example

In `/opt/SUNWndps/src/apps/rlp`, pick the correct build script and run it. For example, to build for Sun multithreaded ethernet adapter on Netra or Sun Fire T2000 systems, enter the following at your shell window.

```
% build_10g cmt1
```

In this example, `build_10g` is used to build IP forwarding application to run on the Sun multithreaded 10Gb ethernet. The <cmt> argument is specified as `cmt1` to build the application to run on UltraSPARC T1-based Netra or Sun Fire T2000 systems.

▼ To Run the `ipfwd` Application

1. **Copy the binary into the `/tftpboot` directory of the `tftpboot` server.**
2. **On the `tftpboot` server, type:**

```
% cp your_workspace/ipfwd/code/ipfwd/ipfwd /tftpboot/ipfwd_xxx
```

3. **On the target machine, type the following at the `ok` prompt:**

```
ok boot net:,ipfwd_xxx
```

Note – `net` is an OpenBoot PROM alias corresponding to the physical path of the network.

Default Configurations

This section shows the default configurations:

Default System Configuration

	NDPS domain (Strand IDs)	Statistics (Strand ID)	Other domains (Strand IDs)
CMT1 NON-LDOM:	0 - 31	31	N/A
CMT1 LDOM:	0 - 19	19	20 - 31
CMT2 NON-LDOM:	0 - 63	63	N/A
CMT2 LDOM:	0 - 39	39	40 - 63

Main files that control the system configurations:

- `ipfwd/src/app/config/ipfwd_swarch.c`
- `ipfwd/src/app/config/ipfwd_map.c`

Default `ipfwd` Application Configuration

Application runs on	# of Ports used	# of Channels per Port	Total # of Q instances	Total # of strands used
1G (ipge):	2	1 (see note)	2	6
1G (CP 1060):	2	1 (see note)	2	6
4G-PCIE (nxge QGC):	4	1	4	12
10G-PCIE (nxge 10G):	1	4	4	12
10G-NIU (niu 10G):	1	8	8	24

Note – `e1000g` (`ipge`) only has one DMA channel per port.

Main files that control the application configurations:

- `ipfwd/src/app/ipfwd_config.c`
- `ipfwd/src/app/ipfwd_config.h`

Other Options

Profiling

To enable profiling, follow the instructions given in each build script that states:

```
# Replace the above gmake command with the following to
# generate code with profiling enabled.
```

Radix Forwarding Algorithm

To enable the Radix Forwarding Algorithm for IP forwarding, uncomment the following line from `Makfile.ipge` or `Makefile.nxge` for `e1000g` 1Gb PCIe ethernet adapter or Sun multithreaded 10Gb and 1Gb PCIe Ethernet adapter, respectively:

```
- DIPFWD_RADIX
```

Bypassing the `ipfwd` Operation

To bypass the `ipfwd` operation (that is, receive --> transmit without forwarding operation), uncomment the following line from `Makfile.ipge` or `Makefile.nxge` for 1Gb PCIe ethernet adapter or Sun multithreaded 10 Gb and 1Gb PCIe Ethernet adapter, respectively:

```
-DIPFWD_RAW
```

Multiple Forward Port Destinations

When the Multiple Forward Destinations option is enabled, the output destination port is determined by the output of the Forwarding Table Lookup. Otherwise, the output destination port is the same as the input port. To enable this option, uncomment the following line from `Makefile.ipge` or `Makefile.nxge` for Ophir or Sun multithreaded 10GbE Ethernet, respectively:

-DIPFWD_MULTI_QS

Hash Policy for Spreading Traffic to Multiple DMA Channels

You can specify a policy for spreading traffic into multiple DMA flows by hardware hashing. TABLE 9-2 describes each policy:

TABLE 9-2 Hash Policy Descriptions

Name	Definition
HASH_IP_ADDR	Hash on IP destination and source addresses.
HASH_IP_DA	Hash on IP destination address.
HASH_IP_SA	Hash on IP source address.
HASH_VLAN_ID	Hash on VLAN ID.

To enable one of the above policies, add the following into the `gmake` line:

`HASH_POLICY=`*policy*

where *policy* is one of those specified in TABLE 9-2. For example, to enable hash on an IP destination and source address, add `HASH_POLICY=HASH_IP_ADDR` to the `gmake` line as shown below:

```
gmake -f Makefile.nxge NETHW_TYPE=2 CMT=N2 HASH_POLICY=HASH_IP_ADDR
```

If none of the policies listed in TABLE 9-2 are specified, a default policy is given. When you use the default policy, all `L2/L3/L4` header fields are used for spreading traffic.

`ipfwd` Flow Configurations

The `ipfwd_config.c` file assists you in mapping application tasks to CPU core and hardware strands. Normally, mapping is set in the `ipfwd_map.c` file in the `config` directory. This configuration file is a productivity tool. This file provides a way to facilitate mapping in a quick manner without any modification to the `ipfwd_map.c` file.

This configuration file is not a replacement of `ipfwd_hwarch.c`, `ipfwd_swarch.c` and `ipfwd_map.c`. This framework is to conduct performance analysis and measurement with different system configurations. The default (`*_def`)

configurations specified assumes a minimum of 16 threads of the system allocated for Netra DPS in `ipfwd_map.c` and all memory pool resources required are declared in `ipfwd_swarch.c`. You still need to specify system resources declarations and mapping in `ipfwd_hwarch.c`, `ipfwd_swarch.c`, and `ipfwd_map.c`. The configuration is assigned to a pointer named `ipfwd_thread_config`.

Note – You have the option to bypass this file entirely and perform all the mapping in `ipfwd_map.c`. In this case, you would also need to modify `ipfwd.c` so that it does not interpret the contents of this file.

Format

Each application configuration is represented in an array of a six-element entry. Each entry (each row) represents a software task and its corresponding resources:

- **Thread ID** – Strand number of the hardware strand (0 ~ 31 on an UltraSPARC T1 system, 0 ~ 63 on an UltraSPARC T2 system) on which this software task is to be run.
- **HW init** – If zero, it indicates no Ethernet port needs to be opened when this task is activated. If non-zero, it indicates ethernet port (port number specified by port#) needs to be opened. The contents of `OPEN_OP` consists of vendor and device id as (`NXGE_VID << 16) | NXGE_DID`.
- **Port#** – This is the port number of the ethernet port to be opened. Port# should start from zero. Port# denotes the first ethernet port probed on the system for this given `OPEN_OP`.
- **Chan#** – If this is a multichannel device (such as Sun multithreaded 10GbE with NIU), this entry indicates the channel number within each port. Sun multithreaded 10GbE device has 24 transmit channels (0~23) and 16 receive channels (0~16) in each port. Sun multithreaded 10GbE with NIU has 16 channels (both `tx` and `rx`) in each port. `e1000g` (`ipge`) Ethernet device has one channel in each port (it is a non-channelized device).
- **Role** – This is the role of the software task.

  `TROLE_ETH_NETIF_RX` — This task performs a receive function.
  `TROLE_ETH_NETIF_TX` — This task performs a transmit function.
  `TROLE_APP_IPFWD` — This task performs IP forwarding function.

  See `common.h` for all definitions. If you do not want to run any software task on this hardware strand, the role field should be set to -1. By default, during initialization of the `ipfwd` application, the hardware strand that encounters a -1 software role is parked.

Note – A parked strand is a strand that does not consume any pipeline cycles (an inactive strand).

- **MemPool#** – This is the identity of the memory pool. Note that in this reference application, each Ethernet port has its own memory pool. Each channel within each port has its own memory pool. Memory pools are declared in `ipfwd_swarch.c`.

Note – The application can be configured such that a single memory pool is dedicated to a particular DMA channel or all DMA channels sharing a global memory pool. The default configuration is one memory pool per DMA channel.

Radio Link Protocol Application

The Radio Link Protocol (RLP) application (`rlp`) simulates Radio Link Protocol operation, which is one of the protocols in the CDMA-2000 High Rate Packet Data Interfaces (HRPD-A). This application implements the forwarding direction fully, with packets flowing from PDSN --> AN --> AT (that is, Packet Data Serving Node to Access Network to Access Terminal). Reverse direction support is also implemented, but requires an AT side application which can generate NAKs (Negative Acknowledges). The application must be modified to process reverse traffic.

Compiling the RLP Application

On a system that has `SUNWndps` installed, go to the `src/app/rlp` directory and use the build scripts shown in TABLE 9-1 to build the application.

Note – First copy the application files from the `/opt/SUNWndps/src/apps/rlp` directory of the package into your workspace before compiling the application.

Build Scripts

TABLE 9-1 shows the RLP (`rlp`) application build scripts.

TABLE 9-3 `rlp` Application Build Scripts

Build Script	Usage
`build_1g <cmt> [ldoms]`	Build `rlp` application to run on the `e1000g` Ethernet interface.
`build_1g_1060 [ldoms]`	Build `rlp` application to run on the e1000g Ethernet interface on the Netra ATCA CP3060 system.
`build_4g <cmt> [ldoms]`	Build `rlp` application to run on Sun multithreaded 10GbE with NIU QGC (quad 1Gbps `nxge` ethernet interface).
`build_10g <cmt> [ldoms]`	Build `rlp` application to run on Sun multithreaded 10G (dual 10Gbps `nxge` Ethernet Interface).
`build_10g_niu [ldoms]`	Build `rlp` application to run on NIU (dual 10Gbps UltraSPARC T2 ethernet interface) on a CMT2 (UltraSPARC T2) based system.

Build Script Arguments

< > – Required arguments

[ ] – Optional arguments

Arguments Descriptions

- `<cmt>`

  This argument specifies whether to build the `rlp` application to run on the UltraSPARC T1 platform or UltraSPARC T2 platform.

  `cmt1` – Build for CMT1 (UltraSPARC T1) architecture

  `cmt2` – Build for CMT2 (UltraSPARC T2) architecture

  This argument is required for scripts that expect `<cmt>`.

- `[ldoms]`

  This i optional argument specifies whether to build the `rlp` application to run on the LDoms environment. When this flag is specified, the RLP LDoms reference application will be compiled. If this argument is not specified, then the non-LDoms application is compiled.

Build Example

In `/opt/SUNWndps/src/apps/ipfwd`, pick the correct build script and run it. For example, to build for Sun multithreaded 10Gb PCIe ethernet adapter on Netra or Sun Fire T2000 systems, enter the following at your shell window:

```
% build_10g cmt1
```

In this example, `build_10g` is used to build RLP application to run on the Sun multithreaded 10Gb Ethernet. The <cmt> argument is specified as `cmt1` to build the application to run on UltraSPARC T1-based Netra or Sun Fire T2000 systems.

▼ To Run the Application

1. **Copy the binary into the `/tftpboot` directory of the `tftpboot` server, and perform.**
2. **On the `tftpboot` server, type:**

```
% cp your_workspace/rlp/code/rlp/rlp /tftpboot/rlp_xxx
```

3. **On the target machine, type the following at the `ok` prompt:**

```
ok boot net:,rlp_xxx
```

Note – `net` is an OpenBoot PROM alias corresponding to the physical path of the network.

Default Configurations

This section shows the default configurations.

Default System Configuration

```
                        NDPS domain     IPC polling,
                                        Statistics      Other domains
                        (Strand IDs)    (Strand IDs)    (Strand IDs)
                        ----------      -----------     -----------
CMT1 NON-LDOM:          0 - 31          31              N/A
CMT1 LDOM:              0 - 19          18, 19          20 - 31
CMT2 NON-LDOM:          0 - 63          63              N/A
CMT2 LDOM:              0 - 39          38, 39          40 - 63
```

Main files that control the system configurations are:

- `ipfwd/src/app/config/rlp_swarch.c`
- `ipfwd/src/app/config/rlp_map.c`

Default `rlp` Application Configuration

```
Application             # of     # of             Total #         Total #
runs                    Ports    Channels         of Q            of strands
on                      used     per Port         instances       used
----------              -----    ---------        ---------       -------
1G (ipge):              2        1 (see note)     2               6
1G (CP 1060):           2        1 (see note)     2               6
4G-PCIE (nxge QGC):     4        1                4               12
10G-PCIE (nxge 10G):    1        4                4               12
10G-NIU (niu 10G):      1        8                8               24
```

Note – `e1000g` (`ipge`) only has one DMA channel per port.

Main files that control the application configurations are:

- `ipfwd/src/app/rlp_config.c`
- `ipfwd/src/app/rlp_config.h`

Other RLP Options

▼ To Enable Profiling

- **Follow the instructions given in each build script that states:**

```
# Replace the above gmake command with the following to
# generate code with profiling enabled.
```

▼ To Bypass the `rlp` Operation

- **To bypass the `rlp` operation (that is, receive --> transmit without `rlp_process` operation), uncomment the following line from `Makfile.ipge` or `Makefile.nxge` for 1Gb PCIe ethernet adapter or Sun multithreaded 10 Gb and 1Gb PCIe Ethernet adapter, respectively:**

  `-DIPFWD_RAW`

Note – This action disables the RLP processing operation *only,* the queues are still used. This is not the default option.

▼ To Use One Global Memory Pool

By default, the RLP application uses a single global memory pool for all the DMA channels.

1. **Enable the single memory pool by using the following flag:.**

   `-DFORCEONEMPOOL`

2. **Update the `rlp_swarch.c` file to use individual memory pools.**

▼ To Run RLP on Four Ports for `ipge`

By default, the RLP application is configured for two port-pairs for the `ipge` driver.

- **To run the application on eight `ipge` interfaces (four port-pairs), enable the following flag:**

  `-DRLP_RUN_ALL_THDS`

RLP Policy for Spreading Traffic to Multiple DMA Channels

You can specify a policy for spreading traffic into multiple DMA flows by hardware hashing or the TCAM lookup table. TABLE 9-4 describes each policy:

TABLE 9-4 RLP Policy Descriptions

Name	Description
`hash_ip_addr`	Hash on IP destination and source addresses.
`hash_ip_da`	Hash on IP destination address.
`hash_ip_sa`	Hash on IP source address.
`hash_vlan_id`	Hash on VLAN ID.
`hash_portnum`	Hash on physical MAC Port number
`hash_l2da`	Hash on L2 destination address.
`hash_proto`	Hash on protocol number.
`hash_src_port`	Hash on L4 source port number.
`hash_dst_port`	Hash on L4 destination port number.
`hash_all`	Hash on all of the above fields.
`tcam_classify`	Perform TCAM lookup.

▼ To Enable an RLP Policy

- **Add the following into the `gmake` line:**

  `FLOW_POLICY=`*policy*

  where *policy* is one of the above specified policies.

  For example, to enable flow on an IP destination and source address, add `FLOW_POLICY=hash_ip_addr` to the `gmake` line as shown below:

```
gmake -f Makefile.nxge NETHW_TYPE=2 CMT=N2 FLOW_POLICY=HASH_IP_ADDR
```

  If none of the policies listed in TABLE 9-4 are specified, a default policy is given. When you use the default policy, all `L2/L3/L4` header fields are used for spreading traffic.

IPSec Gateway Reference Application

The IPSec Gateway Reference Application implements the IP Encapsulating Security Payload (ESP) protocol using Tunnel Mode. This application allows two Gateways (or a host and a gateway) to securely send packets over an unsecure network with the original IP packet tunneled and encrypted (privacy service). This application also implements the optional integrity service allowing the ESP header and tunneled IP packet to be hashed on transmit and verified on receipt.

IPSec Gateway Reference Application Architecture

The design calls for six Netra DPS threads in a classic architecture where four threads are dedicated to packet reception and transmission (two receivers, two senders). In this architecture, a thread takes plain text packets and encapsulates and encrypts them, as well as a thread that de-encapsulates and decrypts. The architecture is shown in FIGURE 9-1.

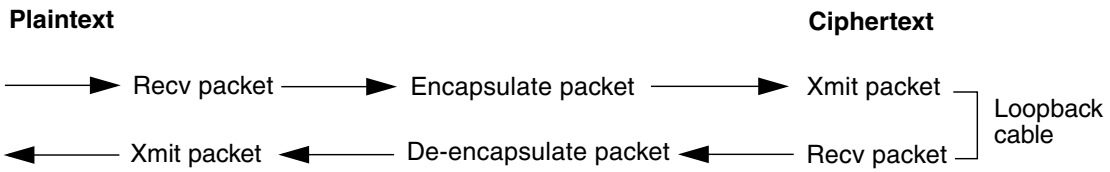


FIGURE 9-1 IPSec Gateway Reference Application Architecture

For testing purposes, this architecture uses four interfaces (NICS) as shown in TABLE 9-5.

TABLE 9-5 IPSec Gateway Reference Application Architecture

Interface	Description
NIC 0	Plaintext packets coming from the traffic generator.
NIC 1	IPSec-encapsulated ciphertext packets going to the peer Gateway (in this case it is a loopback cable going to NIC 2).
NIC 2	For testing purposes, this takes the packets sent over NIC 1 which are looped-back through a loopback cable. These packets are ciphertext. Normally, the packets come in on the NIC 1 interface from the peer gateway. Therefore, there is code that "reverses" source/destination addresses and port numbers to simulate an actual response packet.
NIC 3	IPSec de-encapsulated packets being "returned" to the host. These are plaintext packets that would normally be sent back over NIC 0.

Two extra packet receive threads are also used to listen on NIC 1 and NIC 3 where the test scenario above receives no packets. The main purpose of these threads are to assist with freeing TX buffers (see *Netra Data Plane Software Suite 2.0 Reference Manual* regarding the Netra DPS ethernet API).

Refer to Request for Comments (RFC) documents for a description of IPSec and the ESP protocol:

- 4301 – Security Architecture for the Internet Protocol
- 4303 – IP Encapsulating Security Payload (ESP)

The IPSec RFC refers to outbound and inbound packets. These design notes refer to these terms.

- Outbound packets are those coming into the IPSec Gateway as plaintext (from the unprotected hosts) and being sent to the peer gateway as ciphertext packets (encrypted).
- Inbound packets are the opposite, that is, IPSec-encapsulated (ciphertext) packets coming in from the peer gateway and being decrypted and sent to the unprotected hosts.

IPSec Gateway Reference Application Capabilities

IPSec is a complex protocol. This application handles the following most common processing:

- Static Security Association Database (SADB)

  Contains the type of service to provide (privacy, integrity), crypto and hashing types and keys to be used for a session, among other housekeeping items. An item in the SADB is called a Security Association (SA). An SA can be unique to one connection, or shared among many.

- Static Security Policy database (SPD)

  A partial implementation that is used to contain "selectors" that designate what action should be taken on a packet based on the source and destination IP addresses, protocol, and port numbers.

- SPD cache

  A critical cache used to quickly look up the SA to use for packets coming from the plaintext side. The packet source and destination addresses and ports are hashed to find the action to take on the packet (discard, pass-through, or IPSec protect) and the SA.

- Security Parameter Index (SPI) hash

  A critical cache used to quickly find an SA for ESP packets entering the system from the ciphertext side. The Security Parameter Index is in the ESP header

- ESP Protocol, Tunnel Mode

  This IPSec implementation uses the ESP protocol (it does not currently handle AH, though ESP provides most of the AH functionality). Tunnel Mode is used to encapsulate (tunnel) IP packets between hosts and interface to a peer gateway machine.

- Privacy Service
  - Encrypt/Decrypt traffic
  - Tested Algorithms: AES EBC, CBC 128 and 256 bit
- Integrity Service
  - Authenticate via optional hashing
  - Tested Algorithms: HMAC-SHA256

High-Level Packet Processing

The following describes functions of outbound and inbound packet processing:

Outbound Packets

- Receive packet (from NIC 0)
- Hash the source/destination IP and ports
- Look up in (Security Policy Database caches (SPD-cache) to determine action to take and a pointer to the Security Association (SA)
- If action is IPSec-protect:
  - Build (prepend) outer IP header, ESP header
  - Encrypt payload (original IP packet) using security parameters in SA
  - Optionally calculate and add a hash value
  - Xmit ciphertext packet over NIC 1

Inbound Packets

- Receive packet (from NIC 2)
- If action is an ESP packet:
  - hash Security Parameter Index (SPI) from ESP header to obtain SA
  - Optionally hash and verify hash value (integrity service)
  - Decrypt payload
  - Remove outer IP header, ESP header and trailer
  - Xmit plaintext packet over NIC 3

Security Association (SA) Database and Security Policy Database

The packet encapsulation and encryption code is straight-forward once you have a pointer to the SA. The SA contains the following information:

- Crypto algorithm to use (AES, 3DES, and so on)
- Key length
- Key
- IV
- Type of service to apply (privacy-only or privacy + integrity)
- Hash algorithm (SHA, etc)
- Hash length
- Hash key
- Sequence number
- etc

Packet encapsulation and de-encapsulation is just a matter of determining where the new IP header goes or where the original IP header is, building the new IP header, and invoking the crypto APIs on the correct packet location and length. For the IPSec implementation, you need to find the SA to use when a packet is received (either outbound on inbound). You must use software hashing and hash table lookups for every packet. Note that when this is ported to Sun Multithreaded 10G Ethernet on PCIe, the packet classification features speed-up this hashing.

The following sections describe how the SA is obtained for each packet.

Outbound Packets and Inbound Packets

The following sections describe how the SA is obtained for each packet.

Outbound Packets

You must look at the packet selectors to determine what action to take (either DISCARD, PASS-THROUGH as is, or PROTECT). The selectors are the source and destination IP addresses, the source and destination ports, and the protocol (TCP, UDP, and so on).

The action to take is stored in the Security Policy Database (SPD). For this application, the complete SPD is not implemented. A static SPD exists that consists of rules that must be searched in order using the packet selectors.

For each selector (source IP, destination IP, source port, destination port, protocol), the rule states one of the following:

- Single value (for example, matches on source addr of 129.1.2.3)
- List of values (for example, matches either 129.1.2.3, 129.1.2.5, 129.1.2.10)
- Range of values (for example, 129.1.1.3 - 129.1.1.10)
- Match-all (for example, any source port)
- Mask (for example, matches any source addr after applying mask 0x3F)

If all selectors match the rules, use the SP entry to determine what action to take. If it is `PROTECTED` (IPSec), the inbound and outbound Security Parameter Index (SPI) knows which SA to use. This implies the following:

- An SA can be exclusive to a given connection
- An SA can be shared among many connections (for example, a single SA can be used to protect all traffic between to hosts)
- Each "connection" or flow of traffic has two SAs: one for outbound traffic and one for inbound traffic. Since a loopback cable is used to receive the ciphertext packets for the Xmit, use an SPI that is the SPI of the outbound packet plus 1.

The last rule in the SPD should be a catch-all that says `DISCARD` the packet.

The SPD structures and definitions can be found in `spd.h`

The source code for the SPD can be found in `spd.c`

The function used to lookup a rule is `SPD_Search()`, which is passed the selector values from the packet.

The above lookup is complex for every packet. Because of this, a cache called the SPD-Cache is maintained. The first time you lookup a particular connection, create a SPDC structure, hash the selectors, and place this SPDC in a hash table.

When a packet using the exact combination of selectors comes in, you first hash it (using `SPDC_HASH()`) and then look it up in the SPDC hash table. If found, immediate access to the SA is made.

The definitions for this SPDC can be found in `sadb.h`. The functions can be found in `sadb.c`.

This application does not hash on the protocol and assumes that it is either UDP or TCP because there are source and destination ports.

The SPDC hash table is defined as:

```
spdc_entry_t *spdc_hash_table[SPDC_HASH_TABLE_SIZE];
```

The primary function used to lookup an SPDC entry is:

```
spdc_e *spdc_hash_lookup_from_iphdr(iphdr)
```

For this hash table, take the hash value, mask off the hash table size - 1, then index into this table to get an entry. YOu then have to compare the entry for a match, and if it is not a match, walk the chain until you find one.

Inbound Packets

Inbound IPSec packets contain an ESP header with an SPI. Take the SPI, hash it using `SPI_HASH_FROM_SPI()`, look it up in the SPI hash table, and access the SA pointer from there. You cannot use the same hashing as done for outbound packets since the selectors (source/destination IP address and ports) have been encapsulated and encrypted. Decrypt cannot be done until the SA is looked up.

The SPI has table is defined as:

```
spi_entry_t *spi_hash_table[SPI_HASH_TABLE_SIZE];
```

Static Security Policy Database (SPD) and Security Association Database (SAD)

For the purposes of the application, statically define the test SPD and SAD in compile-time initialized C-code in the following C file: `sa_init_static_data.c`

SPD

Two SPD rules are defined.

- The first rule appears as shown below:

```
sp_t sp_rule1 = {
    1,                      /* rule # */
    SA_OUTB1,               /* outb_spi */
    SA_INB1,                /* inb_spi */
    IPSEC_PROTECT,          /* action */
    SPD_PROTOCOL_MATCH_ALL, /* match on all protocols */
    { SPD_MATCH_ALL },      /* match all connections for now */
    { SPD_MATCH_ALL },
    { SPD_SINGLETON, 0, {6666} },   /* Only match UDP ports 6666, 7777 */
    { SPD_SINGLETON, 0, {7777} },   /* Only match UDP ports 6666, 7777 */
};
```

This rule matches any source or destination IP address and protocol (TCP or UDP), and a source port of 6666 and a destination port of 7777. The load generator is set to send UDP packets with those ports. This needs to be changed if other ports are used.

- The second rule matches everything else and the action is set to `IPSEC_DISCARD`, which means drop the packet.

These rules are added to the SPD at init-time (`init_ipsec()` calls `sa_init_static_data()`) through the following call: `SPD_Add()`

Two other functions are defined but not currently used: `SPD_Delete()` and `SPD_Flush()`.

SAD

The SAD is also statically defined in `sa_init_static_data.c`. There are currently two SA entries: one for the outbound SA and one for the inbound SA. Only the outbound SA needs to be defined as the inbound is just a copy except for the SPI.

To test out various encryption and hashing scenarios, this SA entry is where you need to make changes, as shown below:

```
sa_t sa_outb1 = {               /* First outbound SA */
        (void *)NULL,           /* auth ndps cctx */
        (void *)NULL,           /* encr ndps cctx */
        SA_OUTB1,               /* SPI */
        1,                      /* SPD rule # */
        0,                      /* seq # */
        0x0d010102,             /* local_gw_ip */
        0x0d010103,             /* remote_gw_ip */
        {{0x0,0x14,0x4f,0x3c,0x3b,0x18}},       /* remote_gw_mac */
        PORT_CIPHERTEXT_TX,     /* local_gw_nic */
//#define INTEGRITY
#ifdef INTEGRITY
        IPSEC_SVC_ESP_PLUS_INT, /* service type */
#else
        IPSEC_SVC_ESP,          /* service type */
#endif
        IPSEC_TUNNEL_MODE,      /* IPSec mode */
        0,                      /* dont use ESN */

        (int)NDP_CIPHER_AES128, /* encr alg */
        (int)NDP_AES128_ECB,    /* encr mode */
        /*(int)NDP_AES128_CBC,  /* encr mode */
        128/8,                  /* encr key len */
        0/8,                    /* encr IV len */
        16,                     /* encr block len */

        (int)NDP_HASH_SHA256,   /* auth alg */
        0,                      /* auth mode */
        256/8,                  /* auth key len */
        256/8,                  /* auth hash len - will get a default */

        {{TEST_ENCR_KEY_128}},  /* encr key */
        {{TEST_AUTH_KEY_256}},  /* auth key */
        //{{TEST_ENCR_IV_128}}, /* encr IV */
        {{'\000'}},             /* auth IV  - will get a default*/
        /* everything else is dynamic and does not need initing here */
```

The first element to note is the service type. If you want to test privacy (encryption), leave `INTEGRITY` commented out. No hashing will be done. If you want hashing, comment in the `#define` for `INTEGRITY`.

The next fields you might change are the encryption parameters: `encr alg`, `encr mode`, `encr key len`, `encr IV len`, `encr block len`, and the `encr` key. The IV is only used for certain modes, such as CBC for AES.

Be sure to put the proper key lengths and IV lengths in the table.

You might then need to modify the hashing algorithms in a similar manner assuming you chose `INTEGRITY`.

Eventually, the SPD and SAD need to be integrated with a Control Plane (CP) such that the CP determines the static databases. There are two scenarios on how this takes place: download the tables and shared memory.

Download the Tables

The CP uses the LDoms IPC mechanism to interface with Netra DPS to download (add) or modify the SPD and SA. Some functionality already exists to build these databases once the protocol is defined:

- `SPD_Add()`
- `SPD_Delete()`
- `SPD_Flush()`
- `SADB_ADD()`

Shared Memory

The CP sets up the tables in memory that is accessible from both the CP and Netra DPS and informs the Netra DPS application of updates through the LDoms IPC mechanism.

Packet Encapsulation and De-encapsulation

The main packet processing functions are called from the two processing threads which reside in `ipsecgw.c`.

The main plaintext packet processing thread is called `PlaintextRcvProcessLoop()` and it pulls a newly received packet off of a Netra DPS fast queue and calls:

```
IPSEC_Process_Plaintext_Pkt(mblk)
```

The main ciphertext packet processing thread is called `CiphertextRcvProcessLoop()` and it takes a packet off a fast queue and calls:

```
IPSEC_Process_Ciphertext_Pkt(mblk)
```

You can find the `IPSEC_Process_Plaintext_Pkt()` and `IPSEC_Process_Ciphertext_Pkt()` functions in:

```
ipsec_processing.c
```

The following two functions perform the hashing and invoke the actual processing code:

- `IPSEC_ESP_Encapsulate()`
- `IPSEC_ESP_Deencapsulate()`

The message block (`mblk`) contains pointers to the start and end of the incoming packets (`b_rptr` and `b_wptr`). Because plaintext packets must be prepended with a new outer IP header and ESP header, do not shift the incoming packet data down which is a copy. Therefore, when the ethernet driver asks for a new receive buffer through `teja_dma_alloc()`, a buffer is grabbed from the receive buffer Netra DPS memory pool. The memory pool is 2KB and returns an offset into that buffer which tells the driver where to place the packet data. This offset is set to 256 (`MAX_IPSEC_HEADER`), which is enough space to prepend the IPSec header information.

Packet Encapsulation

This section contains notes on how to calculate the location of the various parts of the ESP packet (outbound and inbound).

Outbound

```
Orig:
    OrigIPStart
    OrigIPLen (from original IP header, includes IP hdr + tcp/udp hdr + payload)
New:
    ETH_HDR_SIZE:       14
    IP_HDR_SIZE:        20
    ESP_HDR_FIXED:       8 (SPI + Seq#)
    EncIVLen:           variable - from SA or cryp_ctx
    EncBlkSize:         variable - from static structs
    AuthICVLen:         variable - from SA or cryp_ctx

    ESPHdrLen   = ESP_HDR_FIXED + EncIVLen
    ESPHdrStart = OrigIPStart - ESPHdrLen
    NewIPStart = OrigIPStart - (ETH_HDR_SIZE + IP_HDR_SIZE + ESP_HDR_FIXED +
                                EncIVLen)
    CryptoPadding = OrigIPLen % EncBlkSize
    ESPTrailerPadLen = 4
```

```
    HashStart = ESPHdrStart
    HashLen = ESPHdrLen + OrigIPLen + CryptoPadding + ESPTrailerPadLen

    CryptoStart = OrigIPStart
    CryptoLen = OrigLen + CryptoPadding + ESPTrailerPadLen

    NewIPLen = IP_HDR_SIZE + HashLen + AuthICVLen

NewPktStart---->0               1
                +---------------+
                |EtherHDR       |
                +---------------+
NewIPStart----->14              15
                +---------------+
                |IP HDR         |
                +---------------+
ESPHdrStart---->32              33
HashStart       +---------------+<====== to be hashed from here
                |ESP HDR        |
                +---------------+
                40              41
OrigIPStart---->+---------------+<====== to be crypted from here
                | Orig IP HDR   |
                +---------------+
                .
                .
                .
CryptoLen       +---------------+=== OrigIPLen + CryptoPadLen +
                                                    ESP_TRAILER_FIXED


ICVLoc--------->+---------------+=== HashStart + HashedBytesLen
HashedBytesLen                   === ESPHdrLen + OrigIPLen + CryptoPadLen +
                                                    ESP_TRAILER_FIXED;

        NDPSCrypt(OrigIPStart, CryptoLen)
        NDPSHashDirect(ICVLoc, HashStart, HashedBytesLen)
```

Inbound

```
OrigIPStart
OrigIPLen (from original IP header, includes IP hdr + tcp/udp hdr + payload)
HashStart = OrigIPStart + IP_HDR_SIZE
HashLen = OrigIPLen - (IP_HDR_SIZE + AuthICVLen)

CryptoStart = HashStart + ESP_HDR_FIXED + EncIVLen
CryptoLen = HashLen - (ESP_HDR_FIXED + EncIVLen)

PadOffset = HashStart + HashLen - 2
PadLen = *PadOffset

NewIPStart = CryptoStart
NewIPLen = same as tunneled IPLen - get from IP header
```

Memory Pools

The IPSec Gateway uses the Netra DPS memory pools shown in table. The names and sizes are defined in `ipsecgw_config.h`:

TABLE 9-6 Netra DPS Memory Pools

Memory Pool	Description
`SPDC_ENTRY_POOL`	Pool for SPDC entries stored in the SPDC hash table.
`SPI_ENTRY_POOL`	Pool for SPI entries stored in the SPI hash table. These hash tables are actually arrays indexed by the hash value (masked with the hash table size).
`SP_POOL`	Pool of SP entries.
`SA_POOL`	Pool of SA entries.
`CRYP_CTX_POOL`	Crypto context structures (maintained by the Crypto API library).

Pipelining

The two main processing threads (`PlaintextRcvProcessLoop` and `CiphertextRcvProcessLoop`) are pipelined into another thread each: one to do most of the packet encapsulation and de-encapsulation, and one to do the encryption and decryption and optional hashing.

An extra fast queue was inserted in each path, therefore, the pipeline for the eight threads are as shown below:

```
PlaintextRcvPacket ->
        PlaintextRcvProcessLoop ->
                EncryptAndHash ->
                        CiphertextXmitPacket ->   NIC 1  ---->
                                                                LOOPBACK
                        <- CiphertextRcvPacket <- NIC 2  <----
                <- CiphertextRcvProcessLoop
        <- HashAndDecrypt
PlaintextXmitPacket
```

The two new threads (`EncryptAndHash` and `HashAndDecrypt`) reside in `ipsec_processing.c` rather than `ipsecgw.c` where the other threads reside.

The packet processing portion of this pipeline must pass the packet to the crypto part of the pipeline. Packets are normally passed on fast queues through the `mblk` pointer. Other vital information also needs to be passed, such as the SA pointer. Rather than allocation of a new structure to pass the data and the `mblk`, this data is piggy-back at the beginning of the receive buffer, which is not used. Refer to the `cinfo` structure defined in `ipsec_processing.c`.

Source Code File Description

The IPSec package comes with the following directories:

- `</opt/SUNWndpsc>/src/apps/ipsec-gw-ipge`

  This directory consists of IPSec code that supports the `ipge` (Ophia 1G Ethernet)

- `</opt/SUNWndpsc>/src/apps/ipsec-gw-nxge`

  This directory consists of IPSec code that supports the Sun multithreaded 10G Ethernet on PCI-E with NIU or on-chip (NIU)).

- `</opt/SUNWndpsc>/src/libs/ndps_crypto_api`

  This directory consists of Crypto API that interface to the Crypto hardware.

- `</opt/SUNWndpsc>/src/libs/ipsec`

  This directory consists of IPSec library functions.

The file descriptions in the following tables are based on the files in the `ipsec-gw-nxge` directory.

TABLE 9-7 lists the build scripts.

TABLE 9-7 Build Scripts

Build Script	Description
`build_qgc`	Build for Sun Multi-thread Quad Gigabit Ethernet.
`build_10g_niu`	Build for Sun Multi-thread 10G Ethernet.
`build_niu_multi`	Build for Sun Multi-thread 10G Ethernet with multi instances configuration.
`build_ipcrypto`	Build for Crypto configuration only.

TABLE 9-8 lists the source files.

TABLE 9-8 Source Files

Source File	Description
`common.h`	Header file consists of common information.
`config.h`	Consists of receive buffer configuration information.
`debug.c`	Used when compiling in `DEBUG` mode (see `IPSEC_DEBUG` in the Makefile to turn on IPSec debugs). This file contains the debug thread that calls `teja_debugger_check_ctrl_c`().
`init.c`	Main initialization code called by Netra DPS runtime for setting up fast queues and initializing the Crypto library and the IPSec code.

TABLE 9-8 Source Files *(Continued)*

Source File	Description
`init_multi.c`	Main initialization code called by Netra DPS runtime for setting up fast queues used by the IPSec multiple instances code.
`ip_crypto.c`	Location of the main application threads for the IPSec crypto (Crypto only, no IPSec overhead).
`ipsec_niu_config.c`	Assists user to map application tasks to CPU core and hardware strands of the UltraSPARC T2 chip specific to the NIU (Network Interface Unit of the UltraSPARC T2 chip) configuration.
`ipsecgw.c`	Contains the main application threads.
`ipsecgw_config.c`	Assists user to map application tasks to CPU core and hardware strands.
`ipsecgw_flow.c`	Contains the classification flow entry.
`ipsecgw_flow.h`	Contains the definitions of the classification flow.
`ipsecgw_impl_config.h`	Contains the information related to `mblk`, receive buffer sizes, number of channels, SA, SPDC.
`ipsecgw_niu.c`	Main application thread for the NIU configuration.
`ipsecgw_niu_multi.c`	Main application thread for the NIU multi-instances configuration.
`lb_objects.h`	Contains memory pool definitions.
`mymalloc.c`	Used by the low-level crypto-code.
`mymalloc.h`	Memory pool definitions used by Crypto library.
`n2profile.c`	Contains the profiling code.
`n2profile.h`	Contains the profiling definitions.
`perf_tools.c`	Used for profiling (not available on UltraSPARC T2).
`perf_tools.h`	Used for profiling (not available on UltraSPARC T2).
`rx.c`	Packet receive code which uses ethernet API.
`skeleton.c`	Code needed to force Netra DPS runtime to include the `malloc` and `free`.
`stats.c`	Functions to collect statistics.
`tx.c`	Packet xmit code which uses ethernet API encryption and hashing algorithms.

TABLE 9-8 Source Files *(Continued)*

Source File	Description
`user_common.c`	Contains the callback functions used by the Netra DPS ethernet APIs.
`user_common.h`	Contains fast queue definitions and function prototypes.
`util.c`	Contains IPSec utility functions.

TABLE 9-9 lists the IPSec library files.

TABLE 9-9 IPSec Library Files

IPSec Library File	Description
`init_ipsec.c`	Code that is called at startup to initialize IPSec structures.
`ipsec_common.h`	Function prototypes, some common macros, other definitions.
`ipsec_defs.h`	IPSec protocol definitions and macros.
`ipsec_proc.c`	This is the main IPSec processing code. This is where all the encapsulation-encryption, de-encapsulation-decryption and hashing functions reside.
`ipsec_processing.c`	The `ipsec_proc.c` corresponding file for supporting `ipge` (Ophia 1G Ethernet).
`ipsecgw.c`	Major IPSec top level functions.
`netdefs.h`	Constant and macro definitions of common ethernet and IP protocols.
`sa_init_static_data.c`	Contains the statically-defined SAD and SPD. This is the file to modify for testing various SA configurations.
`sadb.c`	SADB functions.
`sadb.h`	SADB definitions.
`spd.c`	SPD functions.
`spd.h`	SPD definitions.

TABLE 9-10 lists the Crypto library files.

TABLE 9-10 Crypto Library Files

Crypto Library File	Description
`crypt_consts.h`	Contains various crypto constants.

TABLE 9-10 Crypto Library Files *(Continued)*

Crypto Library File	Description
`ndpscript.c`	Contains Crypto API implementations.
`ndpscrypt.h`	Contains data structures and function prototypes.
`ndpscrypt_impl.h`	Contains Crypto Context structure.

Each UltraSPARC T2 processor CPU core has a crypto unit. This unit leverages the Modular Arithmetic Unit and the Hash and Cipher Unit inside the SPU to accelerate crypto operations. There are a total of eight crypto units in an eight-core system.

Reference Applications Configurations

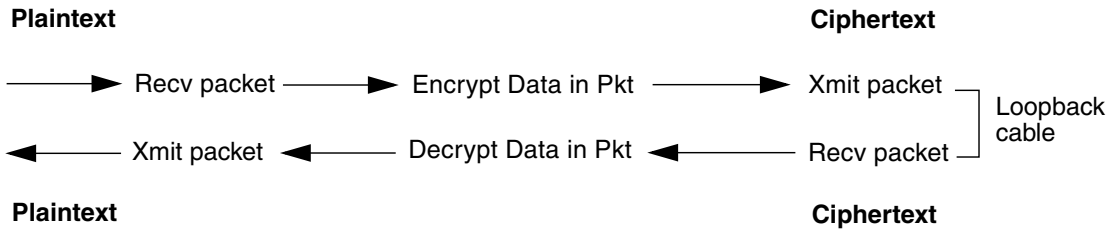
IPSec and Crypto has four reference application configurations:

- IP with Encrypto and Decrypto
- IPSec Gateway on Quad GE
- IPSec Gateway on NIU 10G Interface (One Instance)
- IPSec Gateway on NIU 10G Interface (Multiple Instances)

IP with Encrypto and Decrypto

This application tests how fast the pure accessing Crypto Engine can perform.

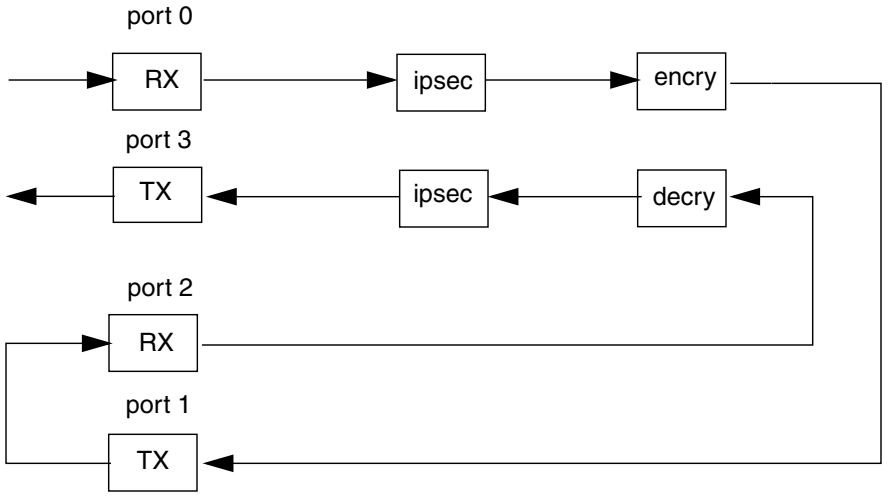
Build script: `build_ipcrypto`



IPSec Gateway on Quad GE

This application is the IPSec Gateway prototyped using the Sun multithreaded Quad Giga-bit Ethernet card.

Build script: `build_qgc`

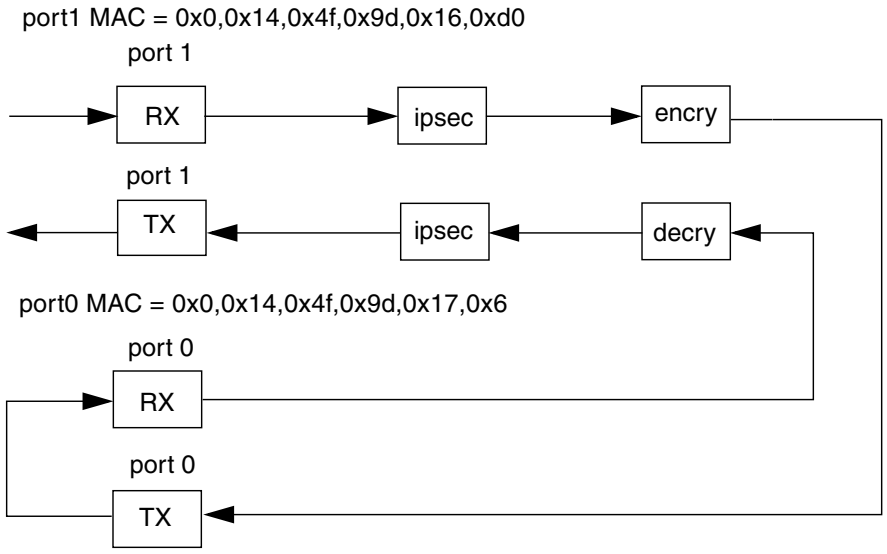


IPSec Gateway on NIU 10G Interface (One Instance)

This application is the IPSec Gateway prototyped using the Sun multithreaded UltraSPARC T2 on-chip NIU.

Build script: `build_10g_niu`

The following is an example of one instance of the IPSec Gateway Application on the Sun multithreaded 10GbE with NIU interface.



Notes

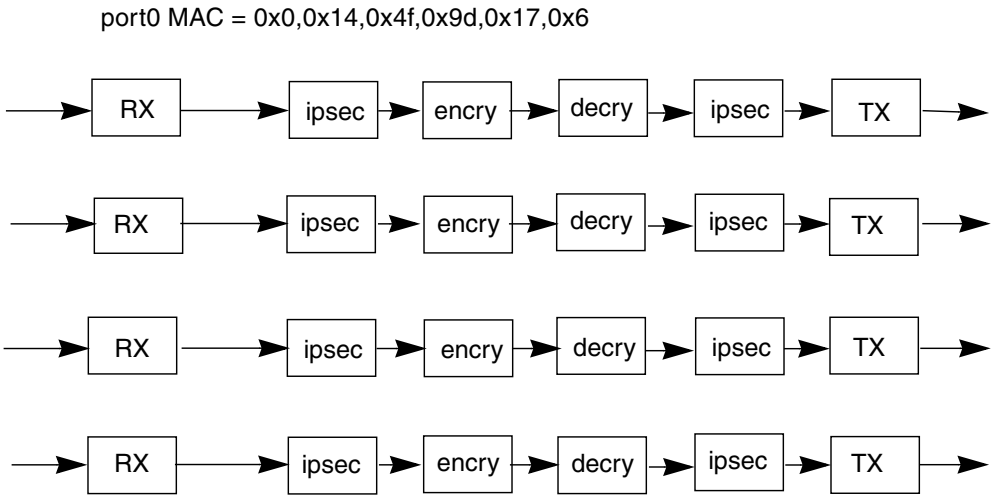
- If your connection is like that above, then you need to configure the traffic generator:
  - FrameBuilder: select IPv4 + UDP
  - DA MAC: 0x0,0x14,0x4f,0x9d,0x16,0xd0
  - IPv4: SA=69.235.4.0 DA=69.235.4.1
  - UDP: SP=6666 DP=7777; this has to be consistent with sp_rule1 in src/libs/ipsec/sa_init_static_data.c
  - Payload: Fill Pattern = 55

IPSec Gateway on NIU 10G Interface (Multiple Instances)

This application simulates multiple flow of IPSec, encrypt, and decrypt instances. It is designed to measure IPSec and Crypto overhead.

Note – This setting of the traffic generator applies to the Sun SPARC Enterprise T5120/T5220 systems. For Netra ATCA CP3260 systems, see "Flow Policy for Spreading Traffic to Multiple DMA Channels" on page 164).

The following is an example of multiple instances of the IPSec gateway application on the Sun multithreaded 10GbE with NIU interface.



Notes

For Sun SPARC Enterprise T5220 system (UltraSPARC T2 server) configurations:

- If your connection is like that above, then you need to configure the traffic generator:
  - FrameBuilder: select IPv4 + UDP
  - DA MAC: 0x0,0x14,0x4f,0x9d,0x17,0x6
  - IPv4: SA=69.235.4.0 DA=69.235.4.1
  - UDP: SP=6666 DP=7777; this has to be consistent with `sp_rule1` in `src/libs/ipsec/sa_init_static_data.c`
  - Payload: Fill Pattern = 55

- In FrameBuilder, insert VLANtag L2 checked
  - Enable VLAN1-VLAN ID: Header select VLAN1
  - Field select VLAN ID, step num = 1
  - Start 25 End 28
- In `src/libs/ipsec/sa_init_static_data.c`:

  `sa_outb1 remote_gw_mac` must be {{0x0,0x14,0x4f,0x9d,0x17,0x6}}
- In `src/apps/ipsec-gw-nxge/src/app/ipsec_niu_config.c:`

  If it is port 0, do not do anything; if it is port 1, then add the following:

  ```
  ..., OPEN_OPEN, NXGE_10G_START_PORT+1, ...
  ```

Notes

For Netra ATCA CP3260 system (UltraSPARC T2 ATCA Blade) configurations:

- If your connection is like that above, then you need to configure the traffic generator:
  - FrameBuilder: select IPv4 + UDP
  - DA MAC: 0x0,0x14,0x4f,0x9d,0x17,0x6
  - IPv4: SA=69.235.4.0 DA=69.235.4.1

Note – You need to select the SA IP to have COUNT 255 and step = 0.0.0.1 so that it will increment the SA IP from 0.0.0.1 to 0.0.0.255 with the step of 0.0.0.1

  - UDP: SP=6666 DP=7777; this has to be consistent with sp_rule1 in `src/libs/ipsec/sa_init_static_data.c`
  - Payload: Fill Pattern = 55
- In `src/libs/ipsec/ipsec_common.h`:

  If it is port 0, do not do anything; if it is port 1, then set the following:
  ```
  #define PORT_PLAINTEXT_RX    1
  ```
- In `src/libs/ipsec/sa_init_static_data.c`:

  `sa_outb1 remote_gw_mac` must be {{0x0,0x14,0x4f,0x9d,0x17,0x6}}
- In `src/apps/ipsec-gw-nxge/src/app/ipsec_niu_config.c`

  If it is port 0, do not do anything; if it is port 1, then add the following:

  ```
  ..., OPEN_OPEN, NXGE_10G_START_PORT+1, ...
  ```

Flow Policy for Spreading Traffic to Multiple DMA Channels

You can specify a policy for spreading traffic into multiple DMA flows by hardware hashing or TCAM lookup (classification). TABLE 9-11 describes flow policies:

TABLE 9-11 Flow Policies

Flow Policy	Description
HASH_IP_ADDR	Hash on IP destination and source addresses.
HASH_IP_DA	Hash on IP destination address.
HASH_IP_SA	Hash on IP source address.
HASH_VLAN_ID	Hash on VLAN ID.
HASH_PORTNUM	Hash on physical MAC port number.
HASH_L2DA	Hash on L2 destination address.
HASH_PROTO	Hash on protocol number.
HASH_SRC_PORT	Hash on L4 source port number.
HASH_DST_PORT	Hash on L4 destination port number.
HASH_ALL	Hash on all of the above fields.
TCAM_CLASSIFY	Perform TCAM lookup.

▼ To Enable a Flow Policy

- **Add the following into the** `gmake` **line:**

  HASH_POLICY=*policy*

  whereas *policy* is one of the above specified policies

For example, to enable hash on an IP destination and source address, add FLOW_POLICY=HASH_IP_ADDR to the `gmake` line in the build script file `build_niu_multi` Remove # in front of FLOW_POLICY=HASH_IP_ADDR:

```
gmake -f Makefile.niu_multi NEPTUNE_CARD=2 FLOW_POLICY=
HASH_IP_ADDR
```

Note – If you specify FLOW_POLICY=HASH_ALL which is backward compatible with Sun SPARC Enterprise T5120/T5220 systems, all fields are used.

If none of the policies in TABLE 9-11 are specified (do not specify the `HASH_POLICY` in the above `gmake` line, for example, `#HASH_POLICY=HASH_IP_ADDR`, a default policy will be given. When the default policy is used, all level (L2/L3/L4) header fields are used for spreading traffic.

CHAPTER 10

Performance Tuning

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

Topics in include:

Performance Tuning Introduction

The UltraSPARC T series 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 and UltraSPARC T2 processor strand-rich environment can be thought of as symmetric multiprocessing on a chip.

The Lightweight Runtime Environment (LWRTE) provides an ANSI C development environment for creating and scheduling application threads to run on individual strands on the UltraSPARC T series processor. With the combination of the UltraSPARC T series processor and LWRTE, developers have a 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 10-1 shows the UltraSPARC T1 architecture.

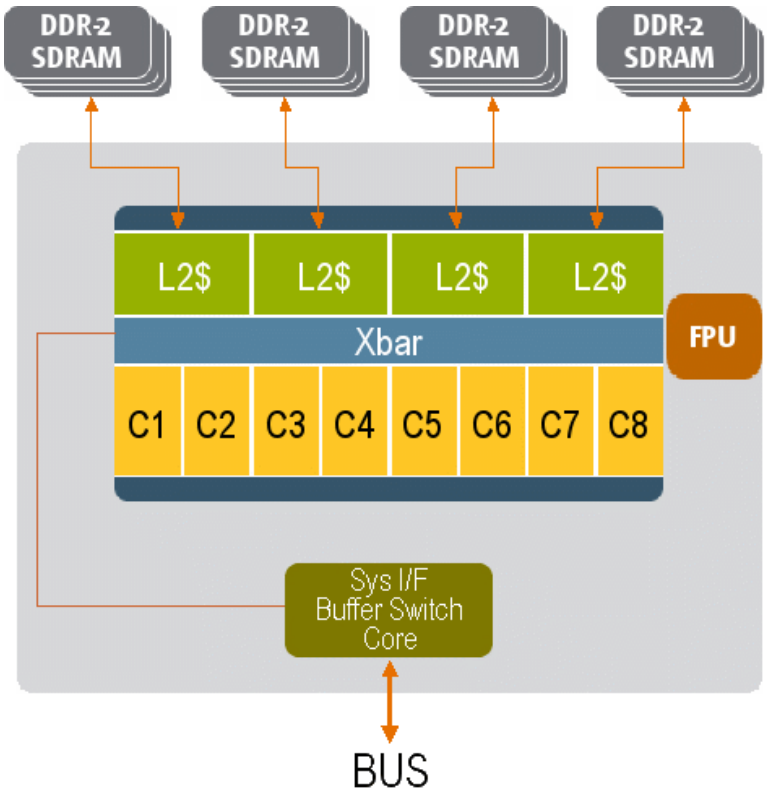


FIGURE 10-1 UltraSPARC T1 Architecture

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 operating system 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. Therefore, 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 second. 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 10-1 summarizes the key performance limits and latencies.

TABLE 10-1 UltraSPARC T1 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	~ 3 cycles
	L2 hit latency	~ 23 cycles
	L2 miss latency	~ 90 ns
	Bandwidth	17 GBps (25 GBps peak)
I/O bandwidth		~ 2 GBps (JBus limitation)

UltraSPARC T2 Processor Overview

The Sun UltraSPARC T2 processor is the second generation of CMT processor. In addition to features found in UltraSPARC T1, UltraSPARC T2 dramatically increases processing power by increasing the number of hardware strands in each core. This processor also increases the floating point performance by introducing one FPU unit per CPU core. The UltraSPARC T2 also includes on-chip 10G Ethernet and crypto accelerator. FIGURE 10-2 shows the UltraSPARC T2 system architecture.

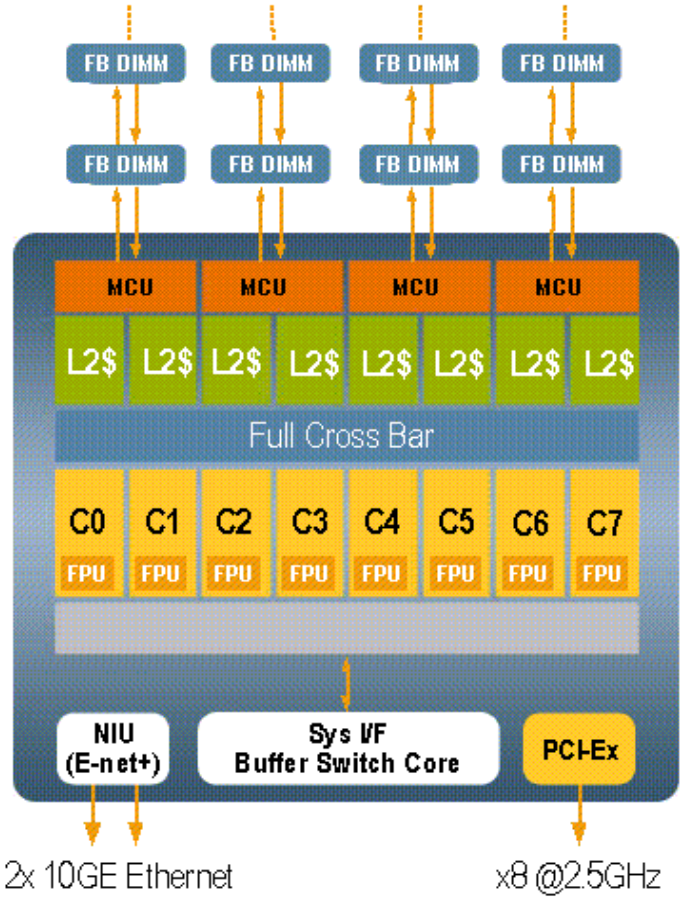


FIGURE 10-2 UltraSPARC T2 Architecture

The processing engine is organized as 8 multithreaded cores, with each core consisting of two independent integer execution pipelines. Each pipeline executes up to 4 strands concurrently. Therefore, the processor has a total of 8 strands per CPU core (64 strands per CPU). The maximum instruction processing rate is 2 instruction/cycle per core or 16 instructions/cycle for the entire 8 core chip. Unlike UltraSPARC T1, in which one FPU is shared by all 8 CPU cores, UltraSPARC T2 has an independent FPU per CPU core.

Similar to UltraSPARC T1, the memory system consists of two levels of on-chip caching and on-chip memory controllers. Each core has separate level 1 instruction and data caches and TLBs. The instruction cache is 16KB, the data cache is 8KB, and the TLBs are 64 entries for instructions (ITLB) and 128 entries for data (DTLB). UltraSPARC T2 has a larger L2 cache compared to its predecessor. The level 2 cache is a 4 Mbyte unified instruction. The cache is 16-way set associative and 8-way banked.

UltraSPARC T2 has doubled the memory capacity of its predecessor. This processor consists of 4 dual-channel FBDIMM memory controllers at 4.8 Gb/sec, capable of controlling up to 256 Gbyte memory per system. Memory bandwidth is increased to 50 Gbyte/sec.

The integrated Network Interface Unit (NIU) provides dual on-chip 10GbE processing capability. All network data is sourced from and destined to memory without having the need to go through the I/O interface. This configuration eliminates the I/O protocol translation overhead and takes full advantage of the high memory bandwidth. The NIU also features line rate packet classification and multiple DMA engines to handle multiple incoming traffic flows in parallel.

Also integrated on-chip is the cryptographic coprocessor, one per CPU core. The Crypto engine facilitates wire-speed encryption and decryption.

UltraSPARC T2 eliminates the JBUS (the I/O bus of the UltraSPARC T1) entirely. I/O is controlled by an on-chip x8 at 2.5 GHz per lane PCIe root complex, providing a total of 3-4 Gbyte/sec I/O bandwidth with maximum payload sizes of 128 bytes to 512 bytes.

TABLE 10-2 summarizes the key performance limits and latencies.

TABLE 10-2 UltraSPARC T2 Key Performance Limits and Latencies

Feeds		Speeds
Processor instruction execution bandwidth		22.4 G instructions/sec (peak@1.4GHz)
Memory		
	L1 hit latency	~ 3 cycles
	L2 hit latency	~ 23 cycles

TABLE 10-2 UltraSPARC T2 Key Performance Limits and Latencies *(Continued)*

Feeds		Speeds
	L2 miss latency	~ 135 ns
	Bandwidth	~ 40 GBytes/sec peak for read ~ 20 GBytes/sec peak for write
I/O bandwidth		3~4 GBytes/sec (PCI-Express)

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. This section discusses UltraSPARC T1 and UltraSPARC T2 performance.

UltraSPARC T1 Performance

In UltraSPARC T1 systems, the I/O limitation of 2 Gbyte per second puts an upper bound on the throughput metric. FIGURE 10-3 shows the packet forwarding rate limited by this I/O bottleneck.

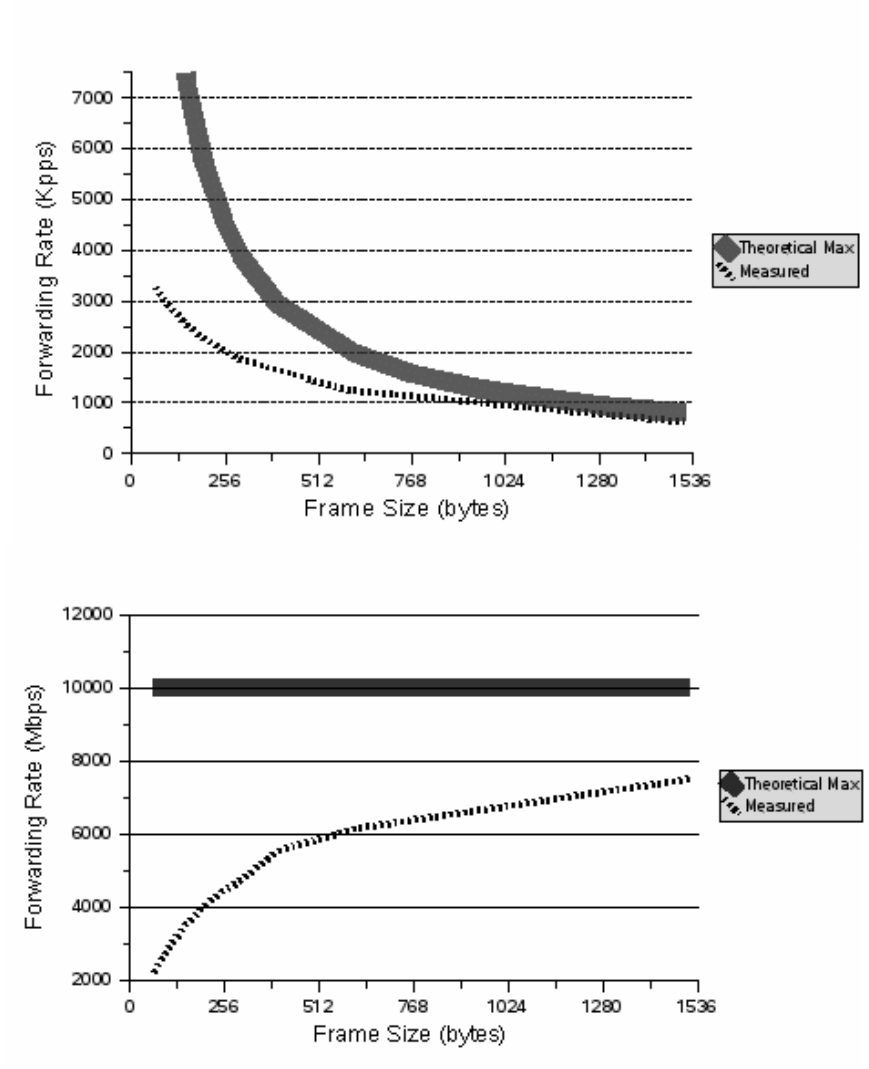


FIGURE 10-3 UltraSPARC T1 Forwarding Packet Rate Limited by I/O Throughput

The theoretical maximum represents the throughput of 10 Gbytes 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. Nothing can be done about the I/O bottleneck, therefore 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 10-4 shows the average maximum number of instructions the processor can execute per packet while maintaining line rate.

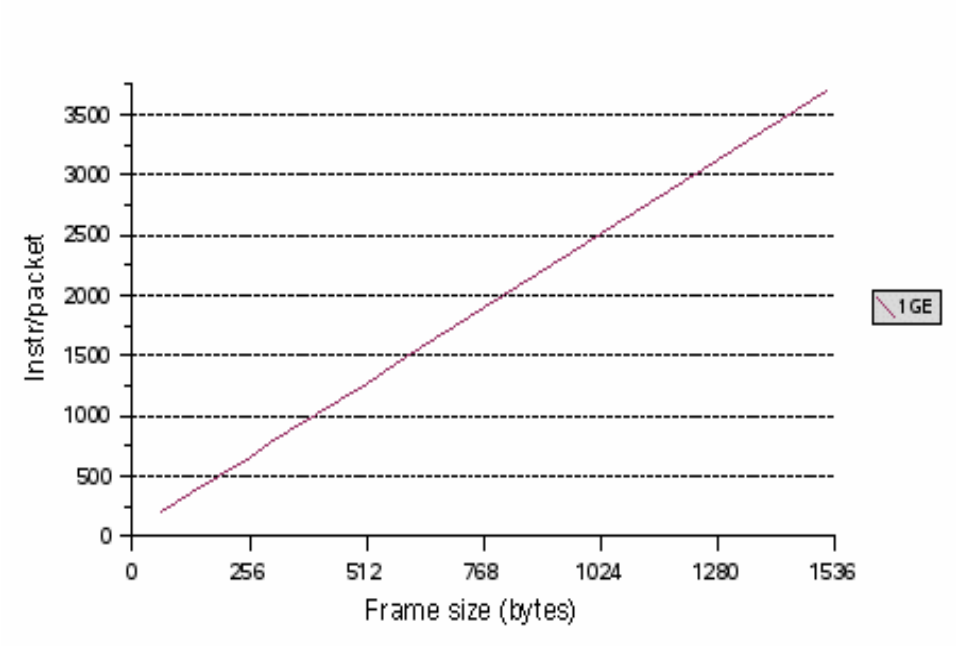


FIGURE 10-4 Instructions per Packet Versus Frame Size

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

UltraSPARC T2 Performance

In UltraSPARC T2 systems, the I/O bandwidth is largely expanded from 2 Gbytes/sec to 3 ~ 4 Gbytes/sec range. This is because the Jbus interface is replaced by the PCI Express interface. The on-chip Ethernet interface substantially improves network performance by removing the entire I/O bus overhead. When the Network Interface Unit (NIU) is utilized, ingress traffic data from input ports enters into memory directly through the DMA engine, and vice versa for egress data. Performance is no longer I/O bound. The next speed bump is determined by the CPU processing power and memory controller capacity. CPU frequency and memory controller capacity on the system platform becomes a factor in determining the maximum packet forwarding rate.

FIGURE 10-5 shows the forwarding packet rate limited by CPU processing power or memory controller bandwidth.

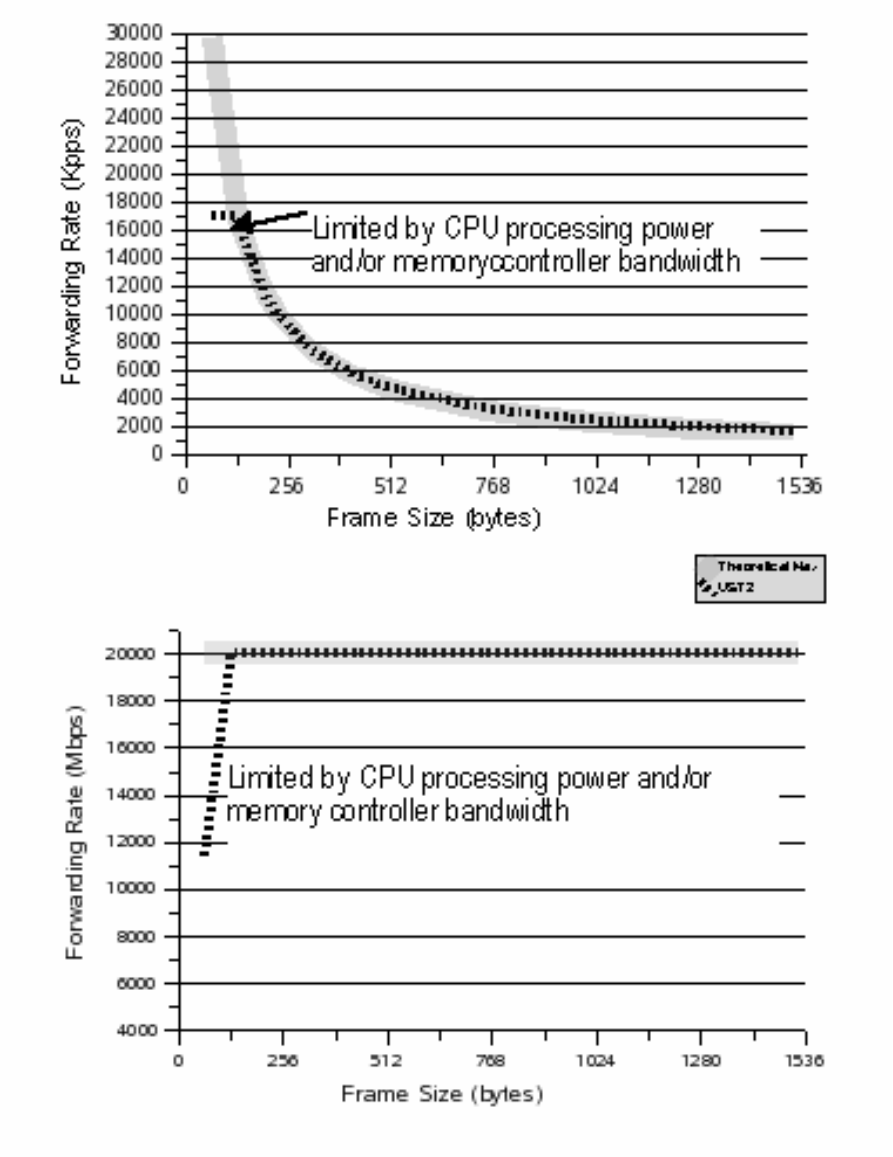

Forwarding Rate (Kpps)
30000
28000
26000
24000
22000
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Limited by CPU processing power
and/or memorycontroller bandwidth
0
256
512
768
1024
1280
1536
Frame Size (bytes)
Forwarding Rate (Mbps)
20000
18000
16000
14000
12000
10000
8000
6000
4000
Limited by CPU processing power and/or
memory controller bandwidth
0
256
512
768
1024
1280
1536
Frame Size (bytes)

FIGURE 10-5 UltraSPARC T2 Forwarding Packet Rate

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 12 compilers provide many optimization flags to tune your application. Refer to the *Sun Studio 12: C Users Guide* for the complete list of optimization flags available. See "Reference Documentation" on page xxii. 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 slow 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 Netra DPS 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 10-6 shows how to split and map an application using fast queues and CMT processor to three strands.

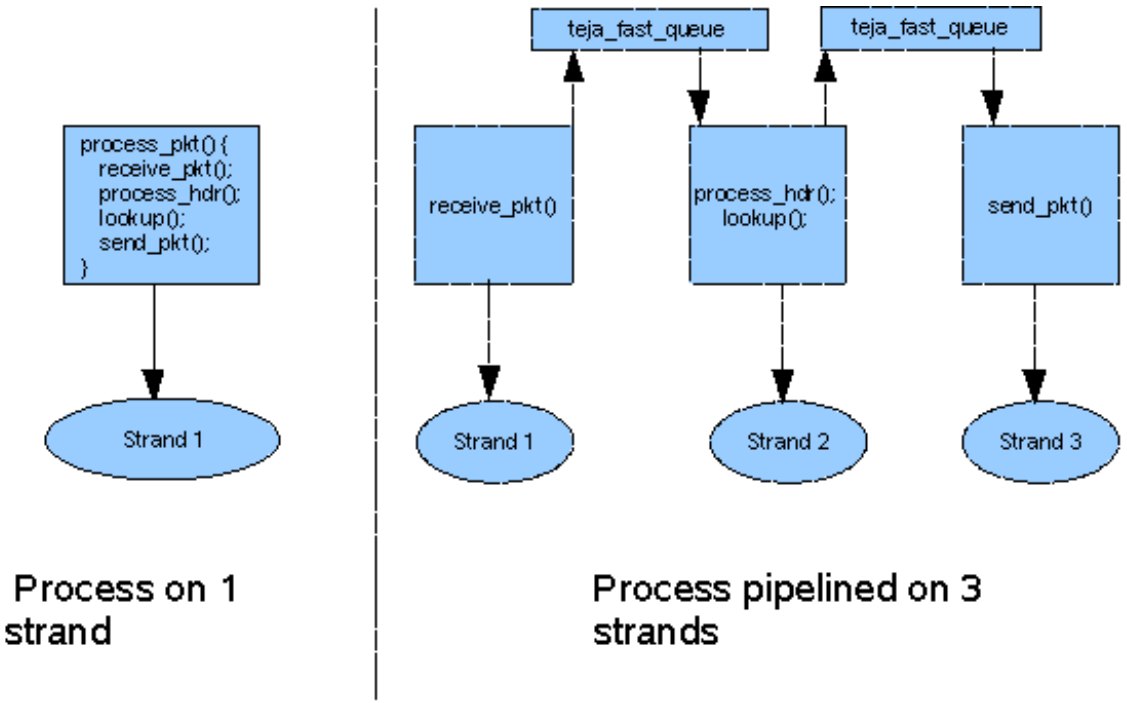


FIGURE 10-6 Example of Pipelining

FIGURE 10-7 shows how pipelining improves the throughput.

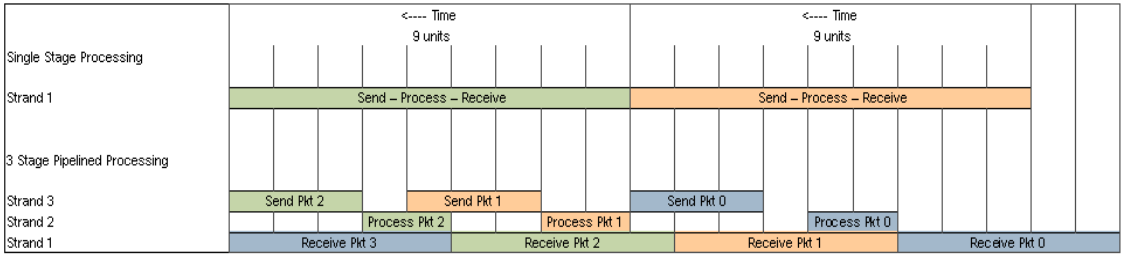


FIGURE 10-7 Pipelining Effect on Throughput

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 10-8 is allocated three strands for encryption.

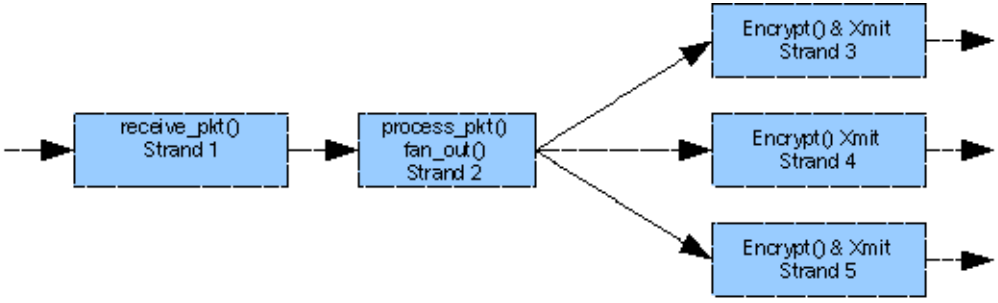


FIGURE 10-8 Parallelizing Encryption Using Multiple Strands

The process strand uses 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 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. Therefore, 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.

Slowing Down Polling

As explained in "Parking Idle Strands" on page 180, 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 T processor, the CPI for a compute-bound thread is less than five and for a memory-bound thread is considerably higher than five.

Cannot 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`, or 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.

Cannot 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 RLP 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 for the four instances of the RLP application. FIGURE 10-9 describes the architecture.

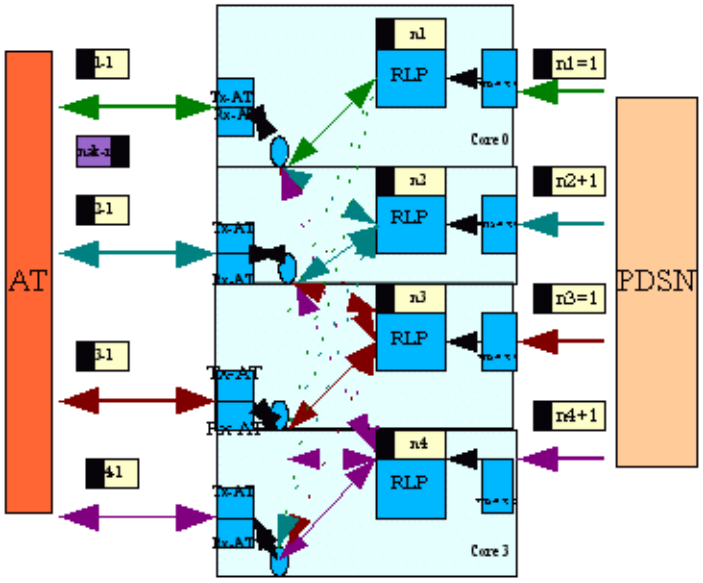


FIGURE 10-9 RLP Application Setup

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 10-3.

TABLE 10-3 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 10-4.

TABLE 10-4 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 Netra DPS Profiling API (refer to the *Netra Data Plane Software Suite 2.0 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 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 10-1 shows the functions that were created to collect the hardware counter data.

CODE EXAMPLE 10-1 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 10-1. For a detailed description of these API functions refer to the *Netra Data Plane Software Suite 2.0 Reference Manual*.

A sample output based on the code in CODE EXAMPLE 10-1 is shown in CODE EXAMPLE 10-2.

CODE EXAMPLE 10-2 Sample Profile Output

```
PROFILE_DUMP_START,ver,2.0
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 44. The output in CODE EXAMPLE 10-2 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 10-5.

TABLE 10-5 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 enables comparison rates from different strands.

These metrics in TABLE 10-5 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 10-10.

	65% Line	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_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 10-10 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 10-11.

	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 10-11 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. Other methods of collecting hardware counter data enable you to optimize a particular section of the application.

Comparing the Instruction per Cycle columns from FIGURE 10-10 and FIGURE 10-11 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 10-1 to calculate the worst case effect of the data cache and L2 cache misses. The results for these calculations are shown in TABLE 10-6 for configuration 1 and in TABLE 10-7 for configuration 2.

TABLE 10-6 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 10-7 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.

Other effects are 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 instructions 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 for Profiling

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.

APPENDIX **A**

Tutorial

This appendix is a tutorial to `tejacc` programming. Topics include:

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. CODE EXAMPLE A-1 lists the `ticktock.c` file and provides comment.

CODE EXAMPLE A-1 `ticktock.c` File and Comments

```
#include <stdio.h>
#include "teja_late_binding.h"


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 */
  }
}

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;
    }
  }
}

int
init(void)
{
  printf("init\n");
  return 0;
}
```

`stdio.h` and `teja_late_binding.h` are included. This action declares the Netra DPS late-binding API.

The `ticker` function uses two late-binding objects, a memory pool called `tick_memory_pool` and a channel called `ticktock_channel`. These functions 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.

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.

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. CODE EXAMPLE A-2 lists the `config.c` file and provides comment.

CODE EXAMPLE A-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);

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;
}
```

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

A user-defined hardware architecture called `top` is created as a container for the PC architecture

CODE EXAMPLE A-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. 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 Netra DPS-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.**

Executing the Binary Image

▼ To Execute the Binary Image

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

  The CMT machine is reset, and the system is booted. See "Building and Booting Reference Applications" on page 8. 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 ...
```

APPENDIX **B**

Frequently Asked Questions

This appendix provides frequently asked questions regarding the Netra DPS.

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 220.

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

Where Are the Tutorials?

The `ticktock` tutorial is described in "Tutorial" on page 197.

Configuration Questions

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

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 220.

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), use the following procedure.

▼ To Debug the Dynamic Libraries

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`.**
4. **Control returns immediately after the user dynamic libraries are loaded.**
5. **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. See "How Can I Debug the Dynamic Libraries?" on page 207.

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 (`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 `tejacc` 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 210.

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 in this section). This action 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. For example, to set the new value for the property, do the following:

```
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. The path can be relative to the top-level application directory or an absolute value.

Note – There is no warning or error if the file does not exist. The compilation continues 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 takes 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`.

You can generate the Makefile as shown below:

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

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

How Do I Compile the Reference Applications?

See Chapter 9, "Reference Applications" on page 127.

Late-Binding Questions

Note – Refer to "Late-Binding API Overview" on page 25 for more information on the Late-Binding API.

What Is the Late-Binding API?

The Late-Binding API is the Netra DPS 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 220.

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 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 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.

What Is a 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 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 212), 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.

Eclipse Questions

How Can I Change the Build Command?

In Eclipse, open the Window/Preferences menu. In the left-side tree, open the C/C++/New CDT project wizard/Makefile project node. In the right-side of the window, select the Builder settings tab. In the section Builder, deselect Use default build command and in the text field below it, type the command of choice.

How Can I Change the Compiler Invocation Command?

In Eclipse, open the Window/Preferences menu. In the left-side tree open the C/C++/New CDT project wizard/Makefile project node. In the right-side of the window select the Discovery options tab and in the Compiler invocation command text field, type the command of choice.

API and Application Questions

Note – Refer to the *Netra Data Plane Software Suite 2.0 Reference Manual* for detailed description of the API functions.

How Do I Synchronize a Critical Region?

Use the mutex API which consists of the following:

- `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()`

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 enable 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 B-1.

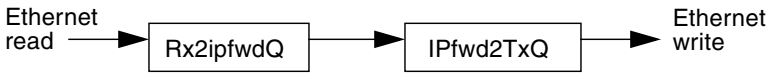


FIGURE B-1 Example for the `ipfwd` Application

Note – If the Netra DPS 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. In some situations, however, the Software Architecture API can provide a better role for a strand.

Is It Possible to Park a Strand under LDoms as Done in a Non-LDoms Environment?

Methods of parking strands are no different in an LDoms environment. Strands not utilized 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 Netra DPS Runtime Environment logical domain are not visible to that domain and cannot be parked.

Can You Assign Partial Cores to a Netra DPS domain?

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

What Is `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.

For example:

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

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

When the example in What Is bss_mem? 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.cmt2board.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 B-1 illustrates the allocation of memory for BSS, code, heap, and DRAM.

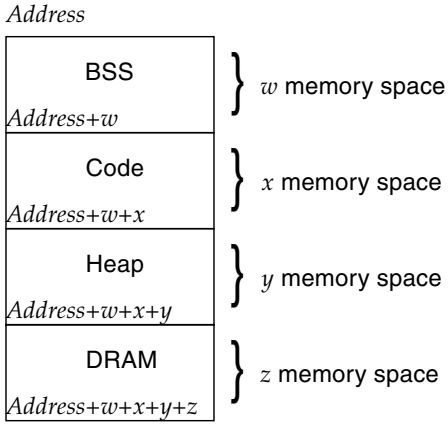


FIGURE B-2 Memory Allocation Stack

Note – These memory regions are not necessarily contiguous. There may be gaps in between each region.

---

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 (VNET) Devices?

The `eth_*` API only supports physical ethernet devices at this time.

Optimization Questions

How Do I Enable Optimization?

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

TABLE B-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 "Late-Binding API Overview" on page 25 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 Netra DPS Application?

You can port pre-existing applications to the Netra DPS environment. There are two methods to integrate legacy application C code with newly compiled Netra DPS application code:

- "Linking Legacy Code to Netra DPS Code" on page 221
- "Changing Legacy Source Code" on page 222

Linking Legacy Code to Netra DPS Code

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

Changing Legacy Source Code

Introducing calls to the Netra DPS 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 Netra DPS 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 Netra DPS calls as only performance-critical parts of legacy code need to be ported to Netra DPS. Error handling and exception code can remain unchanged.

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

Note – See "How Can I Reuse Legacy C Code in a Netra DPS Application?" on page 221.

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

- "Mixing C and C++ Code" on page 222
- "Translating C++ Code to C Code" on page 223

Mixing C and C++ Code

Netra DPS generates C code, so the final program is in C. Mixing C++ and Netra DPS 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 Netra DPS 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 the code to the generated Netra DPS code. The Netra DPS 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-2.0-MP-G-F&TransactionId=Try

Sun CMT Specific Questions

Is There a Maximum Allowed Size for Text and 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 B-2 lists the default memory setup in Sun CMT hardware architecture:

TABLE B-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. 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 224.

Address Resolution Protocol Questions

How Do I Enable ARP in the RLP Application?

▼ To Enable ARP in RLP

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 Netra DPS.

```
#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
```

How Do I Enable ARP Without Relying on a Control Domain?

Netra DPS applications can make use of the LWIP stack, provided in the `SUNWndps` package. LWIP wrapper APIs are provided for the ease of the application writer. These APIs are located in the following header file: `netif/lwrtearp.h` (`/opt/SUNWndps/src/libs/lwip/src/include/netif/lwrtearp.h`). The RLP reference application (`/opt/SUNWndps/src/apps/rlp`) makes use of these APIs.

How Do I Enable ARP Using a Control Domain?

The `ipfwd`-ARP integration makes use of the LWIP stack in the control-plane to update the ARP entries in the Forward Information Base (Forwarding table) and passes the Forwarding table to Netra DPS runtime. If the application writer needs ARP using a control-domain, then they can design their application according to the `ipfwd` reference application (see "Reference Applications" on page 127).

Glossary

A

ADE Netra DPS Eclipse-based Teja Advance Development Environment (ADE) graphical user interface. Teja ADE views three Teja elements: hardware architecture, software architecture and mapping.

AH/ESP Authentication Header/Encapsulating Security Payload.

AN Access network.

API Application programming interface.

AT Access terminal.

ARP Address Resolution Protocol.

B

bsp Header files and low-level Sun UltraSPARC T1 and Sun UltraSPARC T2 platform initialization and management code.

C

CAM Content addressable memory.

CLI Command-line interface.

CG Cipher group.

CMT Chip multithreading.

CMT1 Chip multithreading for Sun UltraSPARC T1 systems.

CMT2 Chip multithreading for Sun UltraSPARC T2 systems.

consumers Threads receiving messages from a channel.

CSP Chip support package. A target-specific section of the code generator aware of hardware features. CSP is responsible for generating thread startup code, mutexes, and so on.

D

dbg Chip multithreading (CMT) debugger program. Netra DPS native debugger is the default debugger and is useful for debugging during development,

DMA Direct memory access.

E

Eclipse An open source community where projects are focused on building extensive development platforms, runtimes, and application frameworks.

ESP IP Encapsulating Security Payload.

G

GDB GNU debugger that enables you to debug your program in C source code level.

GUI Graphical user interface.

I

IPC Interprocess communication software mechanism that provides a means to communicate between processes that run in a domain under the Netra DPS Lightweight Runtime Environment (LWRTE) and processes in a domain with a control plane operating system.

IPSec Internet Protocol Security.

IPv4 Internet Protocol Version 4.

IPv6 Internet Protocol Version 6.

IV Initialization vector.

L

late-binding Provides primitives for the synchronization of distributed threads, communication, and memory allocation.

LDC Logical domain channel.

LDoms Logical Domains.

LWRTE Lightweight Runtime Environment. Provides an ANSI C development environment for creating and scheduling application threads to run on individual strands on the UltraSPARC T series processor.

M

mblk Message Block. A data structure that carries packet information.

N

NAK Negative-Acknowledge is sent by a station to indicate that an error was detected in the previously received block and that the receiver is ready to accept retransmission of that block.

Netra DPS Netra Data Plane Software Suite. In this document, this suite is also referred to as Netra DPS.

Netra DPS Runtime 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.

NIU Network Interface Unit (Sun multithreaded 10GbE with Network Interface Unit). Networking hardware consisting of a Receive Packet Classifier that performs L2/L3/L4 header parsing, matching, and searching functions.

P

parked A parked strand does not consume any pipeline cycles (an inactive strand).

PDSN The Packet Data Serving Node, or PDSN, is a component of a CDMA2000 mobile network. It acts as the connection point between the Radio Access and IP networks. This component is responsible for managing PPP sessions between the mobile provider core IP network and the mobile station.

producers Threads sending messages to a channel.

R

RFC Request for Comments (RFC) documents are a series of memoranda encompassing new research, innovations, and methodologies applicable to Internet technologies.

S

SADE Static Security Association Database.

SCTP Stream Control Transmission Protocol.

source set Consists of one or more source files. The source set is used to map to one or more processes.

SPD Security policy database.

SPI Security parameter index.

SPU Stream processing unit.

strand A hardware thread, multistrand partitioning firmware for Sun CMT platforms.

Studio 12 C compiler software.

`SUNWndps` Netra DPS software package installed in the development server. Contains system-level libraries, header files, and low-level Sun UltraSPARC T1 and Sun UltraSPARC T2 platform code, and tools for compiler and runtime system.

`SUNWndpsd` Netra DPS software package installed on the target deployment system. Contains the Netra Data Plane CMT and IPC Share Memory Driver.

`SUNWndpsc` Netra DPS software package containing the Sun UltraSPARC T2 cryptography driver.

T

TCAM Temary content addressable memory.

TCP Transmission Control Protocol.

`tejacc` A compiler that provides the constructs of threads, mutex, queue, channel, and memory pool within the application code.

Teja NP 4.0 Teja NP 4.0 software platform used to develop scalable, high-performance C applications for embedded multiprocessor target architectures.

thread Runs in a process and is a target for executing a function. Thread management functions offer the ability to start and end threads dynamically.

`tnsmctl` Contained in `SUNWndpsd` package and contains the Netra Data Plane CMT and IPC Share Memory Driver.

U

UDP User/Universal Datagram Protocol.

UltraSPARC T1 Processor that employs chip multithreading, or CMT, which combines chip multiprocessing (CMP) and hardware multithreading (MT). This processor creates a SPARC V9 processor with up to eight 4-way multithreaded cores for up to 32 simultaneous threads.

UltraSPARC T2 Processor that is the second generation of the CMT processor. In addition to features found in UltraSPARC T1, UltraSPARC T2 dramatically increases processing power by increasing the number of hardware strands in each core. UltraSPARC T2 includes on-chip 10G Ethernet and crypto accelerator.

V

VLAN Virtual Local Area Network.

Index

Q

R

S

T

U

V