

Netra Data Plane Software Suite 2.1 Update 1

User's Guide



Part No.: E20948-01
February 2011

Copyright © 2009, 2011, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Copyright © 2009, 2011, Oracle et/ou ses affiliés. Tous droits réservés.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, breveter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est concédé sous licence au Gouvernement des Etats-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l'utilise pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique :

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. UNIX est une marque déposée concédée sous licence par X/Open Company, Ltd.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation



Adobe PostScript

Contents

Using This Documentation xxix

1. Sun Netra Data Plane Software Suite Overview 1

Product Description 1

Supported Systems 2

Software Installation 3

Platform Firmware Prerequisites 4

▼ To Check Your OpenBoot PROM Firmware Version 5

Package Dependencies 7

Package Installation Procedures 7

▼ To Install the Software Into the Default Directory 8

▼ To Install the Software in a Directory Other Than the Default 8

▼ To Remove the Software 10

Building and Booting Reference Applications 10

.cshrc File and Required Compiler Path 10

Building Reference Application Instructions 11

▼ To Boot an Application Image 12

Programming Methodology 13

Reusing Existing C Code 14

tejacc Compiler Basic Operation 15

tejacc Compiler Mechanics	15
tejacc Compiler Options	16
tejacc Compiler Configuration	16
tejacc Compiler and Sun Netra DPS Interaction	17
Architecture Elements	19
Hardware Architecture API Overview	19
Hardware Architecture Elements	19
Architecture Relationships	20
Utility Functions	21
Advanced Hardware Architecture Elements	21
Software Architecture and Late-Binding API Overview	23
Late-Binding Elements	24
Other Elements	27
Utility Functions	28
User API Overview	29
Late-Binding API Overview	29
Sun Netra DPS Runtime API Overview	29
Finite State Machine API Overview	31
Map API Overview	31
2. tejacc Basics	33
Command-Line Options	33
tejacc Command-Line Options	34
Optimization	35
Optimization Options	35
Context-Sensitive Generation	36
▼ To Enable Optimization	36
Language	37
Include Files	37

3. Profiler 39

Profiler Introduction 39

How the Profiler Works 40

Groups and Events 40

Profiler Output 41

Profiler Examples 43

 Profiler API 43

 Profiler API Usage for the Sun UltraSPARC T1 Processor 43

 Profiler API Usage for the Sun UltraSPARC T2 Processor 44

 Profiler Configuration 44

 Profiler Output Example 45

Profiling Application Performance 46

 Sun UltraSPARC T1 Performance Counters 46

 Sun UltraSPARC T2 Performance Counters 49

User-Defined Statistics 52

Profiling Metrics 53

Using the Profiler Script 53

Profiler Scripts 54

 Usage 54

 Raw Profile Data 55

 Summarized Profile Data 58

 Sun UltraSPARC T1 Processor Profiler Output 58

 Sun UltraSPARC T2 Processor Profiler Output 59

Performance Parameters Calculations 63

 Sun UltraSPARC T1 Processor 64

 Instructions per Packet: 64

 Instructions per Cycle (IPC): 64

Packet Rate:	64
SB_full per thousand instructions:	64
FP_instr_cnt per thousand instructions:	64
IC_miss per thousand instructions:	64
DC_miss per thousand instructions:	64
ITLB_miss per thousand instructions:	64
DTLB_miss per thousand instructions:	65
L2_imiss per thousand instructions:	65
L2_dmiss_LD per thousand instructions:	65
Sun UltraSPARC T2 Processor	65
Instruction per Packet:	65
Instructions per Cycle (IPC):	65
Store Instructions per Packet:	65
Load Instructions per Packet:	66
L2 Load misses per Packet:	66
Icache misses per 1000 Packets:	66
Dcache misses per Packet:	66
Packet Rate:	66
▼ To Use a Spreadsheet for Performance Analysis	67

4. Debugger 69

Debugger Introduction	69
Native Debugger	70
Debugging Configuration Code	70
Entering the Debugger	71
Native Debugger Commands	71
Displaying Help	71
help or h	71
Example:	72

Managing Breakpoints	72
break <i>address</i> or b <i>address</i>	72
info break or i break	73
Example:	73
delete breakpoint <i>ID</i> or d breakpoint <i>ID</i>	73
Managing Program Execution	74
cont or c	74
step or s	74
Example:	74
Displaying and Setting Memory	74
x/nfu <i>address</i>	74
w/u <i>address value</i>	75
Example:	75
Managing Threads	75
info threads or i threads	75
thread <i>ID</i>	75
Example:	76
Displaying Registers	76
info reg or i reg	76
Displaying Stack Trace	78
bt <i>frame-count</i>	78
Resolving Symbols Using Options	79
-h	79
-f <i>function-name</i>	79
-g <i>global-variable</i>	79
-l <i>file-name:line-number</i>	80
GNU Project Debugger	80
Configuring Oracle VM Server for SPARC Software for GDB Support	80

- ▼ To Configure the Oracle VM Server for SPARC Software Required to Run the Sun Netra DPS Application With GDB Support 80

- ▼ To Configure the Oracle Solaris Domain for GDB 82

GDB Showcase Application 82

- ▼ To Compile the GDB Showcase 82

- ▼ To Load the GDB Showcase Binary in the Sun Netra DPS Domain 83

- ▼ To Run the GDB Command 83

GDB Commands 84

- ▼ To Run Sun Netra DPS Application With GDB Support 85

5. Interprocess Communication Software 89

IPC Introduction 89

Programming Interfaces Overview 90

Configuring the Environment for IPC 90

Memory Management 90

IPC in the Logical Domains Environment 91

Logical Domain Channel Setup 91

IPC Channel Setup 92

Example Environment for UltraSPARC T1 Based Servers 94

Domains 94

primary 95

ldg1 – LWRTE 95

ldg2 – Control Plane Application 95

ldg3 – Solaris Control Domain 95

Virtual Data Plane Channels 96

Global Control Channel 96

Client Control Channel 96

Data Channel 96

IPC Channels 97

Example Environment for UltraSPARC T2 Based Servers	98
IPC Reference Applications	99
Common Header	99
6. Remote Command-Line Interface	101
Remote Command-Line Interface Introduction	101
IPC Setup for Remote CLI	102
▼ To Configure the Oracle Solaris Domain for Remote CLI	102
Accessing the Remote CLI	103
▼ To Access the CLI Console	104
Debugging Remotely	105
▼ To Access the Sun Netra DPS Debugger	105
Coredump Support	106
System Configuration	106
▼ To Go to the <code>sys</code> Mode From the Remote CLI	106
Compiling the Remote CLI Application	107
Build Script	107
Usage	108
Build Script Arguments	108
Argument Descriptions	108
<code>cmt</code>	108
<code>[profiler]</code>	108
7. Eclipse Development Environment	109
ADE Introduction	109
Starting the Eclipse-Based ADE GUI	110
▼ To Start the Eclipse-Based ADE GUI	110
Creating a Teja Project	110
▼ To Create a Project in the Same Directory as an Existing Teja Application	110

- ▼ To Add the Graphic Files to a Project 115
- Files and Viewers 115
 - Hardware Architecture Viewer 115
 - Software Architecture Viewer 118
 - Mapping Viewer 120
- Build 121
 - ▼ To Compile the Teja Application in the Eclipse-Based ADE 121

8. Receive Packet Classification 123

- Receive Packet Classification Introduction 123
- Supported Networking Interfaces 124
- Sun Multithreaded 10GbE and NIU Receive Packet Classifier 124
- Receive DMA Channel Selection 124
- Hashing Based on Layer 2, Layer 3, and Layer 4 Header Classification 128
 - Hash Algorithm 129
 - Hash Key 129
 - Application 130
 - Hash Policy 130
- Flow Match Based on Layer 2, Layer 3, and Layer 4 Header Classification 131
 - Layer 2 (L2) Classification 131
 - Layer 3 and Layer 4 (L3/L4) Classification 132
 - Applications 132
 - Classification Programming Interface 132
 - opcode 133
 - action 133
 - flow_spec 134
 - channel 135
 - ue or um 135
 - hd 135

flow_spec_ipv4_tab_s 136

flow_spec_ipv6_t 136

flow_spec_l2_t 137

Examples 137

▼ To Use Hash Flow 137

▼ To Use TCAM Classification 137

9. Auto-Configuration 143

Auto-Configuration Introduction 143

Installation 144

Prerequisites 144

User Interface 145

Configuring a Logical Domain Environment for Reference Applications 145

Custom Configuring a Primary Domain 146

Custom Configuring a Guest Domain 148

Configuring LDC and IPC 149

Saving Current Guest Domains Configuration 151

Configuring the Oracle VM Server for SPARC Software from a Saved Location 152

10. Transparent Interprocess Communication 153

Transparent Interprocess Communication Introduction 153

TIPC Components 154

Installing TIPC 155

▼ To Install TIPC 155

Programming Interfaces Overview 157

Configuring Environment for TIPC 158

SUNWndpsd and SUNWndps-tipc Binaries 158

Configuring Sun Netra DPS TIPC Stack from an Oracle Solaris Guest Logical Domain 159

Configuring Sun Netra DPS TIPC Stack from a Linux Guest Logical Domain 159

▼ To Set the TIPC Address 160

Enabling TIPC Ethernet Bearer 160

Enabling the TIPC IPC Bearer 161

Enabling TIPC vnet Bearer for a NDPS TIPC Node 161

11. Reference Applications 163

IP Packet Forwarding Reference Applications 164

Receive Thread 164

Forwarding Thread 165

Transmit Thread 166

Traffic Flows 166

Source Files 166

▼ To Compile the `ipfwd` Application 167

Usage 167

Argument Descriptions 167

▼ To Build the `ipfwd` Application 169

▼ To Run the `ipfwd` Application 169

Default System Configuration 170

Default `ipfwd` Application Configuration 170

Other IP Forwarder Options 170

IP Forward Static Cross Configuration 172

Flow Policy for Spreading Traffic to Multiple DMA Channels 172

`ipfwd` Flow Configurations 173

`ipfwd` Configuration File Format 173

System Configuration 175

Standalone Environment 175

Logical Domain Environment 175

Forwarding Application	175
Data Plane Components	176
Control Plane Components and Utilities	178
IPv4 Packet Forwarding Application with Exception Handling	185
ARP Processing	186
ARP in lwIP	186
ARP in the Oracle Solaris OS	186
ARP in the Oracle Solaris OS or Linux OS Using vnet	186
IPv4 Protocol Exception Handling	187
Fragmentation	187
Reassembly and Local Delivery	187
Reassembly and Local Delivery Using vnet	188
FIB Management	188
FIB Management When Using vnet	191
Exception Path Framework Components	191
IPv4 Forwarder (ipfwd Thread)	191
Exception Application (excpd)	192
lwIP ARP Layer	192
ARP STREAMS Module (lwmodarp)	192
The IPv4 STREAMS Module (lwmodip4)	193
Fastpath Manager	193
Exceptions Path Framework Tools	193
ifctl	194
fibctl	194
insarp	194
▼ To Compile the ipfwd Application for IPv4 Exception Handling	195
▼ To Compile the IPv4 Forwarding Application With Exception Handling By Using Sun Netra DPS	195
Compiling the excpd Application	195

Usage 196

- ▼ To Build the `excpd` Application When `lwIP` ARP Is Used With IPC 196
- ▼ To Build the `excpd` Application When `lwIP` ARP Is Used With TIPC 196
- ▼ To Build the `excpd` Application When the Oracle Solaris OS ARP Is Used With IPC 196
- ▼ To Build the `excpd` Application When the Oracle Solaris OS ARP Is Used With TIPC 196

Compiling the `lwmodip4` STREAMS Module 197

Usage 197

- ▼ To Build the `lwmodip4` STREAMS Module for IPv4 Exception Handling Using IPC 197
- ▼ To Build the `lwmodip4` Module for IPv4 Exception Handling Using TIPC 197

Compiling the `lwmodarp` STREAMS Module 197

Usage 197

- ▼ To Build the `lwmodarp` Module for Oracle Solaris ARP Handling Using IPC 198
- ▼ To Build the `lwmodarp` Module for Oracle Solaris ARP Handling Using TIPC 198

Compiling the `insarp` Tool 198

- ▼ To Compile the `insarp` Tool 198
- ▼ To Run the `ipfwd` Application with IPv4 Exception Handling in `lwIP` 198
- ▼ To Run the `ipfwd` Application with IPv4 Exception Handling and ARP Handling in the Oracle Solaris Host 200
- ▼ To Compile the `ipfwd` Application with IPv4 Exception Handling using `vnet` in Sun Netra DPS 202
- ▼ To Run the `ipfwd` Application with IPv4 Exception Handling and ARP Handling in an Oracle Solaris OS Host Using `vnet` 202
- ▼ To Compile the IPv4 Forwarding Application With Exception Handling Using `vnet` in Sun Netra DPS 204

- ▼ To Run the `ipfwd` Application with IPv4 Exception Handling and ARP Handling in the Linux Host Using `vnet` 204
- IPv6 Packet Forwarding Application with Exception Handling 206
 - Interface Management 207
 - IPv6 Protocol Exception Handling 207
 - IPv6 Protocol Exception Handling Using `vnet` 208
 - FIB Management 208
 - FIB Management Using `vnet` Exception Handling 209
 - IP-IP Tunneling 209
 - Data-Plane and Control-Plane Synchronization 209
 - Exception Path Components 210
 - IPv6 Forwarder (`ipfwd` Strand) 210
 - IPv6 STREAMS Module (`lwmodip6`) 211
 - Fastpath Manager 211
 - Exception Path Tools 211
 - `ifctl` 211
 - `fibctl` 211
 - `fibctl.sh` 212
 - `ipfwd_sync.d` 212
 - ▼ To Compile the Reference Application 212
 - ▼ To Compile the IPv6 Forwarding Application With Exception Handling Using Sun Netra DPS 212
- Compiling the `lwmodip6` STREAMS module 213
 - ▼ To Build the `lwmodip6` Module for IPv6 Exception Handling Using IPC 213
 - ▼ To Build the `lwmodip6` Module for IPv6 Exception Handling Using TIPC 213
 - ▼ To Run the `ipfwd` Application With IPv6 Exception Handling 213
 - ▼ To Compile the IPv6 Forwarding Application With Exceptional Handling Using `vnet` 215

- ▼ To Run the `ipfwd` Application With IPv6 Exception Handling 215
- ▼ Run the `ipfwd` Application That Is Compiled With Exception Handling 216
- ▼ To Compile the IPv6 Forwarding Application Using `vnet` Exceptional Handling in a Linux Guest Logical Domain 217
- ▼ To Run the `ipfwd` Application Using IPv6 Exception Handling in a Linux Guest Logical Domain 217
- ▼ Run the `ipfwd` Application That Is Compiled With Exception Handling 218

Differentiated Services Reference Application 220

Classifiers 221

Differentiated Services Code Point Classifier 221

6-Tuple Classifier 221

Policing (Meter) 222

Single-Rate Three-Color Marker 222

Two-Rate Three-Color Marker 222

DSCP Marker 222

Shaping 222

Deficit Round Robin Scheduler 222

Queue Manager 223

Building the DiffServ Application 223

DiffServ Command-Line Interface Implementation 224

- ▼ To Build the Extended Control Utility 224

Command-Line Interface for the IPv4-DiffServ Application 224

DSCP Classifier 224

`add` 224

`delete` 225

`update` 225

`purge` 226

`display` 226

6-Tuple Classifier	226
add	227
delete	227
update	228
purge	229
display	229
enable or disable	229
TC Meter	230
add	230
delete	231
update	231
purge	233
display	233
stats	233
Scheduler	234
add	234
update	234
display	235
DiffServ References	236
Generic Routing Encapsulation Reference Application	236
Generic Routing Encapsulation Introduction	237
References	237
Data Plane Architecture	237
IPv4 Forwarding Data Plane	238
GRE Over IPv4 Data Plane	238
GRE Over IPv4 Data Plane Internal Block Diagram	239
GRE Over IPv4 Application	239
IPv4 Forwarder	239

GRE Encapsulator	240
GRE Decapsulator	240
Key and Sequence Number Extensions to GRE	240
GRE Command-Line Interface Implementation	241
Directory Structure	241
▼ To Compile the GRE Code	241
▼ To Compile the IPv4 and GRE Application Using Sun Netra DPS	242
▼ To Compile the Command-Line Interface Application	242
▼ To Run the IPv4 and GRE Application	242
▼ To Run the CLI Application	243
CLI for the IPv4-GRE Application	243
add	243
update	244
delete	244
purge	245
display	245
GRE Reference Application Example	246
▼ To Build the GRE Reference Application	246
Traffic Generator Configuration	246
Access Control List Reference Application	247
▼ To Build the ACL Application	248
▼ To Run the ACL Application	248
▼ To Configure the ACL Application Environment Using LDC	248
▼ To Configure the ACL Application Environment Using TIPC	249
Command-Line Interface for the ACL Application	249
▼ To Use <code>acltool</code> in a Linux OS Control Domain	251
Radio Link Protocol Reference Application	252
▼ To Compile the RLP Application	252

Build Script	252
Usage	252
Argument Descriptions	253
▼ To Build the RLP Application	254
▼ To Run the Application	254
Default System Configuration	255
Default RLP Application Configuration	255
Other RLP Options	255
▼ To Bypass the rlp Operation	256
▼ To Use One Global Memory Pool	256
Flow Policy for Spreading Traffic to Multiple DMA Channels	256
IPSec Gateway Reference Application	256
IPSec Gateway Application Architecture	257
IPSec Gateway Application Capabilities	257
High-Level Packet Processing	258
Outbound Packets	258
Inbound Packets	259
Security Association Database and Security Policy Database	259
Outbound Packets and Inbound Packets	260
Static Security Policy Database and Security Association Database	262
SPD	263
SAD	263
Packet Encapsulation and De-encapsulation	265
Packet Encapsulation	266
Memory Pools	268
Pipelining	268
Source Code File Description	269
Build Script	269

Usage	270
Argument Descriptions	270
Reference Application Configurations	273
IP with Encryption and Decryption	273
IPSec Gateway on Quad GE	274
IPSec Gateway on NIU 10-Gbps Interface (One Instance)	275
IPSec Gateway on NIU 10-Gbps Interface (Up to Four Instances)	277
Multiple Instances (Up to Eight Instances) Back-to-Back Tunneling Configuration	279
Flow Policy for Spreading Traffic to Multiple DMA Channels	282
▼ To Enable a Flow Policy	282
Traffic Generator Reference Application	283
Using the User Interface	283
▼ To Start the ntgen User Interface	283
ntgen Option Descriptions	284
Option Descriptions	285
ntgen Parameter Description	294
Notes	295
Traffic Generator Output	296
Template Files	296
Using the Traffic Generator	297
Configuring Logical Domains for the Traffic Generator	297
▼ To Add the tnsmdriver	298
▼ To Prepare Building the ntgen Utility	301
▼ To Set Up and Use Logical Domains for the Traffic Generator	301
▼ To Start the Traffic Generation	301
▼ To Stop Traffic Generation	302
▼ To Compile the Traffic Generator	302
Build Script	302

Usage	302
Argument Descriptions	302
▼ To Run <code>ndpstgen</code>	303
Default Configurations	304
Interprocess Communication Reference Application	304
IPC Reference Application Content	305
Building the IPC Reference Application	306
Usage	306
Argument Descriptions	306
Example	307
Running the IPC Application	307
▼ To Use the <code>ipctest</code> Utility	307
<code>ipctest</code> Commands	308
▼ To Install the <code>lwmod</code> STREAMS Module	308
▼ To Remove the <code>lwmod</code> STREAMS Module	309
Transparent Interprocess Communication Reference Application	310
Source Files	311
Default Configurations	312
▼ To Compile the TIPC Application	312
Build Script	312
Usage	313
Argument Descriptions	313
▼ To Run the TIPC Application	314
IP Forward Reference Application Using TIPC	315
▼ To Build the IP Packet Forward (<code>ipfwd</code>) Application	315
▼ To Configure the Environment for TIPC	315
▼ To Configure Oracle Solaris OS TIPC Stack in Oracle Solaris Domain (<code>ldg2</code>)	316
Command-Line Interface Application using TIPC	317

▼ To Build the Extended Control Utility	317
FIB Table Configuration Command Line Interface (fibctl)	318
Interface Configuration Command Line Interface (ifctl)	318
IPv4 Exception Process (excpd)	318
vnet Reference Application	319
UltraSPARC T2 Platform	319
UltraSPARC T1 Platform	319
Supported Tests	320
testvnet Commands	320
Test Setup	321
Virtual Network Setup	323
vnet Reference Application Content	325
Building the Sun Netra DPS vnet Reference Application	326
Usage	326
Argument Descriptions	326
▼ To Build the vnet Reference Application	327
▼ To Run the vnet Sun Netra DPS Application, vnettest	327
▼ To Build the vnet Guest Logical Domain Application for the Oracle Solaris OS	327
▼ Building the vnet Guest Logical Domain Application for the Linux OS	327
▼ To Run the vnet Guest Logical Domain Application on a Oracle Solaris OS Guest Logical Domain	328
▼ To Run the vnet Guest Logical Domain Application on a Linux OS Guest Logical Domain	330

12. Performance Tuning 333

Performance Tuning Introduction	333
UltraSPARC T1 Processor Overview	334
UltraSPARC T2 Processor Overview	336

Identifying Performance Issues	338
UltraSPARC T1 Performance	338
UltraSPARC T2 Performance	341
Optimization Techniques	343
Code Optimization	343
Pipelining	344
Parallelization	345
Mapping	346
Parking Idle Strands	346
Slowing Down Polling	347
Tuning Troubleshooting	348
What Is a Compute-Bound Versus a Memory-Bound Thread?	348
Cannot Reach Line Rate for Packets Smaller Than 300 Bytes	348
Cannot Scale Throughput to Multiple Ports	349
How Do I Achieve Line Rate for 64-byte Packets?	349
When Should I Consider Thread Placement?	350
Example RLP Exercise	350
Application Configuration	350
Configuration 1	352
Configuration 2	352
Using the Profiling API	352
Profiling Data	355
Metrics	357
Configuration 1 Results	357
Configuration 2 Results	358
Analysis	359
Other Uses for Profiling	361

A. Tutorial 363

Application Code	363
Configuration Code	366
Build Process	368
▼ To Create the Binary Image	368
Executing the Binary Image	370
▼ To Execute the Binary Image	370

B. Frequently Asked Questions 371

Summary	372
General Questions	374
What Is Teja 4.x and How Does It Differ From an Ordinary C Compiler?	374
Where Are the Tutorials?	375
Configuration Questions	375
What Purpose Are the Hardware Architecture, Software Architecture, and Mapping Dynamic Libraries?	375
How Can I Debug the Dynamic Libraries?	375
▼ To Debug the Dynamic Libraries	376
What Should I Do When the <code>tejacc</code> Compiler Crashes?	376
What if the Hardware Architecture, Software Architecture, or Mapping Dynamic Libraries Crash?	376
Can I Build Hardware Architecture, Software Architecture, and Mapping in the Same Dynamic Library?	377
Can I Map Multiple Variables With One Function Call?	377
Building Questions	377
Where Is the Generated Code?	377
Where Is the Executable Image?	378
How Can I Compile Multiple Modules on the Same Command Line?	378
How Can I Pass Different CLI Options to Different Modules on the <code>tejacc</code> Command Line?	378
How Can I Change the Behavior of the Generated <code>makefile</code> Without Modifying it?	378

How Do I Compile the Reference Applications?	379
Late-Binding Questions	380
What Is the Late-Binding API?	380
What Is a Memory Pool?	380
What Is a Channel?	381
What Is the Difference Between OS Based and Non-OS Based Memory Pools and Channels?	381
How Do I Access a Late-Binding Object From Application Code?	381
Can I Define a Symbol in the Software Architecture and Use it in My Application Code?	382
Eclipse Questions	382
How Can I Change the Build Command?	382
How Can I Change the Compiler Invocation Command?	382
API and Application Questions	383
How Do I Synchronize a Critical Region?	383
How Do I Send Data From a Thread to Another Thread?	383
How Do I Allocate Memory?	384
When Should I Use Queues Instead of Channels?	384
Why Is it Not Necessary to Block Interface or Queue Reads?	384
Can Multiple Strands on the Same Queue Take Advantage of the Extra CPU Cycles if the Strands Are Not Being Used?	385
Why Does the Application Choose the Role for the Strand From the Code Instead of the Software Architecture API?	385
Is It Possible to Park a Strand Under Logical Domains as Done in a Non-Logical Domains Environment?	386
Can You Assign Partial Cores to a Sun Netra DPS domain?	386
What Is <code>bss_mem</code> ?	386
What Is the Significance of <code>bss_mem</code> Placement in the Code Listing?	386
How Are <code>app.cmt2board.heap_mem0</code> and Similar Heaps Affected?	387
Can You Clarify BSS, Code, Heap, and DRAM Memory Allocation?	387

Does the <code>eth_*</code> API Support Virtual Ethernet (VNET) Devices?	388
How Do I Calculate the Base PA Address for NIU or Logical Domains to Use with the <code>tnsmctl</code> Command?	388
How Do I Modify the IP Forwarding Application to Use a New Classifier Type Instead of the Default UDP Type?	389
How Do I Add a New Packet Type to <code>ntgen</code> ?	390
Optimization Questions	391
How Do I Enable Optimization?	391
What Is Context-Sensitive Generation?	392
What Is Global Inlining?	392
Legacy Code Integration Questions	392
How Can I Reuse Legacy C Code in a Sun Netra DPS Application?	392
Linking Legacy Code to Sun Netra DPS Code	393
Changing Legacy Source Code	393
How Can I Reuse Legacy C++ Code in a Sun Netra DPS Application?	393
Mixing C and C++ Code	394
Translating C++ Code to C Code	394
Sun CMT Specific Questions	395
Is There a Maximum Allowed Size for Text and BSS in My Program?	395
How Is Memory Organized in the Sun CMT Hardware Architecture?	395
How Do I Increase the Size of the DRAM <code>membank</code> ?	396
Address Resolution Protocol Questions	396
How Do I Enable ARP in the RLP Application?	396
▼ To Enable ARP in RLP	396
How Do I Enable ARP Without Relying on a Control Domain?	397
How Do I Enable ARP Using a Control Domain?	397
Oracle Solaris Domain and Sun Netra DPS Domain Question	398
How Do I Access <code>kstat</code> Information From the Oracle Solaris Domain for Network Interfaces That Are in Use by the Sun Netra DPS domain?	398
Traffic Generation	398

How Do I Stop Traffic Generation?	398
Oracle Solaris TIPC Application	399
What Should I Do When the Oracle Solaris TIPC Application Is Not Able to Create a Socket and Does a Core Dump?	399
Glossary	401
Index	409

Using This Documentation

This user's guide provides information regarding the operation and use of Oracle's Netra Data Plane Software Suite 2.1 Update 1. This document is written for software engineers, developers, programmers, and users who have advanced experience with low-level programming.

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

Related Documentation

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

<http://www.oracle.com/pls/topic/lookup?ctx=E19282-01&id=homepage>

Application	Title	Format	Location
Operation	<i>Sun Netra Data Plane Software Suite 2.1 Update 1 User Guide</i>	HTML, PDF	online
Reference	<i>Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual</i>	HTML, PDF	online
Last-minute information	<i>Sun Netra Data Plane Software Suite 2.1 Update 1 Release Notes</i>	HTML, PDF	online
Documentation Location	<i>Sun Netra Data Plane Software Suite 2.1 Update 1 Getting Started Guide</i>	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-ultrasparc-t1-chip-multithreading-systems.html>
- CoolThreads — CMT Application Tuning and UltraSPARC T2 Server Resources
<http://www.sun.com/servers/coolthreads/tnb/t2.jsp>
- *GNU C Compiler User's Guide*
<http://gcc.gnu.org/onlinedocs>
- GCC for SPARC Systems - Additional command line option flags
<http://cooltools.sunsource.net/gcc/flags.html>
- *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 Oracle VM Server for SPARC software (formerly known as LDoms):
<http://www.sun.com/ldomse>

Documentation, Support, and Training

These web sites provide additional resources:

- Documentation <http://www.oracle.com/technetwork/indexes/documentation>
- Support <https://support.oracle.com>
- Training <https://education.oracle.com>

Sun Netra Data Plane Software Suite Overview

This chapter is an introduction to the Sun Netra Data Plane Software Suite 2.1 Update 1, and provides installation and theoretical information. Topics include:

- [“Product Description” on page 1](#)
- [“Supported Systems” on page 2](#)
- [“Software Installation” on page 3](#)
- [“Building and Booting Reference Applications” on page 10](#)
- [“Programming Methodology” on page 13](#)
- [“tejacc Compiler Basic Operation” on page 15](#)
- [“Architecture Elements” on page 19](#)
- [“User API Overview” on page 29](#)

Product Description

The Sun Netra Data Plane Software (Sun Netra DPS) Suite 2.1 Update 1 is a complete board software package solution. The software provides an optimized rapid development and runtime environment on top of multistrand partitioning firmware for Oracle’s Sun CMT platforms. The software enables a scalable framework for fast-path network processing. Sun Netra DPS 2.1 Update 1 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 Sun Netra Data Plane Software Suite 2.1 Update 1 uses the `tejacc` compiler. `tejacc` is a component of the Teja NP 4.0 Software Platform used to develop scalable, high-performance C applications for embedded multiprocessor target architectures.

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

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

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

Supported Systems

Sun Netra DPS 2.1 Update 1 supports the following Oracle Sun UltraSPARC T1 and UltraSPARC T2 platforms:

- Sun SPARC Enterprise T5120 server
- Sun SPARC Enterprise T5220 server
- Sun SPARC Enterprise T5440 server
- Sun Netra T5120 server
- Sun Netra T5220 server
- Sun Netra T5440 server
- Sun Netra CP3060 ATCA blade server
- Sun Netra CP3260 ATCA blade server

Software Installation

The Sun Netra DPS Suite 2.1 Update 1 is distributed for Oracle's SPARC platforms.

- `Netra_Data_Plane_Software_Suite-2.1-U1.zip` contains the `SUNWndps`, `SUNWndpsd`, `SUNWndps-tipc`, `SUNWndps-tipc-examples`, and `SUNWndps-tipc-headers` packages.
- `Netra_Data_Plane_Software_Suite_Crypto-2.1-U1.zip` contains the `SUNWndpsc` package.

The `SUNWndps` and `SUNWndpsc` packages are installed in the development server. The `SUNWndpsd`, `SUNWndps-tipc`, `SUNWndps-tipc-examples`, and `SUNWndps-tipc-headers` packages are 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
<code>/opt/SUNWndps/bsp</code>	Contains header files and low-level Sun UltraSPARC T1 and Sun UltraSPARC T2 platform initialization and management code.
<code>/opt/SUNWndps/lib</code>	Contains system-level libraries.
<code>/opt/SUNWndps/src</code>	Contains <code>ipfwd</code> , <code>remotecli</code> , <code>rlp</code> , and <code>PacketClassifier</code> reference applications, and network device driver interface header definitions.
<code>/opt/SUNWndps/tools</code>	Contains the compiler and runtime system.
<code>/opt/SUNWndpsc/lib</code>	Contains the Sun Netra DPS Sun UltraSPARC T2 cryptography driver.
<code>/opt/SUNWndpsc/src</code>	Contains Sun Netra DPS Crypto API, IPsec reference application source, and libraries.

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

TABLE 1-2 SUNWndpsd, SUNWndps-tipc, SUNWndps-tipc-examples, and SUNWndps-tipc-headers Package Contents

Directory	Contents
/opt/SUNWndpsd/bin/	Contains the Sun Netra Data Plane CMT/IPC Share Memory drivers which reside on the Oracle Solaris Operating System (Solaris OS) domain. These include: /kernel/drv/sparcv9/tnsm /kernel/drv/tnsm.conf /kernel/drv/sparcv9/tnacl /kernel/drv/tnacl.conf
/opt/SUNWndpsd/lib/	Contains the loadable libraries.
/opt/SUNWndpsd/opt/	Contains user applications and utilities run on the Oracle Solaris domain.
/opt/SUNWndpsd/svc	Contains Sun Netra DPS specific service registry entries.
/opt/SUNWndpsd/linux/src	Contains sources for Linux utilities
/opt/SUNWndpsd/linux/lib	Contains libraries of Linux domain
/opt/SUNWndpsd/linux/bin	Contains binaries executable on the Linux domain.
/opt/SUNWndpsd/etc	Contains service manifest for rcon services in xml files.
/opt/SUNWndps-tipc	Contains TIPC socket library, TIPC configuration utility, and TIPC Oracle Solaris modules.
/opt/SUNWndps-tipc-examples	Contains various TIPC application examples.
/opt/SUNWndps-tipc-headers	Contains TIPC header files.

Platform Firmware Prerequisites

To support Sun Netra Data Plane Software Suite 2.1 Update 1, use the appropriate firmware installed. See *Sun Netra Data Plane Software Suite 2.1 Update 1 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 Oracle's 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 `GCCfss` (GCC for SPARC Systems), Oracle's Java version 1.6.0 and `gmake`. The user must install these packages before applications are built.
- The `SUNWndpsc` `crypto` package requires the `SUNWndps` base package.
- The user 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 currently supported on a SPARC system running the Oracle Solaris 10 Update 5 OS, or above.

Note – The `SUNWndpsd` software package located in the `Netra_Data_Plane_Software_Suite-2.1-U1.zip` file is not installed on the development system. See “[Interprocess Communication Software](#)” on page 89 for details on using this package in a logical domain environment.

Note – If you have previously installed an older version of the Sun Netra Data Plane Software Suite 2.1 Update 1, remove it before installing the new version. See “[To Remove the Software](#)” on page 10.

Note – When installing a new release, ensure that all of the packages within that release are updated. For example, if the `SUNWndps` package is re-installed, ensure that the `SUNWndpsd` package (along with all other packages) are also re-installed. Mismatch of different versions of packages can result in system errors.

Note – The `GCCfss` package (GCC for SPARC Systems 4.3.3) can be downloaded from: <http://www.sun.com/download/>

▼ To Install the Software Into the Default Directory

1. After downloading the Sun Netra Data Plane Software Suite 2.1 Update 1 from the web, as superuser, change to your download directory and go to [Step 2](#).
2. Expand the .zip file. Type:

```
# unzip Netra_Data_Plane_Software_Suite-2.1-U1.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-2.1-U1.zip for crypto drivers. For information on Sun 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 Sun Netra Data Plane Software Suite 2.1 Update 1 from the web, as superuser, change to your download directory and go to [Step 2](#).
2. Expand the zip file. Type:

```
# unzip Netra_Data_Plane_Software_Suite-2.1-U1.zip
```

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

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

The software is installed in *your_directory*.

When using the `pkgadd -R` command, the following warning messages may appear and should be ignored.

```
WARNING:
  The <SUNWcar> package "Core Architecture, (Root)" is a
  prerequisite package and should be installed.
WARNING:
  The <SUNWkvm> package "Core Architecture, (Kvm)" is a
  prerequisite package and should be installed.
WARNING:
  The <SUNWcsr> package "Core Solaris, (Root)" is a prerequisite
  package and should be installed.
WARNING:
  The <SUNWcsu> package "Core Solaris, (Usr)" is a prerequisite
  package and should be installed.
WARNING:
  The <SUNWcsd> package "Core Solaris Devices" is a prerequisite
  package and should be installed.
```

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.

Note – Some reference applications have build scripts and make files hard coded to the default `/opt` install path. If the install directory is not the default path, modify the install path in the build script and make files associated with the application to match the install path.

▼ To Remove the Software

Before installing Sun Netra DPS 2.1 Update 1 software, you must remove previous versions:

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

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

The Sun Netra Data Plane Software Suite 2.1 Update 1 is removed.

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

Building and Booting Reference Applications

The user needs 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.

The `SUNWndps` package requires the `GCCfss` package. Ensure that the correct `PATH` is set and `GCCfss` binaries are used for Sun Netra DPS application compilation.

The Sun Netra DPS application build scripts use `csh`. Therefore, the user `.cshrc` file must contain the correct path setting for `GCCfss`. 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
gccfss compiler is not installed
Please install GCC for Sparc System (see User's Manual)
```

Building Reference Application Instructions

The instructions for building reference applications are located in the individual chapters of this guide. 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 1-3 describes the optional parameters.

TABLE 1-3 Boot Optional Parameters

Option	Description
<i>network_device</i>	The network device used to boot the system.
<i>dhcp bootp</i>	Use DHCP or BOOTP address discovery protocols for boot. Unless configured otherwise, RARP is used as the default address discovery protocol.
<i>server_ip</i>	The IP address of the DHCP, BOOTP, or RARP server.
<i>boot_filename</i>	The file name of the boot script file or boot application image.
<i>client_ip</i>	The IP address of the system being booted.
<i>router_ip</i>	The IP address of a router between the client and server.
<i>boot_retries</i>	Number of times the boot process is attempted.
<i>tftp_retries</i>	Number of times that the TFTP protocol attempts to retrieve the MAC address.
<i>subnet_mask</i>	The subnet mask of the client.
<i>boot_arguments</i>	Additional 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 Sun 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 Sun 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 Sun 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 37](#) 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 yoursrset file2 file3 file4
```

In this example, there are two sets of source files, *mysrcset* and *yoursrset*. The files in *mysrcset* are *file1* and *file2*, and the files in *yoursrset* 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 *yoursrset*. References to *myglobal* within *mysrcset* resolve to the first location, and references to *myglobal* within *yoursrset* 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 yoursrset -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 yoursrset
-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 1-4 lists options to *tejacc*.

TABLE 1-4 Options to *tejacc*

Option	Comment
-include <i>includefile</i>	Where <i>includefile</i> 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 <i>includedir</i>	Where <i>includedir</i> is searched for each file in each source set.
-d <i>destdir</i>	Where the compilation outputs are placed in a directory tree with <i>destdir</i> as the root.

tejacc Compiler Configuration

In addition to the *tejacc* mechanics and options, the behavior of *tejacc* is configured by user libraries that are dynamically linked into *tejacc*.

The libraries describe to `tejacc` the target hardware architecture, the target software architecture, and the mapping of the variables and functions in the source set files to the target architecture. TABLE 1-5 describes some of the configuration options of `tejacc`.

TABLE 1-5 Configuration Options to `tejacc`

Option	Comment
<code>-hwarch myhwarchlib, myhwarch</code>	Load the <i>myhwarchlib</i> shared library and execute the function <i>myhwarch()</i> in it. The execution of <i>myhwarch()</i> creates a memory model of the target hardware architecture.
<code>-swarch myswarehlib, myswareh</code>	Load the <i>myswarehlib</i> shared library and execute the function <i>myswareh()</i> in it. The execution of <i>myswareh()</i> creates a memory model of the target software architecture.
<code>-map mymaplib, mymap</code>	Load the <i>mymaplib</i> shared library and execute the function <i>mymap()</i> in it. Executing the <i>mymap()</i> function in the <i>mymaplib</i> 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`.

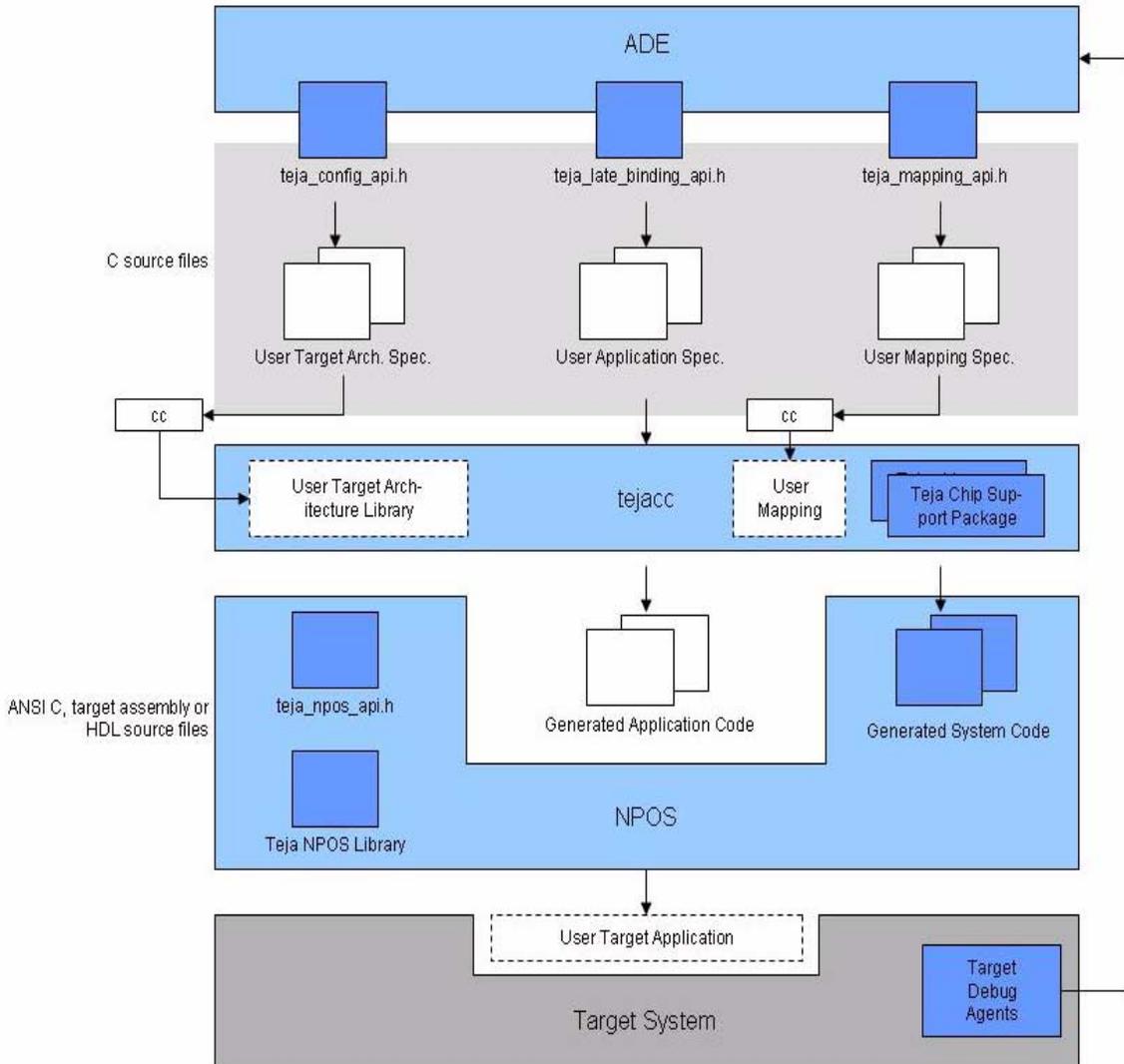
The shared library files can be used for multiple configuration options, but the entry point for each option must be unique, take no parameters, and return an `int`. The trade-off is the ease of maintaining fewer libraries versus the speed of updating only one of several libraries.

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 Sun Netra DPS Interaction

FIGURE 1-1 shows the interaction of `tejacc` with the other platform components of Sun 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 Sun 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 Sun 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 Sun Netra DPS Runtime library include file) or linked to the target-specific Sun Netra DPS Runtime library.

Sun 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 Sun 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()`. The user can assign values to the properties of each function using the `teja_type_set_property()` and `teja_type_get_property()` functions.

TABLE 1-6 describes the basic hardware architecture elements.

TABLE 1-6 Basic Hardware Architecture Elements

Element	Description
Hardware architecture	<p>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 <code>tejacc</code> or defined by the user.</p> <p>Various types of architectures are predefined in the <code>teja_hardware_architecture.h</code> file and are understood by <code>tejacc</code>. The user cannot modify a predefined architecture.</p> <p>User-defined architectures are sometimes desirable to prevent application developers from modifying an architecture. The user can create a user-defined architecture by first populating the architecture and then calling the <code>teja_architecture_set_read_only()</code> function.</p>
Processor	<p>A processor is a target for running an operating system. A processor is contained in an architecture that provides it a name and type.</p>
Memory	<p>A memory is a target for mapping program variables. A memory is contained in an architecture that provides a name and type.</p>
Hardware object	<p>A hardware object is a logic block that is either known to <code>tejacc</code> or is a target for user-defined hardware logic. A hardware object is contained in an architecture that provides it a name and type.</p>
Bus	<p>A bus is used to interconnect elements in a hardware architecture. <code>tejacc</code> uses connection information to validate the user application and reach ability 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.</p>

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 *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*.

Advanced Hardware Architecture Elements

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

[TABLE 1-7](#) describes advanced hardware architecture elements.

TABLE 1-7 Advanced Hardware Architecture Elements

Element	Description
Port	<p>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.</p> <p>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.</p> <p>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 <code>teja_type_set_port()</code> function in the <i>Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual</i>.</p> <p>A port on an architecture might connect to a signal within the architecture as well. See the <code>teja_architecture_set_port_internal()</code> function in the <i>Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual</i>.</p>
Address space and address range	<p>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. <code>tejacc</code> assigns actual values to the address spaces and ranges by resolving these requirements.</p> <p>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 <code>teja_address_space_join()</code> 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.</p> <p>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.</p>

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

The user 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. The user 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

The user declares each of the Late-Binding objects (mutex, queue, channel, and memory pool) using the `teja_type_declare()` function. The user 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 *Sun Netra Data Plane Software Suite 2.1 Update 1 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 1-8 describes the Late-Binding elements

TABLE 1-8 Late-Binding Elements

Late-Binding Element	Description
Mutex	<p>The mutex element provides mutual exclusion functionality and is used to protect critical regions of code.</p> <p>The Late-Binding API for mutex consists of the following:</p> <ul style="list-style-type: none">• <code>teja_mutex_lock()</code> – Lock a mutex.• <code>teja_mutex_trylock()</code> – Try and lock a mutex without blocking.• <code>teja_mutex_unlock()</code> – Unlock a mutex.

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

Late-Binding Element	Description
Queue	<p>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.</p> <p>The Late-Binding API for queue consists of the following:</p> <ul style="list-style-type: none">• <code>teja_queue_dequeue()</code> – Dequeue an element from a queue.• <code>teja_queue_enqueue()</code> – Enqueue an element to a queue.• <code>teja_queue_is_empty()</code> – Check for queue emptiness.• <code>teja_queue_get_size()</code>[†] – Obtain queue size
Memory pool	<p>Memory pools provide an efficient, thread-safe, cross-platform memory management system. This system requires you to subdivide memory in preallocated pools.</p> <p>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 <code>free()</code> and <code>malloc()</code> calls.</p> <p>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.</p> <p>The Late-Binding API for memory pool consists of the following:</p> <ul style="list-style-type: none">• <code>teja_memory_pool_get_node()</code> – Get a new node from the pool.• <code>teja_memory_pool_put_node()</code> – Return a node to the pool.• <code>teja_memory_pool_get_node_from_index()</code> – Provide a pointer to a node, given its sequential index.• <code>teja_memory_pool_get_index_from_node()</code> – Provide the sequential index of a node, given its pointer.

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

Late-Binding Element	Description
Channel	<p>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.</p> <p>Threads sending messages to a channel are called <i>producers</i>, threads receiving messages from a channel are called <i>consumers</i>. 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. The user 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.</p> <p>The Late-Binding API for channel consists of:</p> <ul style="list-style-type: none">• <code>teja_channel_is_connection_open()</code>[‡] – Check if a connection on a channel is open.• <code>teja_channel_make_connection()</code> – Establish a connection on a channel.• <code>teja_channel_break_connection()</code> – Break a connection on a channel.• <code>teja_channel_send()</code> – Send data on a channel.• <code>teja_wait()</code> – 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.

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

Other Elements

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

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

TABLE 1-9 describes other elements.

TABLE 1-9 Other Elements

Other Element	Description
Operating system	An operating system runs on processors and is a target for running processes. An operating system has a name and type. One of the operating system types defined in <code>tejacc</code> 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 *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*.

User API Overview

This section gives an overview of the Sun 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” on page 29](#)
- [“Sun Netra DPS Runtime API Overview” on page 29](#)
- [“Finite State Machine API Overview” on page 31](#)

Late-Binding API Overview

The Late-Binding API is described in [“Software Architecture and Late-Binding API Overview” on page 23](#). 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 *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual* contains API function information.

Sun Netra DPS Runtime API Overview

The Sun 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, Sun 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 Sun Netra DPS Runtime API is provided in the *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*.

Finite State Machine API Overview

The Finite State Machine API enables easy modularization and pipelining of code. Finite state machines are used to organize the control flow of code execution in an application. State machine support is through various macros, which are expanded before they reach `tejacc`. While `tejacc` does not recognize these macros, higher level tools such as the Sun 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 *Sun Netra Data Plane Software Suite 2.1 Update 1 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 1-10](#) describes these relationships.

TABLE 1-10 Mapping of Elements

Elements	Mapping
Functions	Mapped to threads with the <code>teja_map_function_to_thread()</code> function.
Variables	Mapped to memories or process address spaces with the <code>teja_map_variable_to_memory()</code> and <code>teja_map_variables_to_memory()</code> functions.
Processors	Initialized with the <code>teja_map_initialization_function_to_processor()</code> function.
Mapping-specific properties	Assigned with the <code>teja_mapping_set_property()</code> 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.

tejacc Basics

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

- [“Command-Line Options” on page 33](#)
- [“Optimization” on page 35](#)
- [“Language” on page 37](#)

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 <i>hwarch_lib</i> , <i>hwarch_function</i>	The <i>hwarch_function</i> from the dynamic shared library <i>hwarch_lib</i> 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 <i>swarch_lib</i> , <i>swarch_function</i>	The <i>swarch_function</i> from the dynamic shared library <i>swarch_lib</i> 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 <i>map_lib</i> , <i>map_function</i>	The <i>map_function</i> from the dynamic shared library <i>map_lib</i> 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 <i>name</i> [= <i>definition</i>]	Redefines <i>name</i> as a macro, with <i>definition</i> or 1 if not specified. This option is applied to the preprocessing stage of the compilation.
-include <i>includefile</i>	Processes <i>includefile</i> as if #include "file" appeared as the first line of the primary source file.
-I <i>includedir</i>	Adds the directory <i>includedir</i> 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 <i>destdir</i>	Specifies the destination directory for the generated code. The default value is the <i>current_dir</i> /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 tejaccc Options (*Continued*)

Option	Description
-h, ?h, -help, ?help	Prints tejaccc usage.
-srcset <i>srcset_namesrcset_specific_options</i> source_files	Defines a source set consisting of one or more source files. The source set is used to map to one or more processes. <i>srcset_specific_options</i> are applied only to the files listed in the <i>source_files</i> . The -D, -I and -include options are also part of the source set specific options.
-finline= <i>comma separated list of functions</i>	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

This section includes information about optimization options and instructions you can use to enable all of the optimization options at one time or specific options for your situation.

Optimization Options

The user can do the following command-line switches to tejaccc 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 24. 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.
Global inlining	Functions marked with the <code>inline</code> 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 35.

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. The user 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

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 compiler.

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.

Profiler

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

- [“Profiler Introduction” on page 39](#)
 - [“How the Profiler Works” on page 40](#)
 - [“Groups and Events” on page 40](#)
 - [“Profiler Output” on page 41](#)
 - [“Profiler Examples” on page 43](#)
 - [“Profiling Application Performance” on page 46](#)
 - [“User-Defined Statistics” on page 52](#)
 - [“Profiling Metrics” on page 53](#)
 - [“Using the Profiler Script” on page 53](#)
 - [“Profiler Scripts” on page 54](#)
-

Profiler Introduction

The Sun Netra DPS profiler is a set of API calls that help you collect various critical data during the execution of an application. The user 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

The user enables the profiler with the `-pg` command-line option (`tejaccc`). 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. 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.

The user 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. 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 *Sun Netra Data Plane Software Suite 2.1 Update 1 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 *Sun Netra Data Plane Software Suite 2.1 Update 1 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	Represents the CPU ID where the current profiler call was made.
Caller ID	Represents the source location of the <code>teja_profiler</code> call. The <code>your-build-directory/reports/profiler_calls_location.txt</code> file lists all of the IDs and their corresponding source locations. The <code>profiler_calls_location.txt</code> is generated when the application is successfully built.
Call Type	Type of <code>teja_profiler</code> call. The values listed are defined in the <code>teja_profiler.h</code> file.
Completed Cycles	Running total of completed clock cycles so far. The user can use this value to calculate the time between two entries.
Program Counter	Value of the program counter when the current profiler call was invoked.
Group Type	Group number of the events started or being measured.

TABLE 3-1 Profiler Record Fields (*Continued*)

Field	Description
Event Values	<p>Value of the events. This value can be one or more columns depending on the target processor. The target-dependent values are described in the Profiler API chapter in the <i>Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual</i>. The order of the events are the same as the location of the bit set in the event bit mask, passed to <code>teja_profiler_start</code>, starting from left to right. For the entry that represents <code>teja_profiler_start</code>, the values represent the event types. There are two events per record (group) in the dump output:</p> <ul style="list-style-type: none">• <code>event_hi</code> – represents the higher bit set in the event mask• <code>event_lo</code> – represents the lower bit set in the event mask <p>Overflow values consist of the following:</p> <ul style="list-style-type: none">• <code>0x0</code> – no overflow• <code>0x1</code> – overflow of the <code>event_lo</code>• <code>0x2</code> – overflow of the <code>event_hi</code>• <code>0x3</code> – overflow of both <code>event_hi</code> and <code>event_lo</code>
Overflow	<p>Overflow information of one or more events being measured. The value is target-dependent.</p>
User Data	<p>Value of the user-defined data. Zero or more columns, depending on the number of counters allocated and recorded by the user.</p>

Refer to [“Profiler Output Example” on page 45](#) for an example of dump output.

Profiler Examples

For profiler API function descriptions, refer to the *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*.

Profiler API

This section includes profiler API usage for both Sun UltraSPARC T1 and Sun UltraSPARC T2 processors.

Profiler API Usage for the Sun UltraSPARC T1 Processor

The only difference when profiling functions are used for the Sun UltraSPARC T1 processor is in the `teja_profiler_start` function call for CPU group of events. Profiling CPU group on the Sun UltraSPARC T1 processor enables the measuring of only one additional event along with the completed instruction count that is always an available event for this group.

EXAMPLE 3-1 provides an example of profiler API usage for the Sun UltraSPARC T1 processor.

EXAMPLE 3-1 Sample Profiler API Usage for the Sun UltraSPARC T1 Processor

```
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);
            num_pkt = 0;
        }
    }
    teja_profiler_stop(TEJA_PROFILER_CMT_CPU);
}
```

Profiler API Usage for the Sun UltraSPARC T2 Processor

EXAMPLE 3-2 provides an example of profiler API usage for the Sun UltraSPARC T2 processor.

EXAMPLE 3-2 Sample Profiler API Usage for the Sun UltraSPARC T2 Processor

```
main()
{
    /* ...user code... */
    teja_profiler_start(TEJA_PROFILER_CMT_CPU,
                       TEJA_PROFILER_CMT_CPU_IC_MISS |
                       TEJA_
PROFILER_CMT_CPU_DC_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);
            num_pkg = 0
        }
    }
    teja_profiler_stop(TEJA_PROFILER_CMT_CPU);
}
```

Profiler Configuration

You can change the profiler configuration in the software architecture C-based file. 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,ver1.1
CPUID, ID, Type, Cycles, PC, Grp, Evt_Hi, Evt_Lo, Overflow, User Data
4, 15136, 1, 4d048ad5c4, 521f08, 1, 100, 2
4, 30e6, 2, 4d162a0db0, 5128f0, 1, 36c2ba96, ce, 0, 1e8480, 3da594c
4, 18236, 1, 4cf2eb9ce4, 521f08, 1, 100, 1
4, 3a2f, 2, 4d048acb40, 5128f0, 1, 31cffa4, c2a, 0, 1b7740, 3da594c
TEJA_PROFILE_DUMP_END
```

The string, `ver1.1`, 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.

Each profiler record (which normally consists of a lot more lines than the above example) consists of a start delimiter, `TEJA_PROFILE_DUMP_START`, and an end delimiter, `TEJA_PROFILE_DUMP_END`. All profiled data records for a thread are displayed between the start and end delimiter.

In the first record, call type 1 represents `teja_profiler_start`. The values 100 and 1 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 2, respectively.

Cycle counts are accumulative. Thus, 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 18236 and 15136 represent the source location of the profiler API call. The `your-build-directory/reports/profiler_calls_location.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. Sun 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.

Sun UltraSPARC T1 Performance Counters

The CPU, DRAM, and JBus performance counters for Sun UltraSPARC T1 processor are described in [TABLE 3-2](#), [TABLE 3-3](#), and [TABLE 3-4](#), respectively.

TABLE 3-2 Sun UltraSPARC T1 CPU Performance 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. ‡ 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).

TABLE 3-2 Sun UltraSPARC T1 CPU Performance Counters (*Continued*)

Event Name	Description
DTLB_miss	Number of data TLB miss trap taken (includes <code>real_translation</code> misses).
L2_imiss	Number of secondary cache (L2) misses due to instruction cache requests.
L2_dmiss_Id	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.

‡ 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`, `FS-TOX`, `FDTOS`, `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

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

TABLE 3-4 JBus Performance Counters

Event 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. Performance counters are 32 bits wide so they can measure the values in range from 0 to 2^{32} . If measured event has value greater than 2^{32} the corresponding counter will overflow as it will be indicated in the Overflow field of the output record. If the counter will overflow or not depends on properties of the code that is profiled, the clock frequency of the processor, the measured event and the profiling period. In the case of performance counter overflow it is suggested to the user to decrease the profiling period. 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. The user must compile their own aggregate statistics across multiple strands or a core.

Since the JBus and DRAM performance counters are shared across all strands, only one thread should gather these counters.

Sun UltraSPARC T2 Performance Counters

The CPU performance counters for the Sun UltraSPARC T2 processor are described in [TABLE 3-5](#).

TABLE 3-5 Sun UltraSPARC T2 CPU Performance Counters

Event Name	Description
Completed_branches	Number of completed branches.
Taken_branches	Number of branches taken.
FGU_arithmetic_instr	Number of floating-point arithmetic instructions executed.
Load_instr	Number of load instructions executed.
Store_instr	Number of store Instructions executed.
sethi_instr	Number of <code>sethi</code> instructions executed.
Other_instr	Number of all other instructions executed.
Atomics	Number of atomic operations executed.
All_instr	Total number of instructions executed.
Icache_misses	Number of instruction cache misses.
Dcache_misses	Number of L1 data cache misses.
L2_instr_misses	Number of secondary cache (L2) misses due to instruction cache requests.
L2_load_misses	Measures the number of secondary cache (L2) misses due to data cache load requests.
ITLB_ref_L2	For each ITLB miss, this counts the number of accesses the ITLB hardware tablewalk makes to L2 when hardware tablewalk is enabled.
DTLB_ref_L2	For each DTLB miss, this counts the number of accesses the DTLB hardware tablewalk makes to L2 when hardware tablewalk is enabled.
ITLB_miss_L2	For each ITLB miss, this counts the number of accesses the ITLB hardware tablewalk makes to L2 which misses in L2 when hardware tablewalk is enabled. Note: Depending on the hardware tablewalk configuration, each ITLB miss may issue from 1 to 4 requests to L2 to search TSB's.
DTLB_miss_L2	For each DTLB miss, this counts the number of accesses the DTLB hardware tablewalk makes to L2 which misses in L2 when hardware tablewalk is enabled. Note: Depending on the hardware tablewalk configuration, each DTLB miss may issue from 1 to 4 requests to L2 to search TSB's.

TABLE 3-5 Sun UltraSPARC T2 CPU Performance Counters (*Continued*)

Event Name	Description
Stream_LD_to_PCX	Counts the number of SPU load operations to L2.
Stream_ST_to_PCX	Counts the number of SPU store operations to L2.
CPU_LD_to_PCX	Counts the number of CPU loads to L2.
CPU_Ifetch_to_PCX	Counts the number of I-fetches to L2.
CPU_ST_to_PCX	Counts the number of CPU stores to L2.
MMU_LD_to_PCX	Counts the number of MMU loads to L2.
DES_3DES_OP	Increments for each CWQ or ASI operation that uses DES/3DES unit.
AES_OP	Increments for each CWQ or ASI operation which uses AES unit.
RC4_OP	Increments for each CWQ or ASI operation which uses RC4.
MD5_SHA1_SHA256_OP	Increments for each CWQ or ASI operation which uses MD5, SHA-1, or SHA-256.
MA_OP	Increments for each CWQ or ASI modular arithmetic operation.
CRC_TCPIP_Cksum_OP	Increments for each iSCSI CRC or TCP/IP checksum operation.
DES_3DES_Busy_cycle	Increments each cycle when DES/3DES unit is busy.
AES_Busy_cycle	Number of busy cycles encountered per profiling interval when attempting to execute the AES operation.
RC4_Busy_cycle	Number of busy cycles encountered per profiling interval when attempting to execute the RC4 operation.
MD5_SHA1_SHA256_Busy_cycle	Number of busy cycles encountered per profiling interval when attempting to execute the MD5_SHA1_SHA256 operation.
MA_Busy	Increments each cycle when modular arithmetic unit is busy.
CRC_MPA_Cksum	Increments each cycle when CRC/MPA/checksum unit is busy.
ITLB_miss	Includes all misses (successful and unsuccessful tablewalks).
DTLB_miss	Includes all misses (successful and unsuccessful tablewalks).
TLB_miss	Counts both ITLB and DTLB misses (successful and unsuccessful tablewalks).

Note – The final output of the profiler displays the Event names, shown in [TABLE 3-5](#), which are the same as the events listed in *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*.

Each strand has its own set of CPU counters that only tracks its own events and can only be accessed by that strand. Performance counters are 32 bits wide so they can measure the values in range from 0 to 2^{32} . If measured event has value greater than 2^{32} the corresponding counter will overflow as it will be indicated in the Overflow field of the output record. If the counter will overflow or not depends on the properties of the code that is profiled, the clock frequency of the processor, the measured event, and the profiling period. In the case of performance counter overflow, it is suggested to the user to decrease the profiling period.

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. Since a user can measure any two events at a time, in order to measure all 38 CPU counters, 19 measurements are required. The application behavior should be consistent over the entire collection period. To profile each strand on a 64-thread application, each thread must have code to read and set the counters. Sample code is provided in [EXAMPLE 3-2 \(“Sample Profiler API Usage for the Sun UltraSPARC T2 Processor” on page 44\)](#). The user must compile their own aggregate statistics across multiple strands or a core.

The Sun UltraSPARC T2 DRAM Performance Counters are the same as the Sun UltraSPARC T1 DRAM Performance Counters described in [TABLE 3-3](#).

User-Defined Statistics

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

The user can calculate the following metrics after collecting the appropriate hardware counter data using the Sun 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. The user 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 T 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. `profiler_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 Sun 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, d988bfbba, 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 3-6 describes each field in the top section of the summarized Sun UltraSPARC T1 profile data output:

TABLE 3-6 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 3-6 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	8
Cycles	213357798
Store_instr	5157787
L2_instr_misses	549
ITLB_miss_L2	0
CPU_ST_to_PCX	4801072
MA_OP	0
MA_Busy	0
Completed_branches	8346953
Icache_misses	1932
Stream_LD_to_PCX	0
DES_3DES_OP	0
DES_3DES_Busy_cycle	0
Sethi_instr	0
L2_load_misses	59993
DTLB_miss_L2	0
MMU_LD_to_PCX	0
CRC_TCPIP_Cksum_OP	0
CRC_MPA_Cksum	0
Taken_branches	5334546
Dcache_misses	1024428
Stream_ST_to_PCX	0
AES_OP	0
AES_Busy_cycle	0
Other_instr	37370926
FGU_arithmetic_instr	0
ITLB_ref_L2	0
CPU_LD_to_PCX	1779478
RC4_OP	0
RC4_Busy_cycle	0
ITLB_miss	0
Atomics	347142
Load_instr	14564094
DTLB_ref_L2	0
CPU_Ifetch_to_PCX	2603
MD5_SHA1_SHA256_OP	0
MD5_SHA1_SHA256_Busy_cycle	0
DTLB_miss	0
TLB_miss	0
All_instr	65033422
Userdata.1	200000
Userdata.2	0

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 3-7 describes each field in the top section of the summarized Sun UltraSPARC T2 profile data output:

TABLE 3-7 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.
Completed_branches	Number of completed branches per profiling interval.
Taken_branches	Number of branches taken per profiling interval.
FGU_arithmetic_instr	Number of Floating-point arithmetic instructions executed per profiling interval.
Load_instr	Number of Load instructions executed per profiling interval.
Store_instr	Number of Store Instructions executed per profiling interval.
sethi_instr	Number of <code>sethi</code> instructions executed per profiling interval.
Other_instr	Number of all other instructions executed per profiling interval.
Atomics	Number of atomic operations executed per profiling interval.
All_instr	Total number of instructions executed per profiling interval.
Icache_misses	Number of Instruction Cache misses per profiling interval.
Dcache_misses	Number of L1 Data Cache misses per profiling interval.
L2_instr_misses	Number of L2 cache instruction misses per profiling interval.
L2_load_misses	Number of L2 cache load misses per profiling interval.
ITLB_ref_L2	For each ITLB miss, this is the number of accesses the ITLB hardware tablewalk makes to L2 per profiling interval when hardware tablewalk is enabled.
DTLB_ref_L2	For each DTLB miss, this is the number of accesses the DTLB hardware tablewalk makes to L2 per profiling interval when hardware tablewalk is enabled.

TABLE 3-7 Sun UltraSPARC T2 Profile Data Output Field Descriptions (*Continued*)

Field	Description
ITLB_miss_L2	For each ITLB miss, this is the number of accesses the ITLB hardware tablewalk makes to L2 which misses in L2 per profiling interval when hardware tablewalk is enabled. Note: Depending on the hardware tablewalk configuration, each ITLB miss may issue from 1 to 4 requests to L2 to search TSB's.
DTLB_miss_L2	For each DTLB miss, this is the number of accesses the DTLB hardware tablewalk makes to L2 which misses in L2 per profiling interval when hardware tablewalk is enabled. Note: Depending on the hardware tablewalk configuration, each DTLB miss may issue from 1 to 4 requests to L2 to search TSB's.
Stream_LD_to_PCX	Number of SPU load operations to L2 per profiling interval.
Stream_ST_to_PCX	Number of SPU store operations to L2 per profiling interval.
CPU_LD_to_PCX	Number of CPU loads to L2 per profiling interval.
CPU_Ifetch_to_PCX	Number of I-fetches to L2 per profiling interval.
CPU_ST_to_PCX	Number of CPU stores to L2 per profiling interval.
MMU_LD_to_PCX	Number of MMU loads to L2 per profiling interval.
DES_3DES_OP	Number of increments for each CWQ or ASI operation which uses DES/3DES unit per profiling interval.
AES_OP	Number of increments for each CWQ or ASI operation which uses AES unit per profiling unit.
RC4_OP	Number of increments for each CWQ or ASI operation which uses RC4 per profiling interval.
MD5_SHA1_SHA256_OP	Number of increments for each CWQ or ASI operation which uses MC5, SHA-1, or SHA-256 per profiling interval.
MA_OP	Number of increments for each CWQ or ASI modular arithmetic operation per profiling interval.
CRC_TCPIP_Cksum_OP	Number of increments for each iSCSI CRC or TCP/IP checksum operation per profiling interval.
DES_3DES_Busy_cycle	Number of increments per profiling interval for each cycle when DES/3DES unit is busy
AES_Busy_cycle	Number of busy cycles encountered per profiling interval when attempting to execute the AES operation.
RC4_Busy_cycle	Number of busy cycles encountered per profiling interval when attempting to execute the RC4 operation.
MD5_SHA1_SHA256_Busy_cycle	Number of busy cycles encountered per profiling interval when attempting to execute the MD5_SHA1_SHA256 operation.

TABLE 3-7 Sun UltraSPARC T2 Profile Data Output Field Descriptions (*Continued*)

Field	Description
MA_Busy	Number of increments per profiling interval for each cycle when modular arithmetic unit is busy.
CRC_MPA_Cksum	Number of increments per profiling interval for each cycle when CRC/MPA/checksum unit is busy.
ITLB_miss	Number of misses (successful and unsuccessful tablewalks) per profiling interval.
DTLB_miss	Number of misses (successful and unsuccessful tablewalks) per profiling interval.
TLB_miss	Number of both ITLB and DTLB misses, including successful and unsuccessful tablewalks 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

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 Sun Fire T2000 system)

Sun UltraSPARC T1 Processor

Instructions per Packet:

Average number of instructions executed in a packet.

Formula: $\text{value} = (\text{Instr_cnt} / \text{pkts_per_interval})$

Instructions per Cycle (IPC):

Average number of instructions executed per cycle.

Formula: $\text{value} = (\text{Instr_cnt} / \text{cycle})$

Packet Rate:

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

Formula: $\text{value} = ((\text{pkts_per_interval} / (\text{cycle} / \text{cpu_frequency})) / 1000)$

SB_full per thousand instructions:

Average number of SB_full occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{SB_full} / \text{Instr_cnt}) * 1000)$

FP_instr_cnt per thousand instructions:

Average number of FP_instr_cnt occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{FP_Instr_cnt} / \text{Instr_cnt}) * 1000)$

IC_miss per thousand instructions:

Average number of IC_miss occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{IC_miss} / \text{Instr_cnt}) * 1000)$

DC_miss per thousand instructions:

Average number of DC_miss occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{DC_miss} / \text{Instr_cnt}) * 1000)$

ITLB_miss per thousand instructions:

Average number of ITLB_miss occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{ITLB_miss} / \text{Instr_cnt}) * 1000)$

DTLB_miss per thousand instructions:

Average number of DTLB_miss occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{DTLB_miss} / \text{Instr_cnt}) * 1000)$

L2_imiss per thousand instructions:

Average number of L2_miss occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{L2_miss} / \text{Instr_cnt}) * 1000)$

L2_dmiss_LD per thousand instructions:

Average number of L2_Dmiss_LD occurrences per 1000 instructions executed.

Formula: $\text{value} = ((\text{L2_miss} / \text{Instr_cnt}) * 1000)$

Sun UltraSPARC T2 Processor

Instruction per Packet:

Average number of instructions executed in a packet.

Formula: $\text{value} = (\text{All_instr} / \text{pkts_per_interval})$

Instructions per Cycle (IPC):

Average number of instructions executed per cycle.

Formula: $\text{value} = (\text{All_instr} / \text{cycle})$

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: $\text{value} = (\text{Store_instr} / \text{pkts_per_interval})$

Load Instructions per Packet:

Average number of Load instructions executed per packet.

Formula: $\text{value} = (\text{Load_instr} / \text{pkts_per_interval})$

L2 Load misses per Packet:

Average number of L2 cache Load misses per packet.

Formula: $\text{value} = (\text{L2_load_misses} / \text{pkts_per_interval})$

Icache misses per 1000 Packets:

Average number of L1 Icache misses per 1000 packet.

Formula: $\text{value} = (\text{Icache_misses} \times 1000) / \text{pkts_per_interval}$

Dcache misses per Packet:

Average number of L1 Icache misses per packet.

Formula: $\text{value} = (\text{Dcache_misses} / \text{pkts_per_interval})$

Packet Rate:

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

Formula: $\text{value} = ((\text{pkts_per_interval} / (\text{cycle} / \text{cpu_frequency})) / 1000)$

Note – Not all possible parameters are shown here. The user 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* (for UltraSPARC T1) or by *profiler_n2.pl* (for UltraSPARC T2).

2. Insert formulas into the spreadsheet.

See the `sample_analysis.sxc` spreadsheet provided as part of the software package. You can open with an OpenOffice compatible software. This file is included in the `SUNWndps/src/libs/profile` directory. The first spreadsheet in this template (click on the Output from profile script tab) consists of sample output generated from

Step 1. The second spreadsheet in this template (click on the Analysis tab) consists of formulas for computing the data in the first spreadsheet. The format in the Analysis spreadsheet is designed so that you can compare the data generated on each thread side by side.

3. Save the spreadsheet for future reference.

You can form your own spreadsheet templates for your own analysis. For example, each application can have its own data imported to a spreadsheet for analysis.

Debugger

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

- [“Debugger Introduction”](#) on page 69
- [“Native Debugger”](#) on page 70
- [“GNU Project Debugger”](#) on page 80

Debugger Introduction

The Sun 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 Sun Netra DPS native debugger, press Ctrl-C.

To use the GNU Debugger (GDB), you must have their own source code and the binary. You must turn on the flag for this application, for example,
`USR_CFLAGS = -DTEJA_DEBUGGER_MODE=TEJA_DEBUGGER_GDB_MODE`

See [“GNU Project Debugger”](#) on page 80 for detailed setup and example information.

Native Debugger

The native debugger runs on the target and enables you 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 Using Options” on page 79](#).

The native debugger is denoted by `dbg`. See [“Native Debugger Commands” on page 71](#).

Debugging Configuration Code

As seen in [“tejacc Compiler Configuration” on page 16](#), `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 software. 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 Oracle 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

The following section contains descriptions of the native debugger commands.

Displaying Help

`help` or `h`

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` or `b address`

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 Using Options” on page 79](#).

Example:

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

`info break` or `i break`

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` or `d breakpoint ID`

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

Example:

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

Managing Program Execution

`cont` or `c`

Continues execution of the application.

Example:

```
dbg>cont
```

`step` or `s`

Steps to the next assembly instruction within the application.

Example:

```
dbg>step
```

Note – 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`

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

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 or *i threads*

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

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

Example:

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

In “[info threads](#) or [i threads](#)” on [page 75](#) example, the focus (F) was on `classifiertthread`, with Teja ID of 1. In this example, the focus has been moved to `generatorthread`.

Displaying Registers

```
info reg or i reg
```

Displays the register contents for the thread in focus. Refer to the *UltraSPARC T1 Supplement to the UltraSPARC Architecture* and the *UltraSPARC T2 Supplement to the UltraSPARC Architecture* for detailed descriptions of these registers when using the native debugger on the UltraSPARC T1 and UltraSPARC T2 platform. In the following example, the `tpc` (program counter at trap point) is at `0x508c88`. The `tt` (trap type) is `0x7c`. The content of the code at address `0x508c88` can be located by an elf file dump utility, such as *gobjdump* (in `SUNWbinutils` package) or equivalent utility.

```
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 000000000520c30 000000010e01bc8
i[4] : 0000000000000000 0000000000000000 000000010e00d91 00000000051458c

O registers:

o[0] : 000000000000006e 000000000520c30 000000010e01bc8 0000000000000000
o[4] : 000000000600000 0000000000000061 000000010e00cd1 000000000514a18

L registers:

l[0] : 000000000000006e 000000010e0172c 00000000051e8f0 ffffffffef1fe8d4
l[4] : 000000000520c30 0000000000000000 0000000000000000 0000000000000000
gl : 0000000000000001
tl : 0000000000000001
tt : 000000000000007c
tpc : 000000000508c88
tnpc : 000000000508c8c
tstate : 000009914001600
pstate : 000000000000014
tick : 000001884f873558
tba : 000000000500000
asi : 000000000000014

Displaying Stack Trace

`bt frame-count`

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

```
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] : 000000000002000 0000000019ae8ec8 0000000010e02e31 00000000050e888
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 00000000050fcc4
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] : 000000000002000 0000000019ae8ec8 0000000010e02fd1 00000000050fbd8
frame 4, sp 0x10e037d0, call instruction at 0x50e104:
l[0] : ffffffff00000000d8 ffffffff000000fba 0000000000000003 0000000000000000
l[4] : 00000000006170d8 0000000000617000 0000000000000617 0000000000000400
i[0] : 00000000111000e0 000000000000792d 000000000000792d 0000000011100180
i[4] : 000000000000792d 0000000000000000 0000000010e03081 00000000050e104
```

Resolving Symbols Using Options

You can use the `dbghelper.pl` script 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`

Displays help information.

`-f function-name`

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.

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

`-g global-variable`

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.

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

-1 *file-name:line-number*

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.

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

GNU Project Debugger

GDB, the GNU Project debugger, enables you to debug the program in C source code level. The following sections describe the reference Sun Netra DPS application (gdb showcase application) that showcases the GDB support in Sun Netra DPS over the Logical Domain Channel (LDC). In this release, only the IPFwd and GDB showcase applications have been prepared for gdb support. Other applications can easily be instrumented by following these examples.

Configuring Oracle VM Server for SPARC Software for GDB Support

GDB requires the Oracle VM Server for SPARC software. If this product is not installed, download it at: <http://www.sun.com/ldoms>

▼ To Configure the Oracle VM Server for SPARC Software Required to Run the Sun Netra DPS Application With GDB Support

The GDB currently runs over LDC only, not over IPC.

The GDB uses the vdpcc service named `ndps-cli` and the corresponding client named `solaris-cli`. This service-client pair can be created either by using the `ldm` commands in this procedure or by using the respective commands in the auto configuration process.

- Execute the following commands:

```
# /opt/SUNWldm/bin/ldm add-vdpcs ndps-cli ndps-domain-name  
# /opt/SUNWldm/bin/ldm add-vdpc solaris-cli ndps-cli solaris-domain-name
```

▼ To Configure the Oracle Solaris Domain for GDB

After the logical domains are configured and running, perform the following steps to configure the gateway for GDB in the Oracle Solaris domain.

1. Ensure that the `SUNWndpsd` package is installed in the domain.
2. Load the driver:

```
# rem_drv remldc
# add_drv remldc
```

3. Execute the following commands:

```
# echo "remotegw 34980/tcp" >> /etc/services
# svccfg import /var/svc/manifest/network/remotegw.xml
# svcadm enable svc:/network/remotegw:remotegw
```

GDB Showcase Application

The Sun Netra DPS package contains a simple test application to showcase the use of `gdb`. This application is used as shown in [“To Compile the GDB Showcase” on page 82](#).

▼ To Compile the GDB Showcase

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

```
% 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 Oracle Solaris utility binaries are under `src/apps/gdb/solaris-gw/ldc_so`.

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

1. Verify that you have copied your GDB Sun Netra DPS binary `main` into your installation server under `/tftpboot`.
2. Execute the following at the Sun Netra DPS domain OpenBoot PROM `ok` prompt:

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

3. If the file was compiled without the `-o` option, continue from the initial break point.

▼ To Run the GDB Command

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

```
% cd src/apps/gdb/code/main  
% /opt/SUNWndps/bin/gdb main
```

Where `main` is the same binary code that you loaded into your Sun 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)
```

2. Acquire the host name or IP address for the Oracle Solaris domain:

```
% ifconfig -a
```

Assuming your IP address is: 10.1.1.249

```
(gdb) target remote tcp:10.1.1.249:34980
```

This connects to your Sun Netra DPS GDB application. The Sun 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 GDB commands in [“GDB Commands” on page 84](#) to investigate your application.

GDB Commands

GDB commands include the following:

- `target remote tcp:10.1.1.194:34980` – Connects to remote Sun Netra DPS target
- `info thread` – Displays threads
- `thread #` – Switches thread
- `info reg` – Shows the register files
- `info break` – Shows the breakpoint
- `b #` – Sets breakpoint
- `d #` – Clears breakpoint
- `c` – Continues
- `s` – Steps
- `x` – Checks memory location
- `p` – Displays variable
- `list` – Displays source code, for example, `list debug_func`

- `bt` – Backtraces
- `detach` – Enables the Oracle Solaris gateway program to exit which ends the remote communication. See “[To Run Sun Netra DPS Application With GDB Support](#)” on page 85 for more details.

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

▼ To Run Sun Netra DPS Application With GDB Support

As an example, to run the IPFwd application with GDB support, perform the following steps.

1. **Go to `src/apps/ipfwd` and compile with `gdb` as one of the arguments in the command line.**

For example:

```
% ./build cmt2 10g_niu ldoms gdb
```

2. **Load the `src/apps/ipfwd/code/ipfwd/ipfwd` binary into your Sun Netra DPS domain.**
3. **Configure the Oracle Solaris gateway in the Oracle Solaris domain (for example, 10.1.1.194).**
4. **In the Oracle Solaris domain, calculate the `basepaddr` and run the `tnsmctl -P -v` command, if the binary booted in the Sun Netra DPS domain uses NIU.** See “[How Do I Calculate the Base PA Address for NIU or Logical Domains to Use with the `tnsmctl` Command?](#)” on page 388.
5. **Run the commands shown in “[To Configure the Oracle Solaris Domain for GDB](#)” on page 82.**
6. **Go to `/opt/SUNWndps/bin` and run the `gdb` binary from the Oracle Solaris domain:**

```
% ./gdb ipfwd
```

`ipfwd` is the same binary code that you loaded into your Sun Netra DPS domain. The following example output from the `gdb` showcase application shows the usage of all of the GDB commands.

```

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:10.1.1.249:34980
Remote debugging using tcp:10.1.1.249:34980
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.

```

7. After debugging is completed, type `detach`.

```
(gdb) detach
```

Note – After completed, always type `detach` in `gdb`. Otherwise, the `remotegw` process is left with an outdated state in the Oracle Solaris domain. If this happens, stop the `remotegw` process using the `svcadm` command (`svcadm disable remotegw`) before you start again.

8. (Optional) Reload your binary in the Sun Netra DPS domain.

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

Interprocess Communication Software

This chapter describes the Interprocess Communication (IPC) software. Topics include:

- [“IPC Introduction” on page 89](#)
- [“Programming Interfaces Overview” on page 90](#)
- [“Configuring the Environment for IPC” on page 90](#)
- [“Example Environment for UltraSPARC T1 Based Servers” on page 94](#)
- [“Example Environment for UltraSPARC T2 Based Servers” on page 98](#)
- [“IPC Reference Applications” on page 99](#)

IPC Introduction

The Interprocess Communication (IPC) mechanism provides a means to communicate between processes that run in a domain under the Sun 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 logical domains environment in which the IPC mechanism can be used, and explains the IPC specific portions of the IP forwarding reference application (see [“Reference Applications” on page 163](#)).

Programming Interfaces Overview

Chapter 5, Interprocess Communication API, of the *Sun Netra Data Plane Software Suite 2.1 Update 1 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 Sun 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 Oracle 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 Oracle 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 logical domains environment, and the IPC channels.

Memory Management

The IPC framework shares its memory pools with the basic logical domains 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 Sun Netra DPS runtime. This declaration is done in the software architecture definition in `ipfwd_swarch.c`.

IPC in the Logical Domains Environment

In a logical domains 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 Oracle VM Server for SPARC 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 [“Logical Domain Channel Setup” on page 91](#). Each channel has a server defined in the `LWRTE` domain and a client defined in the link partner domain.

Logical Domain 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 tnsn-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-vdpc 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-vdpc 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 Oracle 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 `SUNWndpd` 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
<code>-S</code>	Set up IPC channel.
<code>-C channel-id</code>	Channel ID of the new channel to be set up.
<code>-L local-ldc</code>	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 <code>ldm list-bindings</code> command.
<code>-R remote-ldc</code>	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 <code>ldm list-bindings</code> command with the <code>-e</code> flag.
<code>-F control-channel-id</code>	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 <code>-C</code> and the <code>-F</code> options.

The `tnsm` driver stores the channel configuration so it can be replayed when the Sun Netra DPS domain reboots. This stored configuration can be purged through the following command:

```
# tnsmctl -p
```

Note – This option clears the stored configuration, but does not affect the currently operating channels.

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 5-2 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 Oracle Solaris guest domains.
ldg1	Owns the other PCI bus (<code>bus_b</code>) 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 <code>tnsmctl</code> utility is used here to set up IPC channels.

The primary as well as the guest domains `ldg2` and `ldg3` run the Oracle Solaris 10 11/06 operating system (or later) with the patch level required for logical domain operation. The `SUNWldm` package is installed in the primary domain. The `SUNWndpsd` package is installed in both `ldg2` and `ldg3`.

Assuming 4-GByte 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. Oracle Solaris is installed once the domains are bound and started in a manner described in the Oracle VM Server for SPARC software documentation.

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-udpcc tnsn-gc0 primary-gc ldg3
```

Client Control Channel

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

Data Channel

```
ldm add-vdpcs ldg2-udpcc0 ldg1
ldm add-udpcc udpcc0 ldg2-udpcc0 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
	<code>config-tnsm-ldg2</code>	<code>config-tnsm0@ldg2</code>	6

For `ldg2`:

VDPCS	NAME	SERVICE	LDC
	<code>config-tnsm0</code>	<code>config-tnsm-ldg2@ldg1</code>	5

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

```
# tnsctl -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:

```
# tnsctl -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 94](#) 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 94](#), the primary domain owned one of the PCI buses (`bus_a`), while the Sun 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 Sun Netra DPS Runtime Environment domain (`ldg1`) so that the `LWRTE` domain can utilize NIU for fast packet processing applications.

In addition, the IP forwarding and RLP reference applications can use up to fifty six threads in the UltraSPARC T2 logical domain configurations depending on the configuration, so the Sun Netra DPS Runtime Environment domain must be sized accordingly.

IPC Reference Applications

The Sun Netra DPS package contains an IP forwarding reference application that uses the IPC mechanism. The Sun Netra DPS package contains an IP forwarding application in `LWRTE` and an Oracle Solaris utility that uses an IPC channel to upload the forwarding tables to the `LWRTE` domain. Sun Netra DPS chooses which table to use and where to gather some simple statistics, and displays the statistics in the Oracle Solaris domain. The application is designed to operate in the example setup shown in [“IPC Channels” on page 97](#).

Refer to [“IP Packet Forwarding Reference Applications” on page 164](#) for details on how the IPC mechanism is used.

Common Header

The common header file `fibtable.h`, located in the `src/common/include` subdirectory, contains the data structures shared between the Oracle 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.

Remote Command-Line Interface

This chapter describes the Remote Command-Line-Interface (CLI). Topics include:

- [“Remote Command-Line Interface Introduction” on page 101](#)
- [“IPC Setup for Remote CLI” on page 102](#)
- [“Accessing the Remote CLI” on page 103](#)
- [“Debugging Remotely” on page 105](#)
- [“Coredump Support” on page 106](#)
- [“System Configuration” on page 106](#)

Remote Command-Line Interface Introduction

The Remote Command-Line Interface (CLI) provides you remote access to commands for you to configure and gather the Sun Netra DPS runtime system information (for example, platform information). The CLI also provides you remote access to the Sun 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 Oracle 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 `fastpath_mgr_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`

Note – The IP Forwarding (`ipfwd`) application has the Remote CLI functionality built in. You can use IP Forwarding as a reference on how an application enables Remote CLI functionality.

▼ To Configure the Oracle Solaris Domain for Remote CLI

After the logical domains are configured and running, perform the following steps to configure the IPC channel in the Oracle Solaris domain:

1. On the primary domain, execute the following commands:

```
# ldm add-vdpcs ldg2-vdpcs-cli <ndps-domain>
# ldm add-vdpsc vdpcc-cli ldg2-vdpcs-cli <solaris-domain>
```

These commands set up an IPC data channel between *ndps-domain* and *solaris-domain*. The CLI server (*ldg2-vdpcs-cli*) runs on *ndps-domain* and the CLI client (*vdpcc-cli*) runs on *solaris-domain*. Prior to execute these commands, ensure that the Global and Client Control Channels are already setup (see Chapter5 Interprocess Communication Software).

2. On the domain that has access to the Global Control Channel, execute:

```
# tnsmtcl -S -C 3 -L <local_ldc> -R <remote_ldc> -F 3
```

local_ldc is the LDC number of the Client Control Channel at the server (*ndps-domain*) side.

remote_ldc is the LDC number of the Client Control Channel at the client (*solaris-domain*) side.

3. On the domain that has access to the Global Control Channel, execute:

```
# tnsmtcl -S -C 1 -L <local_ldc> -R <remote_ldc> -F 3
```

local_ldc is the LDC number of the Data Channel at the server (*ndps-domain*) side.

remote_ldc is the LDC number of the Data Channel at the client (*solaris-domain*) side.

The channel ID assigned to this data channel is 1 (the dedicated channel number of the remote CLI communication over IPC in the Sun Netra DPS).

Accessing the Remote CLI

After IPC channel number 1 is set up between Sun Netra DPS Runtime and the remote CLI Solaris host system, enable remote CLI service in the Oracle Solaris host system. Enable remote CLI service with the command below:

```
# svcadm enable rcon
```

After enabling remote CLI service, you are ready to access the remote CLI.

▼ To Access the CLI Console

1. Connect to the Oracle Solaris CLI host system.

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

```
% telnet solaris-domain-host-name 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 lwrt core
                        [-d <dump dir>] dump directory (default: "/tmp")
                        <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

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

▼ To Access the Sun 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 105](#)). From the `dbg` mode, use the `coredump` command to dump the Sun Netra DPS Runtime 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 Oracle Solaris system. By default, the core is saved in `/tmp`.

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 "/tmp"
Total dumped: 74024954 bytes, compressed to: 456741 bytes
finished coredump successfully!
dbg>
```

The preceding core file is created at `/tmp/core-1.gz` on the remote CLI host system (`solarisdomain`). Note that this can take up to several minutes due to the size of the core dump file.

System Configuration

The user 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 103](#).

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>
```

Compiling the Remote CLI Application

Build Script

TABLE 6-1 shows the `remotecli` application build script.

TABLE 6-1 Remote CLI Application Build Scripts

Build Script	Usage
<code>./build</code> (See “Argument Descriptions” on page 108)	Build <code>remotecli</code> application.

Usage

build *cmt* [*profiler*]

Build Script Arguments

[] – Optional arguments

Argument Descriptions

cmt

Specifies whether to build the `remotecli` application to run on the CMT1 (UltraSPARC T1) platform or CMT2 (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>`.

[*profiler*]

Generate codes with profiling enabled.

The above creates the bootable image at `code/main/main`.

Note – The `remotecli` application is a simple application to demonstrate remote CLI functions. The application does not perform any particular useful task. For details on how to integrate remote CLI functions into a large scale application, refer to the IP Forwarding application.

Eclipse Development Environment

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

- “ADE Introduction” on page 109
- “Starting the Eclipse-Based ADE GUI” on page 110
- “Creating a Teja Project” on page 110
- “Files and Viewers” on page 115
- “Build” on page 121

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](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.

Note – Before running the `eclipse.sh` script, check if the Eclipse binary has already been installed in the user's system. If not, download and install Eclipse before proceeding.

▼ 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.

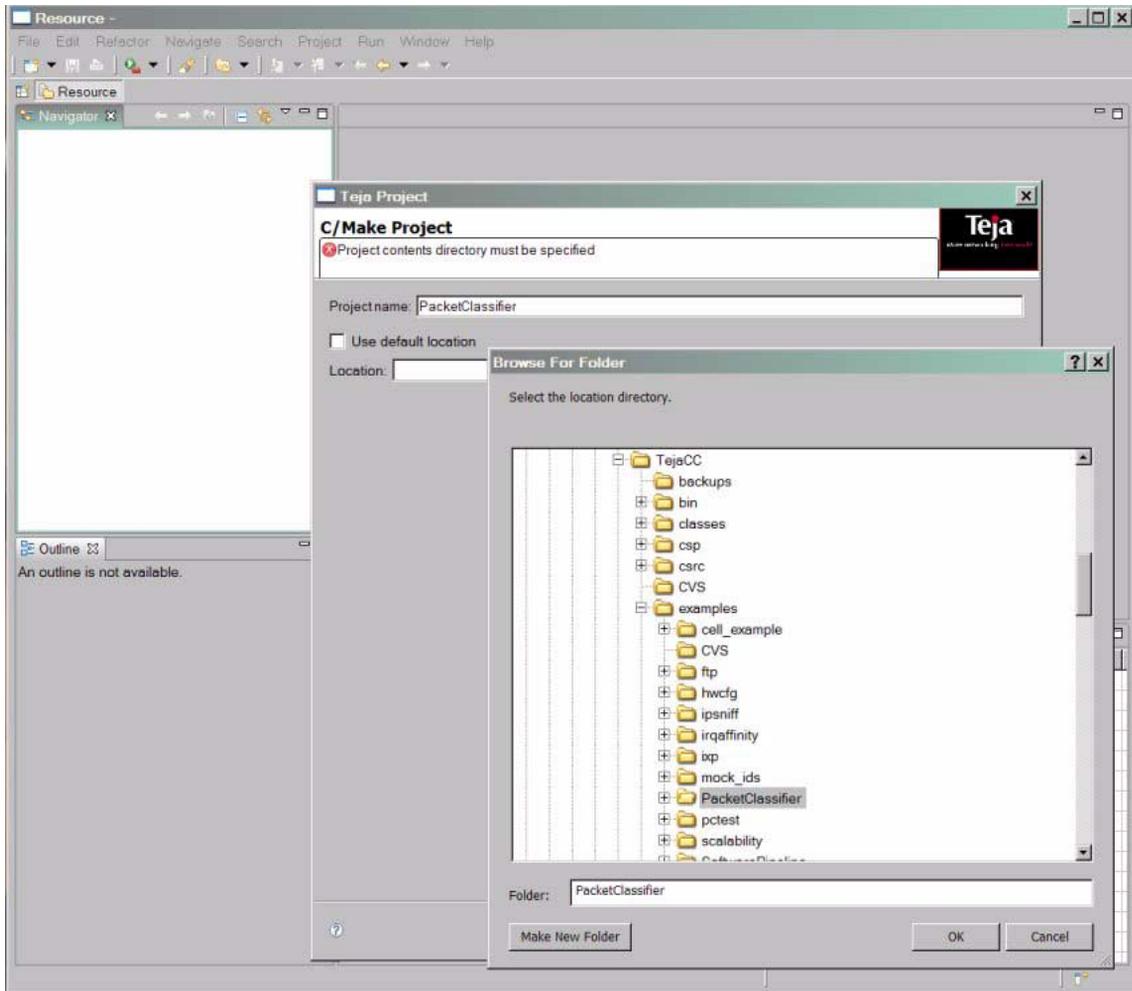
b. Click the Browse button to get to the PacketClassifier directory.

c. Press OK.

Keeping Use default selected would create the project in the workspace (see [FIGURE 7-1](#)).

d. Click Next to go to the C/ Make Project Setting tab.

FIGURE 7-1 Eclipse-Based ADE GUI



5. The user does not need to set the C/Make Project Setting tab, which defines the project and Builder settings, at this point.

a. Click Next to go to the Teja Project Settings tab.

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`. Select whether to generate all three graphic files, or a subset, by putting a tag in the list in the General section.

Populate the Teja Project Setting (FIGURE 7-2) tab in two ways:

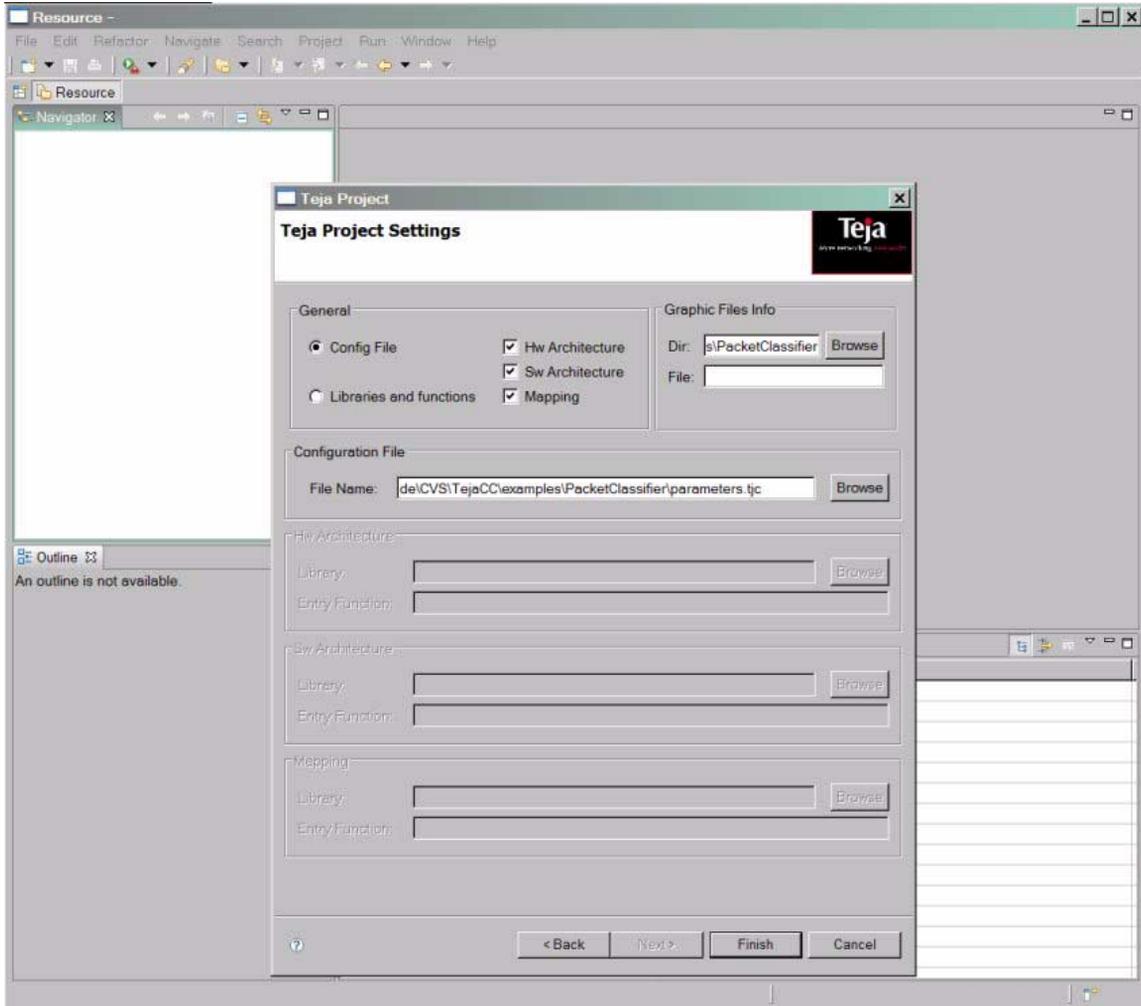
- By specifying a configuration file, selecting the `Config file` button in the General section, and providing the name in the Configuration File section. This file is generated by `tejacc` with the name `parameters.tjc` contains all the information on the parameters `tejacc` was invoked (use the `parameters_file` 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.

Note – To generate the `parameters.tjc` file, first build the application. The `parameters.tjc` file is generated in the top level directory of the application where the build (or make) is executed.

- 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 click Finish. The project is created without the graphic files, which can be added in a second step. From the Welcome Window, switch to the Navigator Window to view project elements.

▼ 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.

Note – To go to the Navigator view from the Welcome Window, click on the Workbench button.

2. Open `New/Other/Teja/Teja Graphic Files` and press **Next**.

The Teja Project Settings tab appears. Fill this out as described in “To Create a Project in the Same Directory as an Existing Teja Application” on page 110.

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

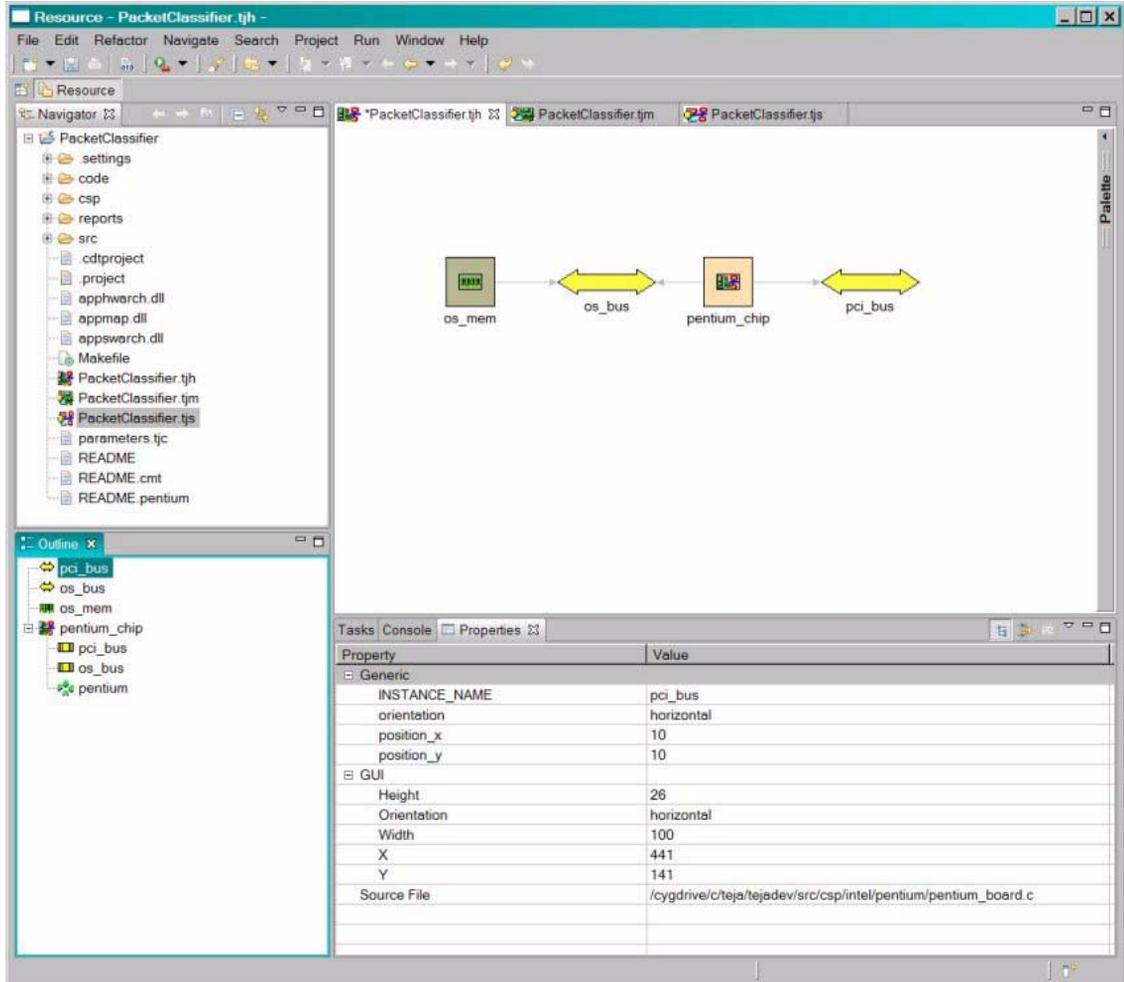
The Hardware Architecture Viewer (FIGURE 7-3) gives a graphical representation of the hardware architecture. Since hardware architectures can contain other hardware architectures, you can navigate the containment by double-clicking on architectures. The Outline tab provides a more straightforward visualization 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.

FIGURE 7-3 PacketClassifier Hardware Architecture – Inner Hardware

The screenshot shows a hardware design tool interface. The main workspace contains a diagram of the inner hardware architecture. It features three components: a yellow rectangle labeled 'pci_bus' on the left, a green square labeled 'pentium_chip.pentium' in the center, and another yellow rectangle labeled 'os_bus' on the right. Red arrows indicate bidirectional connections between 'pci_bus' and 'pentium_chip.pentium', and between 'pentium_chip.pentium' and 'os_bus'. The left sidebar shows a project tree with 'PacketClassifier.tjs' selected. The bottom right shows a Properties window with various parameters.

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/cspi/intel/pentium/pentium_board.c

FIGURE 7-4 PacketClassifier Hardware Architecture – Outer Hardware

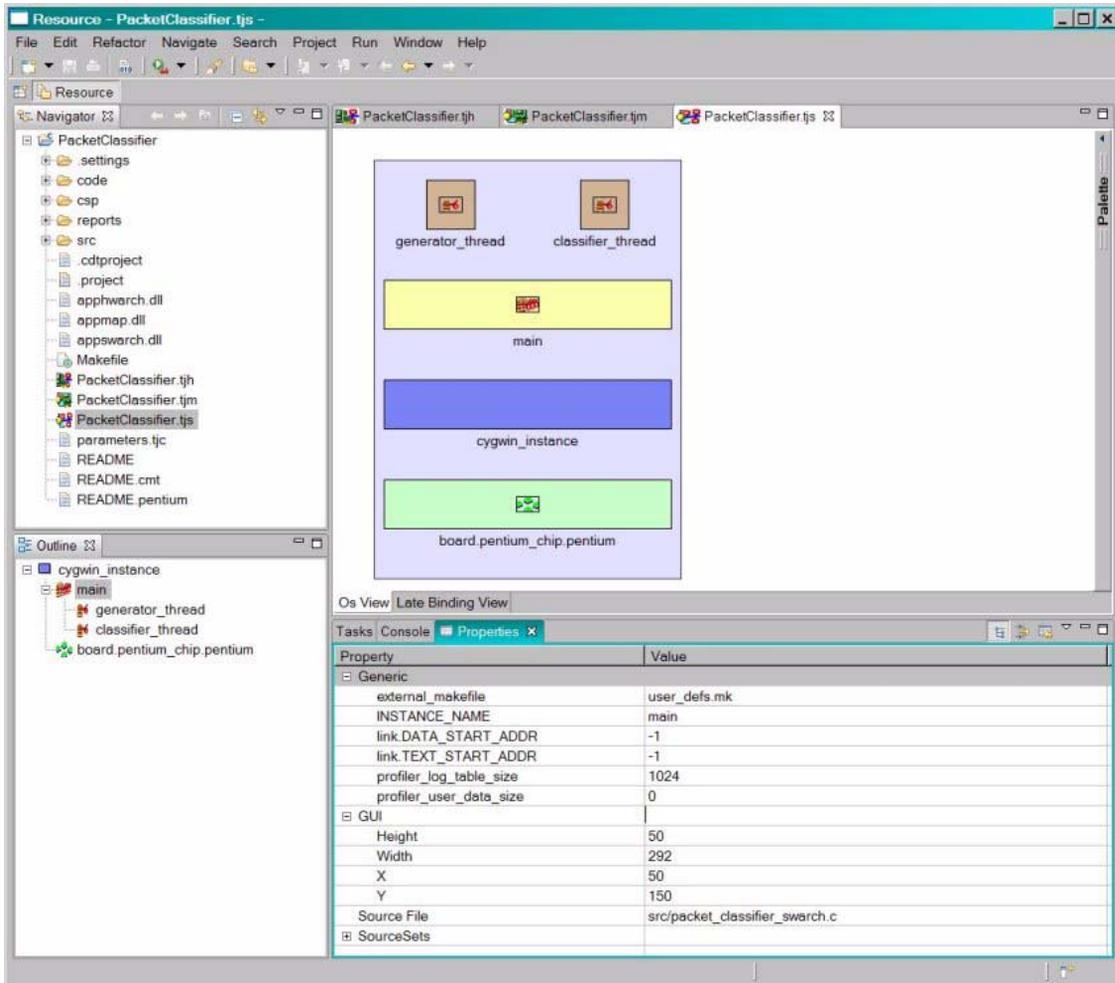


Sun Netra DPS objects have properties with values of potential interest. The Properties tab displays such properties and their values. To open the Properties tab, go to the Window/Show view/Others/Properties menu. Along with the application properties there are also the GUI properties, some of which 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.

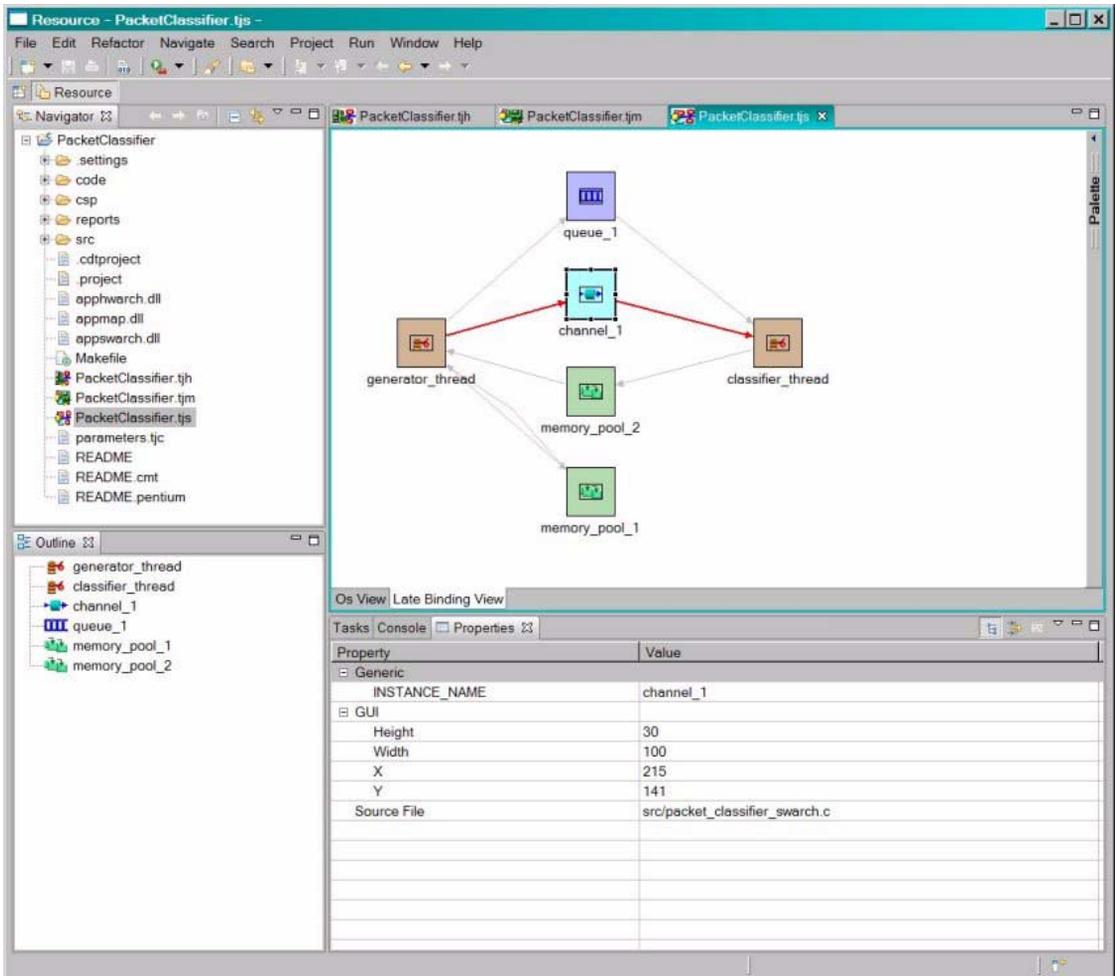
FIGURE 7-5 PacketClassifier Software Architecture – OS View



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 115.)

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

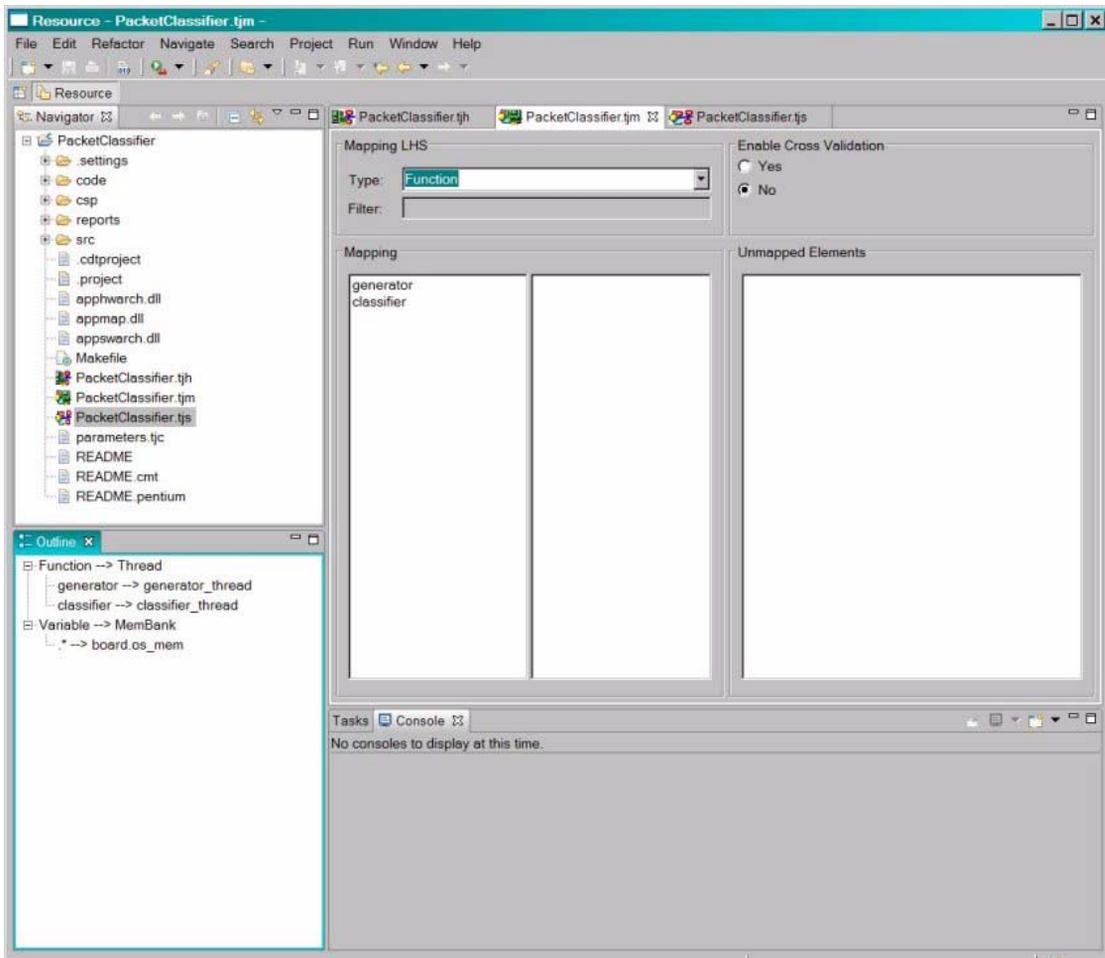


Mapping Viewer

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



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 `regex` 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.

Receive Packet Classification

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

- [“Receive Packet Classification Introduction” on page 123](#)
- [“Sun Multithreaded 10GbE and NIU Receive Packet Classifier” on page 124](#)
- [“Hashing Based on Layer 2, Layer 3, and Layer 4 Header Classification” on page 128](#)
- [“Flow Match Based on Layer 2, Layer 3, and Layer 4 Header Classification” on page 131](#)
- [“Examples” on page 137](#)

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. Sun 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 and 4GbE
- 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 Layer 2, Layer 3, and Layer 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.

Receive DMA Channel Selection

In Sun Multithreaded 10-Gb Ethernet and NIU, there are a total of 16 Receive DMA Channels (RDCs) in hardware. These Receive DMA channels are organized into Receive DMA Channel Groups (RDC Groups). Each RDC Channel Group can have

up to 16 RDC entries. During receive, a RDC group (identified by the RDC group number) is selected to be used. For packets that pass through classification successfully, with no L2 CRC error or IP checksum error, the Receive DMA Group number and the offset from the hardware classifier will be used to select the DMA channel. For packets with checksum errors, the offset will be changed to zero to select the default within the group. A RDC hardware RDC table holds the content of each RDC group. Each table consists of the following entries:

TABLE 8-1 RDC Table

Table Entry	RDC Number
0	RDC n
1	RDC n
2	RDC n
3	RDC n
4	RDC n
5	RDC n
6	RDC n
7	RDC n
8	RDC n
9	RDC n
10	RDC n
11	RDC n
12	RDC n
13	RDC n
14	RDC n
15	RDC n

Where n is any number between 0 and 15.

In the default configuration, each Ethernet port is associated with a default RDC table and all classification results will be based on the value of this RDC table. The RDC used for receive is determined by the RDC table entry that is indexed by the offset value generated by the classifier.

The following tables show the contents of the default RDC table for each reference configuration:

TABLE 8-2 Default RDC Table Content for NIU 1-Port x 10-Gb Configuration

RDC Table #0 at port0	
Table Entry	RDC Number
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	0
9	1
10	2
11	3
12	4
13	5
14	6
15	7

In this configuration, the RDC table entry 0 is *bound* to port0 as the default RDC table entry. All classification results will end up in one of the table entries in this table. The target RDC used to carry traffic will be in a range from RDC#0 to RDC#7.

TABLE 8-3 Default RDC Table Content for NIU 2-Port x 10-Gb Configuration

RDC Table #0 at port0		RDC Table#8 at port1	
Table Entry	RDC Number	Table Entry	RDC Number
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

TABLE 8-3 Default RDC Table Content for NIU 2-Port x 10-Gb Configuration (*Continued*)

RDC Table #0 at port0		RDC Table#8 at port1	
Table Entry	RDC Number	Table Entry	RDC Number
5	5	5	5
6	6	6	6
7	7	7	7
8	0	8	0
9	1	9	1
10	2	10	2
11	3	11	3
12	4	12	4
13	5	13	5
14	6	14	6
15	7	15	7

In this configuration, entry 0 is *bound* to port0 as the default RDC table entry for port0. Entry 8 is *bound* to port1 as the default RDC table entry. All classification results will end up in one of the table entries in these two table. The target RDC used to carry traffic will be in a range from RDC#0 to RDC#7.

TABLE 8-4 4-Port x 1-G Default Configuration

RDC Table#0 at port0		RDC Table#1 at port1		RDC Table#2 at port2		RDC Table#3 at port3	
0	0	0	1	0	2	0	3
1	0	1	1	1	2	1	3
2	0	2	1	2	2	2	3
3	0	3	1	3	2	3	3
4	0	4	1	4	2	4	3
5	0	5	1	5	2	5	3
6	0	6	1	6	2	6	3
7	0	7	1	7	2	7	3
8	0	8	1	8	2	8	3
9	0	9	1	9	2	9	3
10	0	10	1	10	2	10	3

TABLE 8-4 4-Port x 1-G Default Configuration

RDC Table#0 at port0		RDCTable#1 at port1		RDC Table#2 at port2		RDC Table#3 at port3	
0	0	0	1	0	2	0	3
11	0	11	1	11	2	11	3
12	0	12	1	12	2	12	3
13	0	13	1	13	2	13	3
14	0	14	1	14	2	14	3
15	0	15		15	2	15	3

In this configuration, entry 0 is *binded* to port0 as the default RDC table entry for port0. Entry 1 is *binded* to port1 as the default RDC table entry, and so on, up to 4 ports. All classification results end up in one of the table entries in these four table. Only one RDC is used for each port.

The following I/O control functions can be used to override the default RDC configuration:

- ETH_IOC_SET_RDC_GRP
- ETH_IOC_BIND_RDC_GRP

The following I/O control functions show the current RDC group contents and configuration:

- ETH_IOC_SHOW_RDC_GRP
- ETH_IOC_SHOW_RDC_GRP

Hashing Based on Layer 2, Layer 3, and Layer 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 Layer 2, Layer 3, and Layer 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
- Layer 4 source and destination port number.
or a combination of these fields.

Hash Algorithm

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 Sun 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.

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

Hash Key

The hash key is generated by a *seed* value. The following driver parameter can be used to modify the hash key:

`nxge_fflp_h1`

It is set to `0xffffffff` by default.

Application

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.

Hash Policy

Hashing is enabled by default. The hash policy is determined by setting the `FLOW_POLICY` to one of the values shown in [TABLE 8-5](#):

TABLE 8-5 Hash Policy Values

Value	Meaning
<code>HASH_IP_ADDR</code>	Hash on IP destination and source addresses
<code>HASH_IP_DA</code>	Hash on IP destination address
<code>HASH_IP_SA</code>	Hash on IP source address
<code>HASH_VLAN_ID</code>	Hash on VLAN ID
<code>HASH_PORTNUM</code>	Hash on physical MAC port number
<code>HASH_L2DA</code>	Hash on L2 destination address
<code>HASH_PROTO</code>	Hash on protocol number
<code>HASH_SRC_PORT</code>	Hash on L4 source port number
<code>HASH_DST_PORT</code>	Hash on L4 destination port number
<code>HASH_ALL</code>	Hash on all of the above fields
<code>TCAM_CLASSIFY</code>	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 Layer 2, Layer 3, and Layer 4 Header Classification

Layer 2 (L2) Classification

The layer 2 parser (part of the classification hardware) parses the following information from an Ethernet frame:

1. If the frame is VLAN packet, the VLAN ID
2. Ethernet format, whether there is a LLC/SNAP field.

Upon receiving this information, the classifier selects a RDC table to be used for further classification. L2 classification can be based on the following criteria:

- Classification based on VLAN

For VLAN frames, the VLAN ID is used to index into a VLAN table to determine the RDC table number to be used for further classification. The VLAN table consists of 4-K entries. Each entry specifies a VLAN ID and its corresponding target RDC table number.

- Classification based on MAC addresses

The target RDC table can also be determined based on the MAC address information. This information includes the MAC address type (for example, unicast, multicast, self address, address filter, or flow control) and the address.

The following I/O Control functions are used for L2 classification setup:

- `ETH_IOC_SET_MAC_TBL`
- `ETH_IOC_SET_VLAN_TBL`
- `ETH_IOC_SHOW_MAC_TBL`

Because both VLAN table and MAC address table can set the preference, the arbitration between VLAN table and MAC address table is done by setting the priority field in each of these two tables.

Note – In Sun multithreaded 10-Gb Ethernet technology, L2 classification can be seen as a coarse classification mechanism in which the output of the classification is the RDC table number. Further fine classification (such as L3/L4 classification) needs to be performed to obtain the target RDC number for the RDC to be used to carry the receive traffic.

Layer 3 and Layer 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 Sun 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

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_CLASSIFY` "IO Control" command of the Sun 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 8-6](#) shows possible opcode values:

TABLE 8-6 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 8-7](#) shows possible action values:

TABLE 8-7 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 the IPv4 and IPv6 flow. The following shows the flow_spec structure:

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

typedef struct flow_spec_ipv6_s {
    uint8_t protocol;
    uint8_t tos;
    union {
        port_t tcp;
        port_t udp;
        spi_port_t spi;
    } port;
    struct in6_addr src;
    struct in6_addr dst;
} flow_spec_ipv6_t;
```

fs_type

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

TABLE 8-8 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

TABLE 8-8 fs_type Possible Values

Value	Meaning
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 track 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_tab_s

The following is the IPv4 flow specification structure:

```
typedef struct flow_spec_ip4_tab_s {
    int            index;
    uint8_t        protocol;
    uint8_t        tos;
    uint8_t        tos_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;
```

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           tos;
    uint8_t           tos_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, 0, 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, 0, 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, 0, 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, 0, 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, 0, 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, 0, 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, 0, 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, 0, 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, 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 status;
    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.tos = fe[i].tos;
        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;
        status = inet_pton(AF_INET, (char *)fe[i].src_addr,
                          (char *)&clsfy_ioc.flow_spec.ue.ip4.src);
        if (status != 1)
            goto fail;
        status = inet_pton(AF_INET, (char *)fe[i].dst_addr,
                          (char *)&clsfy_ioc.flow_spec.ue.ip4.dst);
        if (status != 1)
            return;
        clsfy_ioc.flow_spec.um.ip4.tos = ~fe[i].tos_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;
        status = inet_pton(AF_INET, (char *)fe[i].src_addr_mask,
                          (char *)&clsfy_ioc.flow_spec.um.ip4.src);
        if (status != 1)
            goto fail;
        clsfy_ioc.flow_spec.um.ip4.src =
            ~clsfy_ioc.flow_spec.um.ip4.src;
        status = inet_pton(AF_INET, (char *)fe[i].dst_addr_mask,
                          (char *)&clsfy_ioc.flow_spec.um.ip4.dst);
        if (status != 1)
            goto fail;
        clsfy_ioc.flow_spec.um.ip4.dst =
            ~clsfy_ioc.flow_spec.um.ip4.dst;
        if (fe[i].action == FLOW_ACCEPT)
            clsfy_ioc.action = IOC_FLOW_ACCEPT;
        else
            clsfy_ioc.action = IOC_FLOW_DISCARD;

        /* 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 10-Gb Ethernet with NIU hardware to enable TCAM classification with matching rules as shown in [Step 1](#) to [Step 3](#).

Auto-Configuration

This chapter describes the Sun Netra DPS Auto-Configuration (`autoconfig`) tool. Topics include:

- [“Auto-Configuration Introduction” on page 143](#)
- [“Installation” on page 144](#)
- [“Prerequisites” on page 144](#)
- [“User Interface” on page 145](#)

Auto-Configuration Introduction

Auto-configuration is a tool for automatically configuring the Logical Domains Environment for Sun Netra DPS applications. Use the `autoconfig` tool for the following:

- Configure primary and guest domains for reference applications
- Custom configure primary and guest domains for your own applications
- Configure Logical domain channel (LDC) and interprocess communication (IPC) channels

The `autoconfig` tool cannot be used for the following:

1. Modify parameters of existing guest domains.
2. Reconfigure primary domain in configuration modes other than factory default.

Installation

The auto-configuration tool is packaged with both `SUNWndps` and `SUNWndpsd` packages.

To invoke the `autoconfig` tool, install the `SUNWndps` package on the primary domain and run the following command:

```
# /opt/SUNWndps/tools/bin/solaris/autoconfig
```

Alternatively, you can copy the auto-configuration tool from the above location from a machine where you have installed the `SUNWndps` package to the primary domain.

The user can also copy the auto-configuration tool to the primary domain from a machine where you installed the `SUNWndpsd` package. In this case, you can find auto-configuration under the `/opt/SUNWndpsd/bin/autoconfig` directory.

Prerequisites

Before running the tool, make sure the following prerequisites are satisfied:

- The system supports Oracle VM Server for SPARC (logical domains 1.0.1, or higher) and has the required firmware.
- The system has Logical Domains Manager Version 1.0.1, or higher, software installed.
- Storage for virtual disks are identified.

For more information, refer to the logical domains documentation for logical domains firmware and logical domains manager software installation.

User Interface

This section describes user interface configurations.

Configuring a Logical Domain Environment for Reference Applications

When the auto-configuration tool is invoked on a system where Logical Domains are yet to be configured, the following screen is displayed:

```
*** Netra Data Plane Suite Configurator ***

The Netra Data Plane Suite (Netra DPS) Configurator can be used to
configure the Logical Domains Environment for Netra DPS applications.
Check for the following prerequisites before proceeding:

* Your system supports Logical Domains and has the required firmware
* Your system has Logical Domains Manager installed
* Storage for virtual disks have been identified

Do you want to proceed? [y] y

Your system is ready to configure the Logical Domains Environment.
You can either choose a Logical Domain Environment that has been
predefined for various Netra DPS applications or create your own.

1) Choose a predefined Logical Domain Environment from a list
2) Custom configure a Logical Domain Environment
3) Quit

Option [1]: 1
```

To configure a Logical Domain Environment, that is, the primary and the guest domains, for a reference application, select option 1. Once you select the option, the auto-configuration tool will list the set of reference applications for which it can create the primary and guest domains automatically.

Select the application for which you want the tool to create the primary and guest domains.

After the auto-configuration tool completes the configuration, you can use the same tool to configure LDC and IPC channels for the application according to your requirements.

Note that the auto-configuration tool requires that the `vntsd` service be in enabled state for its successful operation. While configuring logical domains for reference applications using the auto-configuration tool, the `vntsd` service might take a long time to come up during the guest domains configuration step causing the tool to exit with the following error:

```
INFO: vntsd service has not come online. Please start Autoconfig after vntsd
service comes online
```

In such a scenario, execute the following command to proceed with guest domain configuration after the `vntsd` service is in enabled state:

If configuring the DemoApplication:

```
# /var/NetraDPS/autoconfig/autoconfig_work/CONFIG/main.ksh
guestdomain DemoApplication
```

If configuring the ThreeDomainsExampleApplication:

```
# /var/NetraDPS/autoconfig/autoconfig_work/CONFIG/main.ksh
guestdomain ThreeDomainsExampleApplication
```

Custom Configuring a Primary Domain

When the auto-configuration tool is invoked on a system where Logical Domains are yet to be configured, the following screen is displayed:

```
*** Netra Data Plane Suite Configurator ***
```

```
The Netra Data Plane Suite (Netra DPS) Configurator can be used to
configure the Logical Domains Environment for Netra DPS applications.
Check for the following prerequisites before proceeding:
```

- * Your system supports Logical Domains and has the required firmware
- * Your system has Logical Domains Manager installed
- * Storage for virtual disks have been identified

```
Do you want to proceed? [y] y
```

```
Your system is ready to configure the Logical Domains Environment.
You can either choose a Logical Domain Environment that has been
predefined for various Netra DPS applications or create your own.
```

- 1) Choose a predefined Logical Domain Environment from a list
- 2) Custom configure a Logical Domain Environment
- 3) Quit

```
Option [1]: 2
```

To custom configure a Primary domain, select option 2. After you select the option, the auto-configuration tool will ask a series of questions regarding your primary domain configuration such as memory, VCPU, MAU, disk, network and console services, and so on. Answer the prompts according to your requirements.

After the auto-configuration tool collects all the requirements, the tool configures the primary domain and saves the configuration on the system controller as `ndps-config-initial`.

Note – For the changes to be effective, the system should undergo a power cycle. Allow the tool to do the power cycle or you can do it manually.

After the system is up after the power cycle, you can configure the guest domains using the auto-configuration tool.

Custom Configuring a Guest Domain

When the auto-configuration tool is invoked on a system where Logical Domains are already configured, the following screen is displayed:

```
*** Netra Data Plane Suite Configurator ***

The Netra Data Plane Suite (Netra DPS) Configurator can be used to
configure the Logical Domains Environment for Netra DPS applications.
Check for the following prerequisites before proceeding:

* Your system supports Logical Domains and has the required firmware
* Your system has Logical Domains Manager installed
* Storage for virtual disks have been identified

Do you want to proceed? [y] y

A primary domain has been configured and is in active state.

Please select from one of the following options:

1) Custom configure guest domain
2) Configure guest domains under a configuration directory
3) Save guest domains configuration under a directory
4) Configure LDC channels
5) Configure IPC channels
6) Quit

Option [1]: 1
```

To custom configure a guest domain, select option 1. After you select the option, the auto-configuration tool will ask a series of questions regarding your guest domain configuration such as memory, VCPU, MAU disk, network, and so on. Answer the prompts according to your requirements.

After the auto-configuration tool collects all the information, it configures the guest domain. If the configuration succeeds, you can use the same tool to setup the `tftp` boot server for the domain. To set up `tftp`, you need to provide a private IP address for the guest domain, and if not already present, a private IP address for the primary. As an option, you can also move an image to the `/tftpboot` directory.

To configure the primary domain again at a later time using the auto-configuration tool, perform the following steps:

1. Set the logical domain configuration to factory-default mode, and perform a power cycle.
2. Start the auto-configuration tool.

Configuring LDC and IPC

The auto-configuration tool can be used to configure LDC and IPC communication channels. To configure LDC or IPC, you should have three domains in active state. One domain for the Sun Netra DPS Runtime Environment application, one domain for the Control application, and the last for the Global configuration domain. For more information, refer to [Chapter 5](#).

After you have created the required domains and they are in active state, invoke auto-configuration and select option 4 for LDC or option 5 for IPC from the following Sun Netra Data Plane Suite Configurator display.

Note – The user can only configure IPC *after* configuring the LDC.

```
*** Netra Data Plane Suite Configurator ***
```

```
The Netra Data Plane Suite (Netra DPS) Configurator can be used to
configure the Logical Domains Environment for Netra DPS applications.
Check for the following prerequisites before proceeding:
```

- * Your system supports Logical Domains and has the required firmware
- * Your system has Logical Domains Manager installed
- * Storage for virtual disks have been identified

```
Do you want to proceed? [y] y
```

```
A primary domain has been configured and is in active state.
```

```
Please select from one of the following options:
```

- 1) Custom configure guest domain
- 2) Configure guest domains under a configuration directory
- 3) Save guest domains configuration under a directory
- 4) Configure LDC channels
- 5) Configure IPC channels
- 6) Quit

```
Option [1]: 4 (or) 5
```

Before configuring IPC, you need to have installed the required Oracle Solaris OS on the Control and Global configuration domains. You should also have installed the SUNWndpsd package on those domains and should have rebooted the domain system after the package was installed. While configuring IPC, make sure the Sun Netra DPS Runtime Environment domain has the required Sun Netra DPS application running.

Saving Current Guest Domains Configuration

Save the current guest domains configuration as XML files in a directory. To do this, select option 3 from the following Sun Netra Data Plane Suite Configurator display:

```
*** Netra Data Plane Suite Configurator ***

The Netra Data Plane Suite (Netra DPS) Configurator can be used to
configure the Logical Domains Environment for Netra DPS applications.
Check for the following prerequisites before proceeding:

* Your system supports Logical Domains and has the required firmware
* Your system has Logical Domains Manager installed
* Storage for virtual disks have been identified

Do you want to proceed? [y] y

A primary domain has been configured and is in active state.

Please select from one of the following options:

1) Custom configure guest domain
2) Configure guest domains under a configuration directory
3) Save guest domains configuration under a directory
4) Configure LDC channels
5) Configure IPC channels
6) Quit

Option [1]: 3
```

After you select the option, the auto-configuration tool will save the Logical Domain configurations as XML files in the directory that you provide.

Configuring the Oracle VM Server for SPARC Software from a Saved Location

Create guest domains from their respective XML files present in a directory. To do this, select option 2 from the following Sun Netra Data Plane Suite Configurator display:

```
*** Netra Data Plane Suite Configurator ***

The Netra Data Plane Suite (Netra DPS) Configurator can be used to
configure the Logical Domains Environment for Netra DPS applications.
Check for the following prerequisites before proceeding:

* Your system supports Logical Domains and has the required firmware
* Your system has Logical Domains Manager installed
* Storage for virtual disks have been identified

Do you want to proceed? [y] y

A primary domain has been configured and is in active state.

Please select from one of the following options:

1) Custom configure guest domain
2) Configure guest domains under a configuration directory
3) Save guest domains configuration under a directory
4) Configure LDC channels
5) Configure IPC channels
6) Quit

Option [1]: 2
```

After you select the option, the auto-configuration tool will create guest domains from the directory you provide.

Transparent Interprocess Communication

This chapter describes using the Transparent Interprocess Communication (TIPC) protocol. Topics include:

- “Transparent Interprocess Communication Introduction” on page 153
- “TIPC Components” on page 154
- “Installing TIPC” on page 155
- “Programming Interfaces Overview” on page 157
- “Configuring Environment for TIPC” on page 158

Transparent Interprocess Communication Introduction

The Transparent Interprocess Communication (TIPC) protocol is designed for use in clustered computer environments. TIPC allows designers to create applications that can communicate quickly and reliably with other applications regardless of their location within the cluster.

You first must read the TIPC documentation, which is available at:
<http://tipc.sourceforge.net/>

TIPC is available for Oracle Solaris and can be downloaded at:
<http://opensolaris.org/os/project/tipc/>

TIPC Components

The TIPC implementation for Sun Netra DPS contains the components listed below. These components are required to run a TIPC application on Sun Netra DPS:

- Sun Netra DPS runtime components – These components provide all APIs and the libraries for writing the TIPC application for Sun Netra DPS:
 - `/opt/SUNWndps/lib/tipc/include` – Contains the TIPC header files.
 - `/opt/SUNWndps/lib/tipc/lwrtetipc.o` – Provides the binaries for TIPC. The application writer needs to link the application with this binary.
- Control plane (Oracle Solaris OS) components – To configure TIPC, the following is required:
 - `/opt/SUNWndpsd/bin/tn-tipc-config` – The utility tool for configuring TIPC.
 - `/opt/SUNWndpsd/lib/libtipccfgsocket.so.1` – Contains Socket API implementation over IPC. This library is required for running the `tn-tipc-config` utility tool.
- Oracle Solaris TIPC packages – Oracle Solaris TIPC is modified to support IPC bearer media:
 - `/opt/SUNWndps-tipc` – Contains TIPC with IPC media support.
 - `/opt/SUNWndps-tipc-headers` – TIPC header files to develop TIPC applications in the Solaris OS.
 - `/opt/SUNWndps-tipc-examples` – Contains TIPC example applications.
- Control plane (Linux) components - To configure Sun Netra DPS TIPC, the following is required:
 - `/opt/SUNWndpsd/linux/src/lwrte-tipc-cfg-lnx` – Contains the sources to build the utility tool called `tn-tipc-config` for Linux OS.

Note – The following packages should *not* be installed with the above mentioned packages: `SUNWtipc`, `SUNWtipc-examples` and `SUNWtipc-headers`.

- TIPC source files
 - `/opt/SUNWndps/src/libs/tipc` – Contains the TIPC source files for Sun Netra DPS.

Installing TIPC

This section describes how to install TIPC.

▼ To Install TIPC

1. Go to the `/opt/SUNWndps/src/libs/tipc` directory. Type:

```
# cd /opt/SUNWndps/src/libs/tipc
```

Or, copy `/opt/SUNWndps/src/libs/tipc` to a preferred location and change the directory to the copied location.

2. Compile the Oracle Solaris TIPC application.

Note – Compiling of Oracle Solaris TIPC packages is not necessary unless you want to change the Oracle Solaris source.

Run the following command to build the Oracle Solaris TIPC packages:

```
# gmake SOLARIS_PKG_DIR=./bins TARGET_DIR=./bins release
```

The above command builds the Sun Netra DPS binaries and Oracle Solaris packages under the `./bins` directory.

Three packages are created for Oracle Solaris TIPC, as shown below:

- `SUNWndps-tipc` – Contains the TIPC kernel module and associated library that includes support for IPC bearer.
- `SUNWndps-tipc-headers` – Contains the TIPC public header files useful for development.
- `SUNWndps-tipc-examples` – Contains ready-to-run examples.

The TIPC binaries for Sun Netra DPS are created as shown below:

- `./bins/include` – This directory contains the TIPC header files required for writing Sun Netra DPS TIPC application
- `lwrtetipc.o` – This is the TIPC library for the Sun Netra DPS application.

3. Compile the Sun Netra DPS TIPC application.

Note – Compiling of Sun Netra DPS TIPC is required if you want to change the IPC channel number used for the Sun Netra DPS TIPC configuration.

Compiling of Sun Netra DPS TIPC alone can be done by the following command.

```
# gmake TARGET_DIR=./bins
```

To build the Sun Netra DPS TIPC application with the TIPC library, the application should be compiled with the following options:

- /opt/SUNWndps/lib/tipc/include – This directory path should be added to include the path to pick the header files.
- /opt/SUNWndps/lib/tipc/lwrtetipc.o – This is added to USR_LIBS for linking to your application with the TIPC library.

4. Install TIPC.

Do the following to install Oracle Solaris TIPC.

a. Log on to your target machine as root.

b. Install the `SUNWndps-tipc` package using the `pkgadd` command as shown in the example below:

```
# pkgadd -d path-to-packages SUNWndps-tipc
```

c. If necessary, install the `SUNWndps-tipc-examples` package using the `pkgadd` command as shown below:

```
# pkgadd -d path-to-packages SUNWndps-tipc-examples
```

d. Reboot if necessary.

In certain conditions, the `tipc` module will not load automatically. In this case, reboot the system or load the `tipc` module manually by using the following command:

```
# add_drv tipc
```

- e. You must set up the environment to preload the TIPC socket library as shown below before running the Oracle Solaris TIPC applications:

```
# LD_PRELOAD_32=/opt/SUNWndps-tipc/lib/libtipcsocket.so
# LD_PRELOAD_64=/opt/SUNWndps-tipc/lib/sparcv9/libtipcsocket.so
# export LD_PRELOAD_32 LD_PRELOAD_64
```

See “[SUNWndpsd and SUNWndps-tipc Binaries](#)” on page 158 for a description of the `tipc-config` in the `SUNWndpsd` and `SUNWndps-tipc` binaries.

Programming Interfaces Overview

You can use the Transparent Interprocess Communication (TIPC) Socket APIs to write TIPC applications. The *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual* contains a detailed description of the TIPC APIs.

In a Sun Netra DPS application, you must call the framework initialization routines for the IPC, LDC, and TIPC frameworks, and must ensure that `tipc_process()` is called periodically.

The application developer must implement `tipc_pbuf_alloc()` and `tipc_pbuf_free()` to allocate and free message block buffers used by the TIPC stack. The buffers must be 8-byte aligned. The application developer must also implement `tipc_eth_pbuf_alloc()` and `tipc_eth_pbuf_free()` to allocate and free message block buffers to manage outgoing and incoming Ethernet or `vnet` packets.

To support Ethernet or `vnet` bearer in a Sun Netra DPS application, the programmer must implement the `tipc_eth_get_mac()` function to provide the MAC address of the Ethernet or `vnet` port to the TIPC stack. And also, the application must open the corresponding Ethernet or `vnet` port using `eth_open()`. To get the TIPC messages to be transmitted over the Ethernet or `vnet` media, the application needs to poll on the `fastq` returned by `tipc_eth_get_fastq(dev_type, port, chan, TIPC_FASTQ_TX)`. Then, on receiving a TIPC message over the Ethernet or `vnet`, the application needs to enqueue it into the `fastq`, returned by `tipc_eth_get_fastq(dev_type, port, chan, TIPC_FASTQ_RX)` for the corresponding Ethernet or `vnet` port.

Note – `poll()` is not supported in Sun Netra DPS.

Configuring Environment for TIPC

This section describes the configuration of the environment needed to use the TIPC framework. This section also covers setup of the Ethernet bearer for the NDPS application

SUNWndpsd and SUNWndps-tipc Binaries

The `tn-tipc-config-bin` tool in the `SUNWndpsd` package configures Sun Netra DPS TIPC node from Oracle Solaris guest logical domain. This is the same binary as `tipc-config` that is present in the `SUNWndps-tipc` package. Only the libraries that are pre-loaded are different for Oracle Solaris TIPC and Sun Netra DPS TIPC. The `tipc-config` binary is placed in two packages because there are cases where both packages will not be installed on the same system. For example, if only the `SUNWndpsd` package is installed and in the control domain, the `tn-tipc-config-bin` and the corresponding `tn-tipc-config` script are required to configure the Sun Netra DPS TIPC. Thus, both scripts are in the `SUNndpsd` package.

Also, the packaging of `SUNWndps-tipc` is the same as the Oracle Solaris TIPC that can be found at <http://tipc.sourceforge.net/>. Therefore, `SUNWndps-tipc` also has the `tipc-config` binary.

The `/opt/SUNWndpsd/bin/tn-tipc-config` script sets the environment to pre-load `/opt/SUNWndpsd/lib/libtipccfgsocket.so.1`. The script then loads the `/opt/SUNWndpsd/bin/tn-tipc-config-bin` binary (this binary is the same as `/opt/SUNWndps-tipc/sbin/tipc-config`).

When a Linux guest logical domain is used, the `tn-tipc-config-bin` and the script file `tn-tipc-config` that are present in `SUNWndpsd` cannot be used. The tool `tn-tipc-config-bin` uses IPC which is not supported for the Linux environment. For the Linux environment, the source files for building the same tool is provided in `/opt/SUNWndpsd/linux/src/lwrte-tipc-cfg-lnx`. To build this tool, please do the following:

```
# cd /opt/SUNWndpsd/linux/src/lwrte-tipc-cfg-lnx
# tar -cvf lwrte-tipc-cfg-lnx.tar lwrite-tipc-cfg-lnx/
```

In a system that has a UltraSPARC T2 cross-compiler installed:

```
# tar -xvf lwrte-tipc-cfg-lnx.tar
# cd lwrte-tipc-cfg-lnx
# make
```

This creates a binary called `tn-tipc-config`. Deploy this binary in the Linux guest logical domain.

Configuring Sun Netra DPS TIPC Stack from an Oracle Solaris Guest Logical Domain

You can configure the TIPC stack using the `tn-tipc-config` tool located at `/opt/SUNWndpsd/bin` in the Solaris Control-Plane. This tool is the same as `tipc-config` which is distributed with the Oracle Solaris TIPC package. (Refer to the TIPC documentation available at <http://tipc.sourceforge.net/> for the options supported by the `tipc-config` tool). The `tn-tipc-config` tool uses the IPC channel to configure the TIPC stack of Sun Netra DPS. `tn-tipc-config` uses `TIPC_IPC_CHANNEL_ID` channel, which is set to 10 by default in `tipc_ipc_cfgsrv.h`. This value can be modified by compiling the TIPC stack with a different `TIPC_IPC_CHANNEL_ID` and replacing `/opt/SUNWndpsd/lib/libtipccfgsocket.so.1`.

Configuring Sun Netra DPS TIPC Stack from a Linux Guest Logical Domain

The Sun Netra DPS TIPC stack can be configured from a Linux logical domain using the `tn-tipc-config` tool built from the sources in `/opt/SUNWndpsd/linux/src/lwrte-tipc-cfg-lnx`. This tool supports the same options as the `tipc-config` tool that is distributed with the Linux TIPC package (refer to the TIPC documentation available at <http://tipc.sourceforge.net/> for information on the supported options). The `tn-tipc-config` tool uses a proprietary protocol to communicate with the Sun Netra DPS TIPC stack over a `vnet` interface. This protocol uses Ethernet encapsulated frames with an Ethernet Type of `0x3c21` to communicate between the stack and the tool running in Linux over a `vnet` interface. The `vnet` interface name that is to be used for this communication must be assigned to an environment variable called `TN_TIPC_CFGDEV`.

The prerequisites for using this tool are:

1. One vnet interface must be configured for the Oracle VM Server for SPARC software and the Linux Guest logical domain. These vnet interfaces must be reachable through a vswitch.
2. Ethernet multicast frame processing must be enabled on these vnet interfaces.
3. The Sun Netra DPS application must have vnet driver packet processing.
4. The `TN_TIPC_CFGDEV` environment variable to the vnet interface name that is used for `tn-tipc-config` exchanges.

Refer to the *Sun Netra Data Plane Software 2.1 Update 1 Reference Manual* for instructions on how to handle the `tn-tipc-config` frames.

The following commands set the TIPC address for Sun Netra DPS TIPC node from a Linux control plane logical domain:

```
# export TN_TIPC_CFGDEV=eth1
# ./tn-tipc-config -addr=10.3.7
```

▼ To Set the TIPC Address

The TIPC addressing mechanism can be found at the TIPC homepage (<http://tipc.sourceforge.net/>).

- **Set the address using the `-addr` option of the `tn-tipc-config` tool.**

The following example shows setting the TIPC address to 10.3.4:

```
# /opt/SUNWndpsd/bin/tn-tipc-config -addr=10.3.4
```

Enabling TIPC Ethernet Bearer

The TIPC Ethernet bearers in the Sun Netra Data Plane is named as a port number, for example, `port0`, `port1`, and so on.

For example, to enable `eth:port0`, the corresponding Ethernet port0 must have been opened by the application.

The following example enables the Ethernet port0 bearer for the Sun Netra DPS application with netmask of 10.3.0:

```
# /opt/SUNWndpsd/bin/tn-tipc-config -be=eth:port0/10.3.0
```

For more details on the `-be` option in `tipc-config`, refer to the TIPC documentation available at <http://tipc.sourceforge.net>.

Enabling the TIPC IPC Bearer

The TIPC IPC bearers must be named as *channel.type*, for example, 5.200, 6.100, and so on. Before enabling the bearer, ensure that the corresponding IPC channel is created. See “[Interprocess Communication Software](#)” on page 89 to create an IPC channel. For example, to enable IPC bearer with IPC Channel 5 and type 200, the corresponding IPC channel 5 must be created using the `tnsmctl` tool.

Note the following requirements:

- The IPC channel number must not be the same as `TIPC_IPC_CHANNEL_ID` which is used by TIPC configuration service.
- The channel and type number should be the same in both Oracle Solaris and Sun Netra DPS for communication through the IPC bearer.

The following is an example of enabling IPC bearer for Sun Netra DPS using `tn-tipc-config`:

```
# /opt/SUNWndpsd/bin/tn-tipc-config -be=ipc:5.200/10.3.0
```

The following is an example of enabling IPC bearer for Oracle Solaris using `tipc-config`:

```
# /opt/SUNWndps-tipc/sbin/tipc-config -be=ipc:5.200/10.3.0
```

Enabling TIPC vnet Bearer for a NDPS TIPC Node

The `-be` option of the `tn-tipc-config` command enables the `vnet` bearers for a Sun Netra DPS TIPC node. The media type in the bearer name must be specified as `eth`. The interface name in the bearer name must be `vnet` followed by the instance number, for example, `vnet{instance number}`. The instance number can be obtained by executing the following command from the primary Logical Domain

```
ldm list-bindings -e control-ldom
```

For example, the following command enables a vnet bearer using a vnet device with instance number 5:

```
# tn-tipc-config -be=eth:vnet5/10.3.0
```

Reference Applications

This chapter describes Sun Netra DPS reference applications.

Topics include:

- [“IP Packet Forwarding Reference Applications”](#) on page 164
- [“Differentiated Services Reference Application”](#) on page 220
- [“Generic Routing Encapsulation Reference Application”](#) on page 236
- [“Access Control List Reference Application”](#) on page 247
- [“Radio Link Protocol Reference Application”](#) on page 252
- [“IPSec Gateway Reference Application”](#) on page 256
- [“Traffic Generator Reference Application”](#) on page 283
- [“Interprocess Communication Reference Application”](#) on page 304
- [“Transparent Interprocess Communication Reference Application”](#) on page 310
- [“vnet Reference Application”](#) on page 319

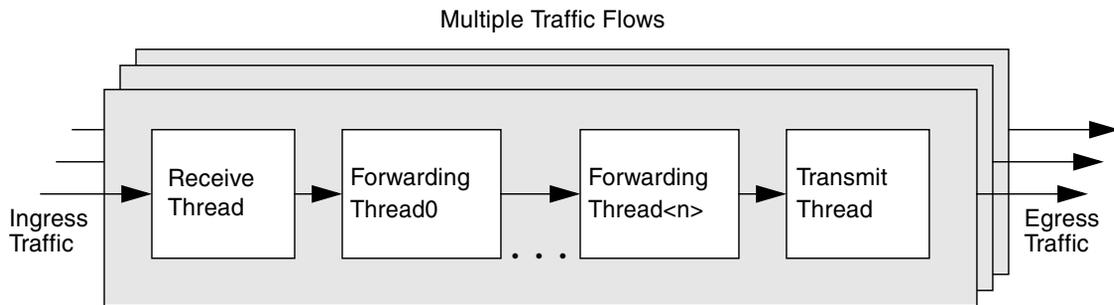
Reference applications illustrate how users applications are written to exploit full capability of Sun Netra DPS running on chip multithread architecture. Each reference application consists of extensive examples. In many cases, these examples can be leveraged as building blocks of the users deployment application.

IP Packet Forwarding Reference Applications

The IP Packet Forwarding Application (`ipfwd`) performs IPv4 (Internet Protocol Version 4) and IPv6 (Internet Protocol Version 6) forwarding operations. When packet traffic is received, the application performs forwarding table searches and determines the destination (next hop). It then re-writes the packet header of the packet to be forwarded.

The basic IP Forwarding application consists of three or more software threads forming a traffic flow with multiple traffic flow running in parallel. The following figure depicts the basic IP Forwarding structure.

FIGURE 11-1 IP Forwarding Traffic Flows



Receive Thread

The receive thread performs the following tasks:

1. Polls packets received from a particular DMA channel's HW descriptor ring.
2. Checks for received packet status.
3. Delivers the packet to the forwarding thread through fast queue.

The bulk of implementation of the receive thread resides in the device driver. Normally, no user modification is required.

Forwarding Thread

The forward thread performs the following tasks:

1. Polls packet from Rx fast queue enqueued by Rx thread.
2. Verifies the packet header.
3. Checks the received packet's integrity.
4. Encapsulates or decapsulate packet header, if necessary.

5. If the packet is destined to host, forwards the packet to the host. Otherwise, performs lookup for next hop information, based on a selected lookup algorithms, such as:
 - Direct match or hashing
 - Linear search
 - Longest prefix match (LPM)
 - Binary search of prefix length (BSPL)
6. Updates the packet header with next hop's address.
7. Delivers the packet to the Tx thread through fast queue.

You can form single or multiple threads in a pipeline depending on the workload of the forwarding tasks.

Transmit Thread

The transmit thread performs the following tasks:

1. Polls packet from IP forwarding thread through fast queue.
2. Posts the packet to the target transmit descriptor ring of the Tx DMA channel.

Similar to the receive thread, the majority of the code of the transmit thread resides in the device driver.

Traffic Flows

In this reference application, each software thread is mapped into a hardware CPU strand. The hardware classifier and the hashing mechanism spread ingress traffic into multiple parallel traffic flows, each implemented in a multiple threads pipeline described above. Multiple traffic flows can run in parallel. The overall forwarding packet rate is the aggregate packet rate of each traffic flow.

Source Files

All `ipfwd` source files are located in the following directories:

`SUNWndps/src/apps/ipfwd`

`user_workspace/SUNWndps/src/apps/ipfwd`

▼ To Compile the `ipfwd` Application

1. Copy the `ipfwd` reference application from the `SUNWndps/src/apps/ipfwd` directory to a desired directory location.
2. Execute the build script in the `ipfwd` directory.

Usage

```
./build cmt type [ldoms [diffserv] [acl] [gdb] [excp] [tipc]  
[no_freeq] [gre] [ipv6]] [profiler] [2port] [vnet] -hash  
POLICY_NAME
```

Note – *cmt* (processor type) and *type* (network interface type) must be specified in each build.

Argument Descriptions

The build script supports the following arguments:

- *cmt*

Specifies whether to build the `ipfwd` application to run on the CMT1 (UltraSPARC T1) platform or CMT2 (UltraSPARC T2) platform.

 - *cmt1* – Build for CMT1 (UltraSPARC T1)
 - *cmt2* – Build for CMT2 (UltraSPARC T2)
- *type*
 - *4g* – Build `ipfwd` application to run on 4-Gbps Ethernet QGC (quad 1-Gbps *nxge* Ethernet interface).
 - *10g* – Build `ipfwd` application to run on 10-Gbps Ethernet (dual 10-Gbps *nxge* Ethernet interface).
 - *10g_niu* – Build `ipfwd` application to run on NIU (dual 10-Gbps UltraSPARC T2 Ethernet interface) on a CMT2-based system.
- [*ldoms*]

Specifies whether to build the `ipfwd` application to run on the logical domain environment. When this flag is specified, the IP forwarding logical domain reference application will be compiled. If this argument is not specified, then the non-logical domains (standalone) application will be compiled. Note that the options under the *ldoms* parameter (such as *diffserv*, *acl*, and *gdb*) can be

enabled only when this option is specified. See [“How Do I Calculate the Base PA Address for NIU or Logical Domains to Use with the `tnsmct1` Command?”](#) on page 388.

- `[diffserv]`
Enables the differentiated services reference application.
- `[acl]`
Enables the access control list (ACL) reference application.
- `[gdb]`
Enables `gdb` support in the logical domain environment.
- `[excp]`
Enables processing of IPv4 protocol exceptions and support of address resolution protocol (ARP).
- `[tipc]`
Enables application to use TIPC to communicate with control plane application.
- `[ipv6]`
Enables IPv6 packet forwarding. Note that when this option is not specified, the application performs IPv4 forwarding.
- `[no_freeq]`
Disables the use of free queues. Can be used with the `diffserv` option in an logical domain environment.
- `[gre]`
Enables the GRE reference application.
- `[profiler]`
Generates code with profiling enabled.
- `[2port]`
Compiles dual ports on the 10-Gbps Ethernet or the UltraSPARC T2 NIU.
- `[vnet]`
Enables the usage of `vnet` interfaces for exception handling by the `ipfwd` Sun Netra DPS application.
- `[-hash POLICY_NAME]`
Enables flow policies. For more information, see [“Other IP Forwarder Options”](#) on page 170.

▼ To Build the `ipfwd` Application

- In `/src/sys/lwrte/apps/ipfwd`, **pick the correct build script, and run it.**
For example, to build for 10-Gbps Ethernet on a Sun Netra or Sun Fire T2000 system, type:

```
% ./build cmt1 10g
```

In this example, the build script with the `10g` option is used to build the IP forwarding application to run on the 10-Gbps Ethernet. The `cmt` argument is specified as `cmt1` to build the application to run on UltraSPARC T1-based Sun 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 user-workspace/ipfwd/code/ipfwd/ipfwd /tftpboot/ipfwd
```

3. At the `ok` prompt on the target machine, type:

```
ok boot network-device:,ipfwd
```

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

Default System Configuration

The following table shows the default system configuration.

TABLE 11-1 Default System Configuration

	NDPS Domain (strand IDs)	FastPath Manager (strand ID)	Other Domain (strand IDs)
CMT1 non-logical domain	0 to 31	31	N/A
CMT1 logical domain	0 to 19	19	20 to 31
CMT2 non-logical domain	0 to 63	63	N/A
CMT2 logical domain	0 to 55	55	56 to 63

The main files that control the default system configuration are:

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

Default `ipfwd` Application Configuration

The following table shows the default `ipfwd` application configuration.

TABLE 11-1 Default `ipfwd` Application Configuration

Application Runs On	Number of Ports Used	Number of Channels per Port	Total Number of Q Instances	Total Number of Strands Used
4-Gbps PCIE (<code>nxge QGC</code>)	4	1	4	12
10-Gbps PCIE (<code>nxge 10-Gbps</code>)	1	4	4	12
10-Gbps NIU (<code>niu 10-Gbps</code>)	1	8	8	24

The main files that control the `ipfwd` application configuration are:

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

Other IP Forwarder Options

Other IP forwarding application options can be enabled during the compile time by enabling them in the makefiles.

- `IPFWD_RAW`

This option is used to bypass the `ipfwd` operation (that is, receive -> transmit without forwarding operation), uncomment the following line from `Makefile.nxge` to compile for the Sun multithreaded 10-Gbps NIU, 10-Gbps PCIe Ethernet adapter, and quad 1-Gbps PCIe Ethernet adapter.

```
-DIPFWD_RAW
```

- `IPFWD_MULTI_QS`

When this 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.nxge` to compile for the Sun multithreaded 10-Gbps Ethernet, respectively:

```
-DIPFWD_MULTI_QS
```

- `N2_1_MODE`

This option is enabled by default. You must disable this flag when running Sun Netra DPS on UltraSPARC T2 version 2.2 and above for optimal performance.

```
-DN2_1_MODE
```

- `KSTAT_ON`

This option enables the device driver to collect statistical information. To enable this option, uncomment the following line from `Makefile.nxge`. Note that there is a slight performance reduction when this option is enabled:

```
-DKSTAT_ON
```

- `IPFWD_DISPLAY_STATS`

This option enables the IP forwarding application to display statistical information to the console. This option must be accompanied by the `KSTAT_ON` option. To enable this option, uncomment the following line from `Makefile.nxge`:

```
-DIPFWD_DISPLAY_STATS
```

- `FORCEONEMEMPOOL`

The default memory pool configuration of the IP forwarding application is one memory pool per traffic flow. This option overrides the default memory pool configuration. When this option is enabled, all traffic flows share one memory pool. To enable this option, uncomment the following line from `Makefile.nxge`:

```
-DFORCEONEMEMPOOL
```

- `VNET_TIPC_CONFIG`

This option enables the TIPC stack in `ipfwd` reference application to be configured using the Linux `tn-tipc-config` tool. The Linux `tn-tipc-config` tool uses `vnet` for exchanging commands/data. When the Linux `tn-tipc-config` tool is used, then the `ipfwd` reference application must be compiled with the `-DTIPC_VNET_CONFIG` flag enabled in the makefiles (for example `Makefile.nxge`):

```
-DFORCEONEMEMPOOL
```

IP Forward Static Cross Configuration

When IP Forwarding is configured as cross configuration, the `IPFWD_STATIC_CROSS_CONFIG` flag must be enabled. The following is one example of cross configuration:

```
Port0 ---> Port1  
Port1 ---> Port0
```

Flow Policy for Spreading Traffic to Multiple DMA Channels

Specify a policy for spreading traffic into multiple DMA flows by hardware hashing. [TABLE 11-2](#) describes each policy:

TABLE 11-2 Flow Policy Descriptions

Name	Definition
IP_ADDR	Hash on IP destination and source addresses.
IP_DA	Hash on IP destination address.
IP_SA	Hash on IP source address.
VLAN_ID	Hash on VLAN ID.
PORTNUM	Hash on port number.
L2DA	Hash on L2 destination address.
PROTO	Hash on Protocol number.
SRC_PORT	Hash on source port number.
DST_PORT	Hash on destination port number.
ALL	Hash on all of the above fields.
TCAM_CLASSIFY	Performs TCAM lookup.

To enable one of the above policies, use the `-hash` option.

If none of the policies listed in [TABLE 11-2](#) are specified, a default policy is given. The default policy is set to `HASH_ALL`. When you use the default policy, all L2, L3, and 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 Sun 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 can by-pass 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.

ipfwd Configuration 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 to 31 on an UltraSPARC T1 system and 0 to 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 match the physical port number displayed on the console when the boot command (with `-v` option used) is executed to perform `tftpbboot` of the binary. For example, use the `port#` if the network device you would like to use for IP forwarding shows up as the following in the console output during boot:

- `netdev[4]: Vendor ID 0x108e Dev ID 0xabcd`
- `netdev[4]: Subsystem Vendor ID 0x108e Subsystem ID 0x0`
- `netdev[4]: Revision ID 0x1`
- `netdev[4]: PhyType xgf`
- `netdev[4]: Compatible pciex108e,abcd.108e.0.1`
- `netdev[4]: cfg_addr 0x120000 pio_addr 0xc106000000`
- `netdev[4]: mac_addr 0x0:14:4f:6c:74:a8`

In this case, the port number specified in the `port#` field of the application configuration should be set to 4.

- `chan#`

If this is a multi-channel device (such as Sun multithreaded 10-Gbps Ethernet with NIU), this entry indicates the channel number within each port. Sun multithreaded 10-Gbps Ethernet device has 24 transmit channels (0 to 23) and 16 receive channels (0 to 16) in each port. Sun multithreaded 10-Gbps Ethernet with NIU has 16 channels (both `tx` and `rx`) in each port.

- `Role`

This is the role of the software task.

`TROLE_ETH_NETIF_RX` (performs a receive function)

`TROLE_ETH_NETIF_TX` (performs a transmit function)

`TROLE_APP_IPFWD` (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.

System Configuration

The IP forwarding application can be set up in two different environments: standalone and logical domain.

Standalone Environment

In the standalone environment, Sun Netra DPS gains control of the entire system. All system resources are dedicated for Data Plane usage. When the `ldoms` option is not specified in the build script, then the `ipfwd` application is built for running on the standalone environment. In the standalone environment, no forwarding information base (FIB) is specified.

All packets are forwarded based on *hard-coded* information in the program. The users must modify the program to change the default forwarding information and its corresponding forwarding path. Using the IP forwarding application build script without specifying the `ldoms` option will generate the executable for the standalone environment.

Logical Domain Environment

In a logical domain environment, Sun Netra DPS and other logical domains share the system resources. Sun Netra DPS is used as the data plane, other logical domains are used as the control plane. The `ipfwd` application must be built with the `ldoms` option for this environment. The logical domain environment has more flexibility over the standalone environment on controlling the forwarding information and specifying the forwarding path.

Forwarding Application

The forwarding application consists of two major groups of components: data plane components that run on the Sun Netra DPS runtime and the control plane components and utilities that run on the Oracle Solaris OS.

Data Plane Components

The forwarding application fast path code are reside mainly in the following subdirectories:

- 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 logical domain will have. Also, the memory pools used in the `malloc()` and `free()` implementation for the logical domain and IPC frameworks are declared here.

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

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 forwarder state machine implementation code resides in the following files and their corresponding header files:

- `ipfwd_state.c`
- `ipfwd_eth.c`
- `ipfwd_ip4.c`
- `ipfwd_ip6.c`
- `ipfwd_lib.c`

`ipfwd_config.c`, and its header file, consists of default configuration entries that determine how application threads are mapped into hardware CPU strands for the forwarding application. In the `ipfwd` application, all software thread entry points (except the fast path manager) are mapped into the `_main` entry point (see `ipfwd_map.c`). In the `_main()` function, each thread is further assigned a particular task to perform based on the information specified in the file.

`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.

`ipfwd_ipc.c` contains the IPC logical domain framework initialization functions. The initialization of the logical domain framework is accomplished using calls to the functions `mach_descrip_init()`, `lwrtex_cnex_init()`, and `lwrtex_init_ldc()`. After this initialization, the IPC framework is initialized by a call of `tnipc_init()`. The previous four functions must be called in this specific order. The data structures are then initialized for the forwarding tables.

`ipfwd_tipc.c`, and its header files, contains the TIPC logical domain functions. When you specify the `tipc` option during the build, TIPC will be used as the communication protocol between control and data plane. Otherwise, IPC will be used by default.

`ipv4_excp.c`, and its header files, consists of code that handles exceptions, such as IP fragmentation and re-assembly.

`ipfwd_flow.c`, and its header files, specifies the L3/L4 classification flow entries. When `TCAM_CLASSIFY` is used in the `-hash` option during the build, these entries will be programmed into the TCAM during initialization of the application.

The `diffserv/` directory consists of the `diffserv` implementation.

The `gre/` directory consists of the GRE tunneling implementation.

The `radix/` directory consists of the radix forwarding algorithm implementation.

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. After booting the application, the IPC channels are

initialized. After the IPC or TIPC channels are up, you can use the Oracle Solaris OS control plane utilities to set up the network interface, to manipulate the forwarding tables, and to gather statistics.

Control Plane Components and Utilities

The code for the Oracle Solaris control plane components and utilities are located in the `src/solaris` subdirectory. This file implements a simple CLI to control the forwarding application running in the Sun Netra DPS runtime (LWRTE) domain. These applications are not built when `ipfwd` is built. They must be built separately using `gmake` in the directory and deployed into a domain that has an IPC channel to the LWRTE domain established.

The code for the Linux control plane components and utilities are located in `src/linux`. The applications for Linux are not built when `ipfwd` is built. They must be built separately using the makefile in `src/linux` and deployed into a domain that is running Linux. By default, the makefile in `src/linux` uses `gcc` version 4.3.2 which is a part of Wind River Linux Sourcery G++ 4.3-85 toolchain. The compiler is a cross-compiler for UltraSPARC T2 platform that is installed on a Linux/x86-64 machine.

Interface Configuration Utility (ifctl)

The `ifctl` interface is used to configure interfaces of the Sun Netra DPS `ipfwd` application, as well as displaying the interface parameters. It is similar to the `ifconfig` utility in the Oracle Solaris OS, but the available commands and parameters provide the basic functionality only.

The following shows the usage of the `ifctl` tool:

```
ifctl iface-name port-num address tun [tunnel-address] tuntype
4in4|4in6|6in4|6in6|gre|none up|down netmask [netmask] mtu [mtu] vtag
[vid]
```

Starting the tool without any options will display the current interfaces along with their configuration.

- `-h` or `--help`
Gives a brief description of the command syntax.
- *iface-name*
Specifies the name of the interface. The first non-numeric string on the command line is interpreted as interface name, except the valid command words (up or down). The interface name can be up to 5 characters long.
- *port-num*

Specifies the Ethernet port number assigned to the interface. The port number should always starts from 0.

- *address*

Specifies the IP address to be assigned to the interface. The `ifctl` tool accepts IPv4 and IPv6 addresses in the following formats:

- IPv4 address:

D.D.D.D (where *D* is a octet in decimal format)

- IPv6 address:

H:H:H:H:H:H:H:H (where *H* is a 16 bit value in hex). `ifctl` supports the simplified forms of the IPv6 address string representations. The following formats are accepted:

H:H:H:H:H::H:H

H:H:H:H:H:H

H:H:H::H

- *tun*

Specifies the IP address of the remote end of the tunnel.

- *tuntype*

Specifies the type of the tunnel configured on the interface. The types of tunnels supported are:

- `4in4` – Indicates IP-in-IP tunnel is configured on the interface.
- `4in6` – Indicates IPv4-in-IPv6 tunnel is configured on the interface.
- `6in4` – Indicates IPv6-in-IPv4 tunnel is configured on the interface.
- `GRE` – Indicates that GRE tunnel is configured on the interface.
- `none` – Disables tunneling on an interface.

- *up*

Activate the interface. If the interface has been added previously and brought down subsequently, then the interface can be brought up without specifying the parameters again. This option must be used when adding the interface for the first time.

- *down*

Shuts down the interface. All packets received on or forwarded to this interface will be dropped.

- *mtu*

Configures the MTU of the interface. The value supplied is in bytes. It must be between 46 bytes and 1500 bytes. For interfaces that have tunneling enabled, the value represents the maximum L3 packet size, excluding the encapsulating headers, but including the payload L3 header.

- *netmask*

Configures the netmask for the IPv4 interface. The netmask supplied must be in dotted decimal format.

- *vtag*

Configures the VLAN ID (VID) of the interface. To disable VLAN tagging on an interface, provide a value of 0 for the VLAN ID using this option.

Note – On Oracle Solaris OS platforms, `ifctl` communicates with the `ipfwd` application through IPC. Therefore, `ifctl` must have read and write permission to the `tasm` device node, and the LDC channels must be configured between logical domains. The `ipfwd` application must be running to accept `ifctl` commands.

Note – On Linux platforms, `ifctl` communicates with the `ipfwd` application only using TIPC. On Linux platforms, IPC is not supported. Therefore, the `ifctl` application must be built with TIPC support in it.

ifctl Examples

This section contains examples that show how to use the `ifctl` options.

▼ To Add an IPv4 Interface

- Execute the following command:

```
% ./ifctl port0 0 1.2.3.4
```

▼ To Add an IPv6 Interface

- Execute the following command:

```
% ./ifctl port0 0 1111:2222:3333::aaaa
```

▼ To Enable IP-in-IP Tunneling on an Interface

- Execute the following command:

```
% ./ifctl port0 0 192.168.100.100 tun 192.168.100.2 tuntype 4in4
```

▼ To Disable Tunneling on an Interface

- Execute the following command:

```
% ./ifctl port0 0 192.168.100.100 tun 192.168.100.2 tuntype none
```

▼ To Add an IPv6 Interface and Bring the Interface Up

- Execute the following command:

```
% ./ifctl port1 1 1111:2222:3333::aaaa up
```

▼ To Disable Interface port0

- Execute the following command:

```
% ./ifctl port0 down
```

▼ To Set the MTU for an Interface That Does Not Have Tunneling Enabled

- Execute the following command:

```
% ./ifctl port0 0 mtu 1500
```

▼ To Set the MTU for an Interface That Has IPv4-in-IPv4 Tunneling Enabled

- Execute the following command:

```
% ./ifctl port0 0 mtu 1480
```

▼ To Set the MTU for an Interface That Has GRE Tunneling Enabled Where GRE Header Includes Checksum, Key, and Sequence Number Fields

- Execute the following command:

```
% ./ifctl port0 0 mtu 1464
```

▼ To Set the Netmask on an Interface

- Execute the following command:

```
% ./ifctl port1 1 netmask 255.255.255.0
```

▼ To Enable VLAN on an Interface With VLAN ID

- Execute the following command:

```
% ./ifctl port0 0 vtag 8
```

▼ To Disable VLAN on an Interface

- Execute the following command:

```
% ./ifctl port0 0 vtag 0
```

FIB Control Utility (fibctl)

The FIB Control utility (*fibctl*) is used to download the FIB table data from the control plane to the data plane. When *fibctl* is started in the control plane, the *fibctl>* prompt will appear. The program 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 command.

- *load file_name*

Loads an FIB table file that consists of FIB table data. The IP Forwarding Reference Application uses the following FIB table data file with the application:

```
SUNWndps/src/apps/ipfwd/src/solaris/fibctl_tables
```

- *write-table Table_ID*

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

- *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, corresponding to predefined tables. Before a table can be used, it must be transmitted using the *write-table* command described above.

- `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.

▼ To Build the `ifctl` and `fibctl` Utility

1. Execute the appropriate `gmake` command.

- a. To use the `fibctl` and `ifctl` utilities on an Oracle Solaris OS logical domain, execute the `gmake` in the Oracle Solaris OS subtree (`SUNWndps/src/apps/ipfwd/src/solaris`):

```
% gmake
```

2. Execute the appropriate `make` command.

- a. To use the `fibctl` and `ifctl` utilities on a Linux OS logical domain, copy the sources in `src/linux` and `src/common` onto a machine that has the cross-compiler installed.

For all utilities built for Linux logical domains, the `TIPC=on` option must be used.

```
% tar -cvf ipfwd-utils.tar SUNWndps/src/apps/ipfwd/src/linux  
SUNWndps/src/apps/ipfwd/src/common
```

- b. In the `linux` directory, execute the `make` command.

c. On the system that has the cross-compiler installed, perform the following:

```
% mkdir ipfwd-utilities
% cp ipfwd-utils.tar ipfwd-utilities
% cd ipfwd-utilities
% tar -xvf ipfwd-utils.tar
% cd linux
% make ifctl TIPC=on
% make fibctl TIPC=on
```

Note – To include `diffserv` and GRE functionalities, enable the GRE flag and `DIFFSERV` flag. Along with `gmake`, set `DIFFSERV` to `on` and `GRE` to `on`. In the IP forwarding reference application, `DIFFSERV` and `GRE` flags cannot be enabled simultaneously.

After the channel to be used is initialized using `tnsmctl` (must be channel ID 4 which is hard coded into the `ipfwd` application), use `fibctl` to change the behavior of `ipfwd` as shown below example:

```
fibctl> connect 4
fibctl> load fibctl_tables
fibctl> write-table 0
fibctl> write-table 1
fibctl> use-table 0
fibctl> use-table 1
fibctl> quit
```

Exception Daemon (excpd)

The `excpd` application is responsible for:

- Managing the FIB table.
- Managing the interfaces when using the 1wIP ARP layer for ARP processing.
- Interfacing with the ARP layer.
- Communicating with the data plane for exchanging FIB and interface information.

To build the `excpd` application, the application source is provided with the Sun Netra DPS `ipfwd` reference application. The application source is present in the `ipfwd/src/solaris/excpd` directory. The following build options are provided:

Usage

```
./build lwip|sol [tipc]
```

- `lwip` – Use the lwIP ARP layer.
- `sol` – Use the Solaris ARP layer.

Note – The `excpd` application is not used when `ipfwd` reference application is used with Linux guest logical domain.

IPv4 Packet Forwarding Application with Exception Handling

The IPv4 packet forwarder with exception handling consists of:

- Address resolution protocol (ARP)
- IPv4 protocol exception handling (fragmentation and reassembly)
- FIB table management

ARP (RFC 826) is a protocol that enables dynamic mapping of IPv4 addresses to Ethernet addresses. It is used with the IPv4 forwarding application to map the next-hop IPv4 addresses in the FIB table to their Ethernet addresses.

The IPv4 exception handling enables fragmentation of egress packets and reassembly of fragmented packets that are destined to the local host.

FIB table management enables the updates of the next-hop IP addresses in the Data Plane FIB table, with their Ethernet addresses. When new Ethernet addresses are learnt, the FIB entries are updated by the FIB management layer and passed to the Data Plane application. When the exception handling is handled in control plane host using `vnet` for packet transfers, the FIB entries are updated by the learning module within the data plane application itself.

Exception handling is enabled only when the `ipfwd` application is built with the `ldoms` and `excp` options (see “[IP Packet Forwarding Reference Applications](#)” on [page 164](#) for an explanation of these build options).

The `ipfwd` reference application is extended with a framework that allows handling of ARP and IPv4 protocol exceptions. [FIGURE 11-2](#) depicts the exception handling framework in the `ipfwd` application that use either `LwIP` or `Solaris Host (TIPC/TNIPC)` methods. [FIGURE 11-3](#) depicts the exception handling frame framework that uses `Oracle Solaris` or `Linux Host` with `vnet` for packet transfers.

ARP Processing

Three methods of ARP processing are provided in the `ipfwd` reference application when Oracle Solaris OS is used in the control plane logical domain. One method uses the `lwIP` ARP protocol layer to process ARP packets and to maintain the ARP cache. Another method uses the Oracle Solaris ARP layer to process ARP packets and to maintain the ARP cache, but uses either `TNIPC` or `TIPC` for packet transfers with the Oracle Solaris OS logical domain. A third method uses the Oracle Solaris ARP layer to process ARP packets and to maintain the ARP cache, but uses `vnet` interfaces for packet transfers with the Oracle Solaris OS logical domain.

When Linux OS is used in the control plane logical domain, only one method of ARP processing is provided. The Linux ARP layer is used to process ARP packets and to maintain the ARP cache. The `vnet` interfaces are used for packet transfers with the Linux OS logical domain.

ARP in `lwIP`

When the `lwIP` ARP layer is used for ARP processing, the ARP layer is a part of the `excpd` application. `lwIP` is a static library that implements the TCP/IP protocol stack. The `excpd` application uses the ARP layer of `lwIP` to process the ARP packets and for ARP table maintenance.

ARP in the Oracle Solaris OS

In this method, the ARP layer in the Oracle Solaris OS control plane is used for ARP processing. The ARP cache is also managed in the Oracle Solaris OS. The `excpd` application is responsible only for FIB management. A STREAMS module named `lwmodarp` is used in the Oracle Solaris OS to interface with the Oracle Solaris ARP layer. For each interface enabled in the data plane, a corresponding `vnet` interface is configured in the Oracle Solaris domain. The `lwmodarp` module is inserted into the ARP-device STREAM of each configured `vnet` interface. This module communicates with the data plane application to receive and transmit ARP packets over IPC/TIPC.

ARP in the Oracle Solaris OS or Linux OS Using `vnet`

In this method, the ARP layer in the Oracle Solaris OS or Linux OS is used for ARP processing. The ARP cache is also managed in the Oracle Solaris or Linux OS. The differences from the previous method are:

1. This method does not use `TNIPC` or `TIPC` for packet transfers with the control plane OS

2. This method does not use `excpd`, `lwip`, or `lwmodarp` modules

The FIB management is done in the `ipfwd` Sun Netra DPS application. The FIB table is pushed to the data plane using `fibctl` tool. The `ipfwd` application in Sun Netra DPS will learn the MAC addresses from ARP packets received from external hosts and from ARP packets that are transmitted from the control plane to external hosts. The learnt MAC addresses are used to update the FIB table that is currently in use.

Note – Currently, when ARP packets are handled using `vnet` interfaces for communication with the control plane, the learning mechanism in the data plane learns MAC addresses only for those IP addresses that are present in the `dest-addr` column of the FIB table file (that is, the learning mechanism learns MAC addresses only for the gateways in the FIB table). Thus, the user must push a FIB table to the data plane before exception packets and control plane packets can be handled using this method. In addition, if the user requires that the learning mechanism learns MAC addresses of any host, even if the host is not a gateway, then the learning mechanism must be extended with this functionality.

IPv4 Protocol Exception Handling

IPv4 protocol exception handling involves fragmentation, reassembly, and local delivery. This section contains descriptions of these handling processes.

Fragmentation

When a packet that must be forwarded needs to be fragmented, the IPv4 forwarder thread passes the packet to the fastpath manager thread. The fastpath manager thread calls the IPv4 fragmentation routine that fragments the packet. The fragments are then sent to the transmit threads of the outgoing interface.

Reassembly and Local Delivery

When a packet is received in the data plane, the data plane IPv4 layer determines if the packet is destined to one of the configured local interfaces. If true, then the packet is passed to the fastpath manager that sends the packet to the IPv4 layer of the Oracle Solaris control domain. If such packets are fragments, then the Oracle Solaris IPv4 layer handles the reassembly. A STREAMS module named `lwmodip4` is used in the Oracle Solaris OS to interface with the Oracle Solaris IPv4 layer. For each interface enabled in the data plane, a corresponding `vnet` interface is configured in

the Oracle Solaris domain. The `lwmodip4` module is inserted into the ARP-IP-device STREAM of each configured `vnet` interface. This module communicates with the data plane application to receive and transmit IPv4 packets over IPC/TIPC.

Reassembly and Local Delivery Using `vnet`

When a packet is received in the data plane, the data plane IPv4 layer determines if the packet is destined to one of the configured local interfaces. If true, then the packet is passed to the fastpath manager that sends the packet to the IPv4 layer of the Oracle Solaris OS or Linux control domain using one of the `vnet` interfaces in Sun Netra DPS that is connected to a `vnet` interface in the Oracle Solaris OS or Linux OS logical domain. If such packets are fragments, then the Oracle Solaris OS or Linux IPv4 layer does the reassembly of the fragments. Note that when `vnet` is used to transfer IPv4 protocol exception packets, `lwmodip4` is not used in the Oracle Solaris OS and Linux OS logical domain.

FIB Management

FIB management is performed by the `excpd` application. The `excpd` application receives FIB tables from the `fibctl` utility. When a FIB table is received, the `excpd` application performs ARP cache lookup for the next-hop IP addresses in the FIB. It fills the MAC addresses in the FIB entries and transfers the completed FIB entries to the data plane. For FIB entries whose MAC addresses are not found in the ARP cache, it monitors the ARP cache until the MAC addresses are found.

FIGURE 11-2 Internal Block Diagram for the `ipfwd` Reference Application Using IwIP or Oracle Solaris OS Host With TIPC and TNIPC

FIGURE 11-3 Internal Block Diagram for the `ipfwd` Reference Application Using Oracle Solaris OS or Linux Host With `vnet`

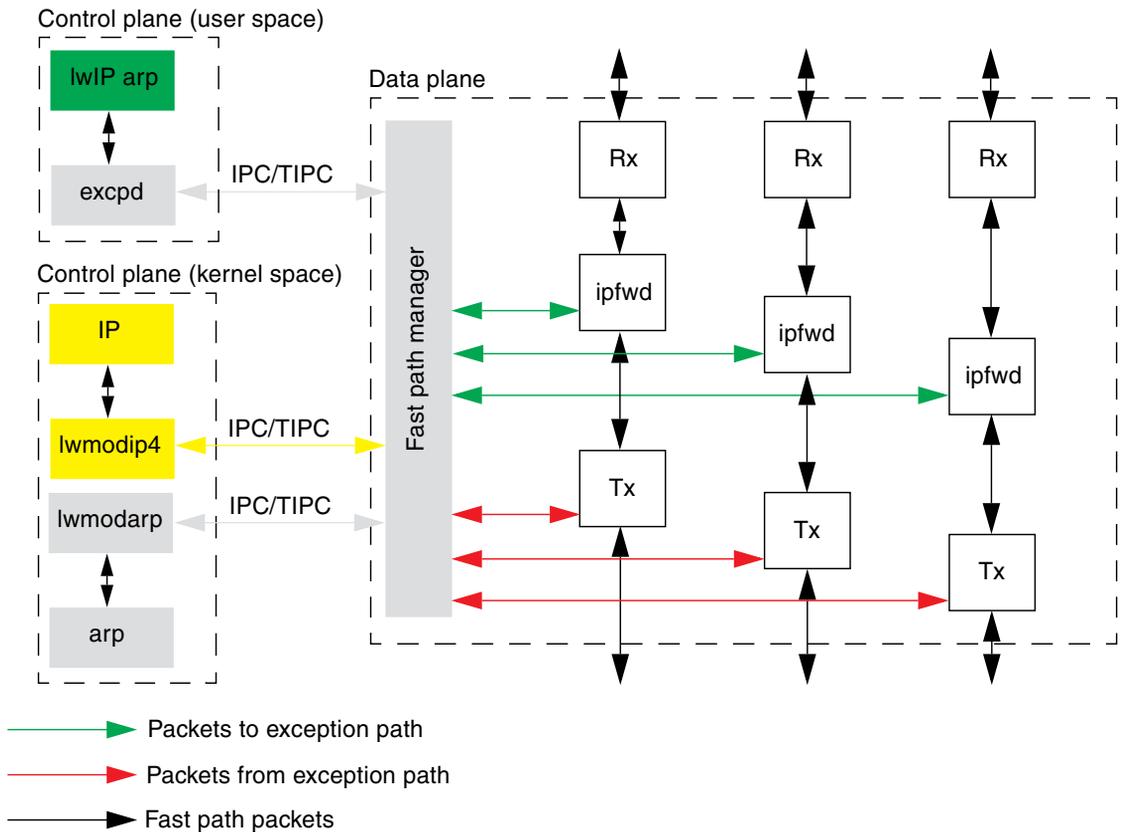


FIGURE 11-2 depicts the exception handling framework in the `ipfwd` reference application that use either IwIP or Oracle Solaris OS Host (TIPC and TNIPC) methods. The boxes in gray and the arrows in green and red illustrate the exception path framework.

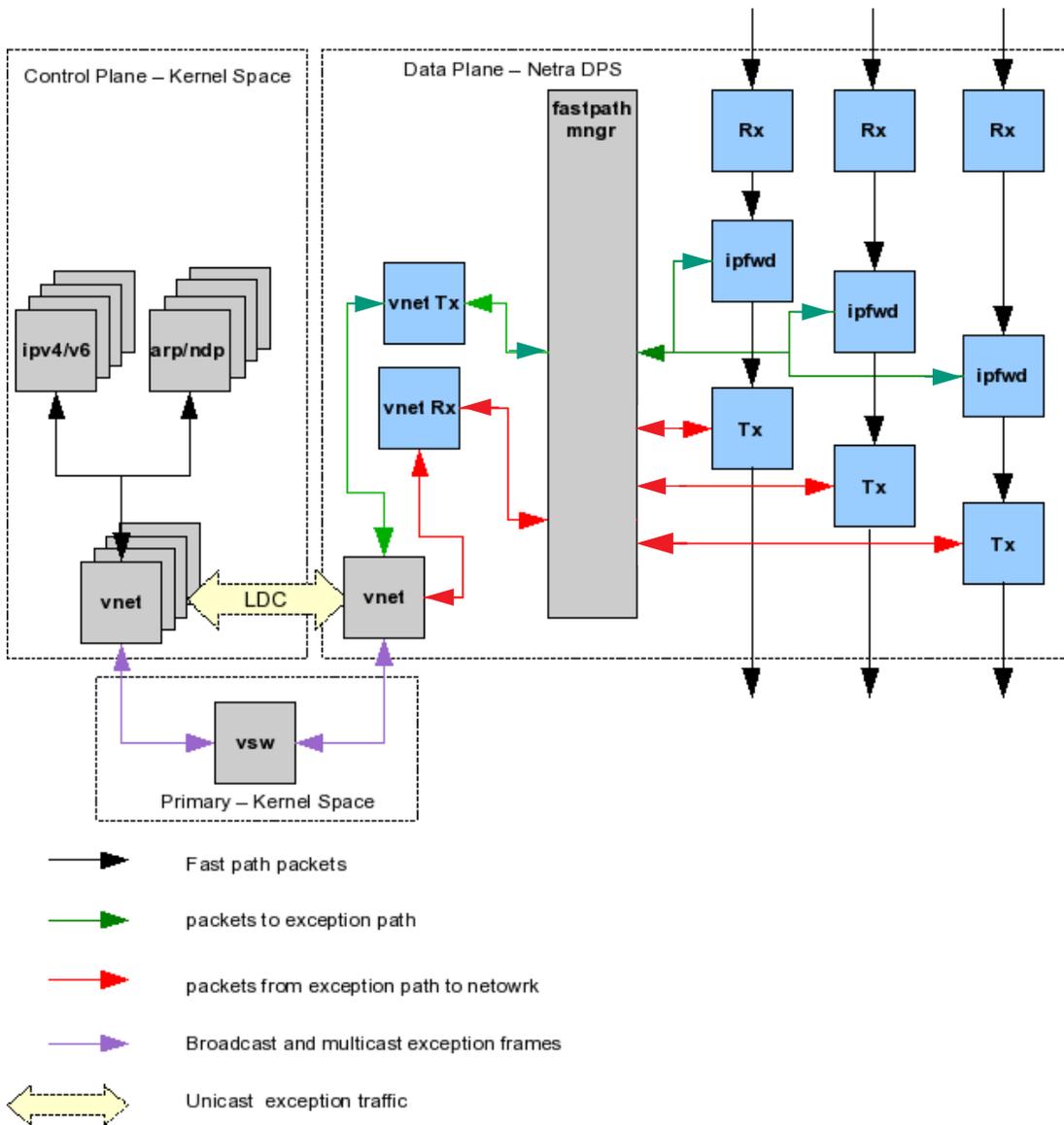


FIGURE 11-3 depicts the exception handling framework in the `ipfwd` reference application that use either Oracle Solaris OS host or Linux host using `vnet`. The boxes in gray and the arrows in green and red illustrate the exception path framework.

FIB Management When Using `vnet`

When exception handling is done in the control plane Oracle Solaris OS or Linux OS using `vnet` for packet transfers, FIB management is done in the data plane application itself. The FIB is pushed by the user using the `fibctl` tool. When ARP packets are received by the data plane application, either from external hosts (on fast path Ethernet interfaces) or from the control plane (on `vnet` interfaces), the data plane learns MAC addresses of the hosts. The learnt addresses are used to update the MAC addresses of the FIB table entries.

Exception Path Framework Components

The exception path framework consists of the following components:

- IPv4 forwarder
- `excpd` application
- lwIP ARP layer
- `lwmodarp`
- `lwmodip4`
- Fastpath manager
- `vnet` driver

IPv4 Forwarder (`ipfwd` Thread)

The IPv4 forwarder receives Ethernet frames from the Rx strand. The forwarder checks if the frames received contain IPv4 packets. All frames that do not contain IPv4 packets are passed to the fastpath manager (green arrows).

All frames that contain IPv4 packets are further processed by the IPv4 forwarder thread. While processing the IPv4 packets, if any IPv4 protocol exception is detected, the IPv4 forwarder thread passes those packets to the fastpath manager thread for processing the exception (green arrows).

The following IPv4 protocol exceptions will result in an exception condition:

- The TTL in the packet expired while forwarding.
- The packet is destined to a network or host that does not have an entry in the FIB table.
- The packet must be forwarded to a host or gateway which has an Ethernet address that is unresolved.
- The length of the packet is larger than the MTU of the outgoing interface and must be fragmented.

- The packet is destined to an interface owned by the `ipfwd` application (local delivery)

Exception Application (`excpd`)

The `excpd` application is a user-space Oracle Solaris OS application that is responsible for:

- Managing the FIB table.
- Managing the network interfaces when using `lwIP` ARP layer for ARP processing.
- Interfacing with the ARP layer.
- Communicating with the data plane for exchanging FIB and interface information.

Note – When ARP is processed in the Oracle Solaris OS or Linux OS using `vnet` for ARP packet transfer, the `excpd` exception application must not be used.

`lwIP` ARP Layer

`lwIP` is a static library that implements the TCP/IP protocol stack. This is used when ARP processing is done in `excpd` application. To use the `lwIP` ARP layer, the `excpd` application is built with the `lwip` option (see [“To Build the `excpd` Application When `lwIP` ARP Is Used With IPC” on page 196](#)).

ARP STREAMS Module (`lwmodarp`)

This is used when ARP processing is done in the control domain Oracle Solaris ARP layer. This module is used to pass ARP packets between the Oracle Solaris ARP layer and the data plane `ipfwd` application. It uses IPC or TIPC to communicate with the data plane application.

Note – When ARP is processed in the Oracle Solaris OS, the `lwIP` ARP layer is not used in the `excpd` application. The `excpd` application must be compiled with the `sol` option (see [“To Build the `excpd` Application When `lwIP` ARP Is Used With IPC” on page 196](#)).

Note – When the `lwIP` ARP layer is used, the `lwmodarp` module must not be used.

Note – When ARP is processed in the Oracle Solaris OS or Linux OS using `vnet` for ARP packet transfer, `lwmodarp` must not be used.

The IPv4 STREAMS Module (`lwmodip4`)

This module is used for the processing of IPv4 packets that are destined to the local interfaces. The module passes IPv4 packets to and from the control plane Oracle Solaris IPv4 layer and the data plane `ipfwd` application. It uses IPC or TIPC to communicate with the data plane application.

Note – This module must not be used when IPv4 exception handling is done in the Oracle Solaris OS or Linux OS using `vnet` for packet transfer.

Fastpath Manager

The fastpath manager performs the following functions related to IPv4 exception handling and ARP processing:

- Interfaces with the control plane components like `excpd`, `lwmodip4`, `lwmodarp`, `fibctl`, `ifctl` using IPC, or TIPC.
- Passes egress packets from control plane to transmit strands.
- Receives packets from IP forwarder strands and sends them to control plane.
- Receives packets that need to be forwarded, but need to be fragmented, from IP forwarder strands, performs fragmentation, and sends the fragments to transmit strands.
- Interfaces with the `vnet` transmit and receive strands to transmit and receive packets over the `vnet` interfaces.
- Executes the MAC Address learning algorithm and the FIB management when exception handling is done using `vnet` for communication.

Exceptions Path Framework Tools

The following tools are required to use the `ipfwd` application with exception handling and ARP handling.

ifctl

See “Control Plane Components and Utilities” on page 178.

fibctl

See “Control Plane Components and Utilities” on page 178.

insarp

The `insarp` tool is used to insert the `lwmodarp` STREAMS module into the ARP-dev stream of an IPv4 interface. By default, the tool expects a module named `lwmodarp`.

```
# ./insarp
```

The tool provides the following options:

- `add`

Inserts the `lwmodarp` module into the ARP-dev stream of the IPv4 interface. The module is inserted between the device driver and the ARP STREAMS module. The following shows the usage:

```
insarp interface-name add
```

```
# ./insarp vnet2 add
```

- `rem`

Removes the `lwmodarp` module after ARP module in ARP-dev STREAM of the IPv4 interface. The following shows the usage:

```
insarp interface-name rem
```

```
# ./insarp vnet2 rem
```

- `list`

Lists the modules present in ARP-IP-dev STREAM and the ARP-dev stream of an IPv4 interface. The following shows the usage:

```
insarp interface-name list
```

```
# ./insarp vnet2 list
ARP-IP-dev STREAM Mod List: 4
0 arp
1 ip
2 lwmodip4
3 vnet

ARP-dev STREAM Mod List: 3
0 arp
1 lwmodarp
2 vnet
```

▼ To Compile the `ipfwd` Application for IPv4 Exception Handling

- **Copy the `ipfwd` reference application from `/opt/SUNWndps/src/apps/ipfwd` directory to a desired directory location, and execute the build script in that location.**

▼ To Compile the IPv4 Forwarding Application With Exception Handling By Using Sun Netra DPS

1. **On a system that has `/opt/SUNWndps` installed, go to the `user_workspace/src/apps/ipfwd` application directory.**
2. **Build the application using the build script.**
The `ldoms` and the `excp` options must be provided.

```
% /build cmt2 10g_niu ldoms excp
```

Compiling the `excpd` Application

The `excpd` application source is provided along with the Sun Netra DPS `ipfwd` reference application in the `ipfwd/src/solaris/excpd` directory. The application is built using the `build` file in this directory.

Usage

```
build lwip|sol [tipc]
```

The following build options are provided:

- `lwip` – Use the lwIP ARP layer.
- `sol` – Use the Oracle Solaris OS ARP layer.
- `tipc` – Use TIPC to communicate with data plane. Otherwise, use TNIPC.

▼ To Build the `excpd` Application When lwIP ARP Is Used With IPC

- Execute the following command:

```
% ./build lwip
```

▼ To Build the `excpd` Application When lwIP ARP Is Used With TIPC

- Execute the following command:

```
% ./build lwip tipc
```

▼ To Build the `excpd` Application When the Oracle Solaris OS ARP Is Used With IPC

- Execute the following command:

```
% ./build sol
```

▼ To Build the `excpd` Application When the Oracle Solaris OS ARP Is Used With TIPC

- Execute the following command:

```
% ./build sol tipc
```

Compiling the lwmodip4 STREAMS Module

The `lwmodip4` module is provided in the `ipfwd/src/solaris/module` directory. The module is built using the `build` file in this directory.

Usage

```
build ipv4|ipv6 [tipc]
```

The following build options are provided:

- `ipv4` – Build `lwmod` for IPv4 interface.
- `ipv6` – Build `lwmod` for IPv6 interface.
- `tipc` – Use TIPC to communicate with data plane. Otherwise, use TNIPC.

▼ To Build the lwmodip4 STREAMS Module for IPv4 Exception Handling Using IPC

- Execute the following command:

```
% ./build ipv4
```

▼ To Build the lwmodip4 Module for IPv4 Exception Handling Using TIPC

- Execute the following command:

```
% ./build ipv4 tipc
```

Compiling the lwmodarp STREAMS Module

The `lwmodarp` module is provided in the `ipfwd/src/solaris/excpd/module` directory. The module is built using the `build` file in this directory.

Usage

```
build tipc|ipc
```

The following build options are provided:

- `tipc` – Use TIPC to communicate with data plane.
- `ipc` – Use TNIPC to communicate with data plane.

▼ To Build the `lwmodarp` Module for Oracle Solaris ARP Handling Using IPC

- Execute the following command:

```
% ./build ipc
```

▼ To Build the `lwmodarp` Module for Oracle Solaris ARP Handling Using TIPC

- Execute the following command:

```
% ./build tipc
```

Compiling the `insarp` Tool

The `insarp` tool source is provided in the Sun Netra DPS `ipfwd` reference application. The source is provided in the `ipfwd/src/solaris/excpd/tools` directory.

▼ To Compile the `insarp` Tool

- Execute the following command:

```
% gmake
```

▼ To Run the `ipfwd` Application with IPv4 Exception Handling in `lwIP`

1. Set up logical domains on the target system with one Sun Netra DPS domain and the following Oracle Solaris OS domains:
 - `primary` – primary domain for running logical domain manager (`ldm`)
 - `ndps` – Sun Netra DPS domain for running the Sun Netra DPS `ipfwd` application

- 1dg2 – Oracle Solaris OS domain for running the `excpd` application
- 1dg3 – Oracle Solaris domain for establishing IPC channels

One `vnet` interface is needed in the 1dg2 for each data plane port. These `vnet` interfaces are connected to isolated `vswitches` of the primary. Add `vswitches` for each `vnet` that will be configured.

```
# ldm add-vswitch vsw1 primary
# ldm add-vswitch vsw2 primary
```

2. Reboot the primary domain for these changes to take effect.

3. Add the `vnet` interfaces to the control domain 1dg2.

The MAC addresses must be the same as that of the Sun Netra DPS domain interfaces.

```
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX vnet1 vsw1 1dg2
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX vnet2 vsw2 1dg2
```

4. Run the `ipfwd` application that compiled with exception handling:

a. Place the `ipfwd` binary in the `tftpboot` server:

```
% cp user-dir/ipfwd/code/ipfwd/ipfwd tftpserver-boot/tftpboot
```

b. At the `ok` prompt on the target machine, type:

```
ok boot network-device:,ipfwd
```

5. Place the IPv4 STREAMS module in 1dg2, and load it:

```
# modload lwmodip4
```

6. Enable the `vnet` interface for each data plane port in 1dg2, and insert `lwmodip4` for each interface:

```
# ifconfig vnet1 plumb
# ifconfig vnet1 modinsert lwmodip4@2
# ifconfig vnet1 12.12.12.13 netmask 255.255.255.0 up
# ifconfig vnet2 plumb
# ifconfig vnet2 modinsert lwmodip4@2
# ifconfig vnet2 11.11.11.12 netmask 255.255.255.0 up
```

7. Place the `excpd` application, the `fibctl` application, the `ifctl` application in the `ldg2` domain, and execute the `excpd` application:

```
% ./excpd log &
```

8. Configure the Sun Netra DPS network interface with the `ifctl` application:

```
% ./ifctl port0 0 12.12.12.13 netmask 255.255.255.0 mtu 1500 up
% ./ifctl port1 0 12.12.12.12 netmask 255.255.255.0 mtu 1500 up
```

9. Configure the FIB tables using the `fibctl` application:

```
% ./fibctl fibctl_tables
```

▼ To Run the `ipfwd` Application with IPv4 Exception Handling and ARP Handling in the Oracle Solaris Host

1. Set up logical domains on the target system with one Sun Netra DPS domain and the following Oracle Solaris domains:
 - `primary` – Primary domain for running logical domain manager (`ldm`)
 - `ndps` – Sun Netra DPS domain for running the Sun Netra DPS `ipfwd` application
 - `ldg2` – Oracle Solaris domain for running the `excpd` application
 - `ldg3` – Oracle Solaris domain for establishing IPC channels

One `vnet` interface is needed in `ldg2` for each data plane port. These `vnet` interfaces are connected to isolated `vswitches` of the primary domain. Add `vswitches` for each `vnet` interface that will be configured

```
# ldm add-vswitch vsw1 primary
# ldm add-vswitch vsw2 primary
```

2. Reboot the primary domain for these changes to take effect.

3. Add the `vnet` interfaces to the control domain `ldg2`.

The MAC addresses must be the same as that of Sun Netra DPS domain interfaces.

```
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet1 vsw1 ldg2
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet2 vsw2 ldg2
```

4. Run the `ipfwd` application that compiled with exception handling.

a. Place the `ipfwd` binary in the `tftpboot` server:

```
% cp user-dir/ipfwd/code/ipfwd/ipfwd tftpserver-boot/tftpboot
```

b. At the `ok` prompt on the target machine, type:

```
ok boot network-device:,ipfwd
```

5. Place the IPv4 STREAMS module and the ARP STREAMS module in `ldg2`, and load it:

```
# modload lwmodip4
# modload lwmodarp
```

6. Place the `insarp` tool in the Oracle Solaris control domain.

7. Configure one `vnet` interface for each data plane port, and insert `lwmodip4` and `lwmodarp` for each interface.

```
# ifconfig vnet1 plumb
# ifconfig vnet1 modinsert lwmodip4@2
# ./insarp vnet1 add
# ifconfig vnet1 12.12.12.13 netmask 255.255.255.0 up
# ifconfig vnet2 plumb
# ifconfig vnet2 modinsert lwmodip4@2
# ./insarp vnet2 add
# ifconfig vnet2 11.11.11.12 netmask 255.255.255.0 up
```

8. Place the `excpd` application, the `fibctl` application, the `ifctl` application in the `ldg2` domain, and execute the `excpd` application:

```
% ./excpd log &
```

The `excpd` application can be passed a log file name for logging all errors and warnings as shown above. The log file name can also be omitted. If omitted, all errors and warnings will be printed to the screen.

Note – The `excpd` application must run as a background process.

9. Configure the Sun Netra DPS network interface with the `ifctl` application:

```
% ./ifctl port0 0 12.12.12.13 netmask 255.255.255.0 mtu 1500 up
% ./ifctl port1 0 12.12.12.12 netmask 255.255.255.0 mtu 1500 up
% ./ifctl vnet2 2 0.0.0.0 netmask 255.255.255.0 mtu 1500 up
```

10. Configure the FIB tables using the `fibctl` application:

```
% ./fibctl fibctl_tables
```

Note – The `excpd` application must be started before interfaces are configured using `ifctl` and FIB tables are downloaded using `fibctl`.

▼ To Compile the `ipfwd` Application with IPv4 Exception Handling using `vnet` in Sun Netra DPS

1. On a system with `/opt/SUNWndps` installed, go to the `user_workspace/src/apps/ipfwd` application directory.
2. Build the application using the build script.
The `ldoms`, `excp`, and `vnet` options must be provided.

```
% ./build cmt2 10g_niu ldoms excp vnet
```

▼ To Run the `ipfwd` Application with IPv4 Exception Handling and ARP Handling in an Oracle Solaris OS Host Using `vnet`

1. Set up the logical domains on the target system with one Sun Netra DPS domain and the following Oracle Solaris OS domains:
 - `primary` – Primary domain for running logical domain manager (`ldm`)
 - `ndps` – Sun Netra DPS domain for running the Sun Netra DPS `ipfwd` application
 - `ldg2` – Oracle Solaris OS domain for handling exceptions
 - `ldg3` – Oracle Solaris OS domain for establishing IPC channels

One `vnet` interface is needed in `ldg2` for each data plane port. One `vnet` interface is needed in `ndps` each ethernet port in the data plane. One `vswitch` is needed in the `primary` domain for each data plane port. Add the `vswitch` devices in the `primary` domain for the `vnet` devices in `ldg2` and `ndps` that will be used for exception handling.

```
# ldm add-vswitch vsw1 primary  
# ldm add-vswitch vsw2 primary
```

2. Reboot the primary domain for these changes to take effect.

3. Add the `vnet` interfaces to the control domain `ldg2`.

The MAC address must be the same as the interfaces in the Sun Netra DPS domain (`ndps`):

```
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet1 vsw1 ldg2
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet2 vsw2 ldg2
```

4. Add the `vnet` interface that is used for exception handling in `ndps`.

```
# ldm add-vnet vnet1 vsw1 ndps
# ldm add-vnet vnet2 vsw2 ndps
```

5. Run the `ipfwd` application that is compiled with exception handling:

a. Place the `ipfwd` binary in the `tftpboot` server:

```
% cp user-dir/ipfwd/code/ipfwd/ipfwd tftpserver-boot/tftpboot
```

b. At the `ok` prompt on the target machine, type:

```
ok boot network-device:,ipfwd
```

6. Configure one `vnet` interface for each data plane port in `ldg2`:

```
# ifconfig vnet1 plumb
# ifconfig vnet1 12.12.12.13 netmask 255.255.255.0 up
# ifconfig vnet2 plumb
# ifconfig vnet2 11.11.11.12 netmask 255.255.255.0 up
```

7. Place the `ifctl` application and the `fibctl` application in the `ldg2` domain.

8. Configure the Sun Netra DPS network interfaces with the `ifctl` application:

```
# ./ifctl port0 0 12.12.12.13 netmask 255.255.255.0 mtu 1500 up
# ./ifctl port1 1 11.11.11.12 netmask 255.255.255.0 mtu 1500 up
```

9. Configure the FIB tables using the `fibctl` application:

```
# ./fibctl fibctl_tables
```

From this moment, the MAC address learning module will start learning MAC address for the next-hops mentioned in the FIB table. The data plane will start transferring packets to and from the control plane using the `vnet` interface in `ndps`.

▼ To Compile the IPv4 Forwarding Application With Exception Handling Using `vnet` in Sun Netra DPS

This procedure is used for the Linux guest logical domain.

1. **On a system that has `/opt/SUNWndps` installed, go to the `user_workspace/src/apps/ipfwd` application directory.**
2. **Enable the `-DVNET_TIPC_CONFIG` flag in the required makefile.**
For example: `Makefile.nxge`
3. **Build the application using the build script.**
The `ldoms`, `excp`, `tipc`, and `vnet` options must be provided:

```
# ./build cmt2 10g_niu ldoms excp tipc vnet
```

▼ To Run the `ipfwd` Application with IPv4 Exception Handling and ARP Handling in the Linux Host Using `vnet`

1. **Set up the logical domains on the target system with one Sun Netra DPS domain and the following guest domains:**
 - `primary` – Primary domain for running logical domain manager (`ldm`)
 - `ndps` – One `vnet` interface is needed in each Sun Netra DPS domain for each Ethernet port in the data plane
 - `ldg2` – Linux domain for handling exceptions
 - `ldg3` – Oracle Solaris OS domain for executing the `tnsmctl -P -v` command

One `vnet` interface is needed in `ldg2` for each data plane port. One `vnet` interface is needed in `ndps` for each Ethernet port in the data plane. One `vswitch` is needed in the `primary` domain for each data plane port. Add the `vswitch` devices in the `primary` domain for the `vnet` devices in `ldg2` and `ndps` that will be used for exception handling.

```
# ldm add-vswitch vsw1 primary
# ldm add-vswitch vsw2 primary
```

2. **Reboot the primary domain for these changes to take effect.**

3. Add the `vnet` interfaces to the control domain `ldg2`.

The MAC address must be the same as the interfaces in the Sun Netra DPS domain.

```
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet1 vsw1 ldg2
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet2 vsw2 ldg2
```

4. Add the `vnet` interface that is used for exception handling in `ndps`:

```
# ldm add-vnet vnet1 vsw1 ndps
# ldm add-vnet vnet2 vsw2 ndps
```

5. Run the `ipfwd` application that is compiled with exception handling:

a. Place the `ipfwd` binary in the `tftpboot` server:

```
# cp user-dir/ipfwd/code/ipfwd/ipfwd tftpserver-boot/tftpboot
```

b. At the `ok` prompt on the target machine, type:

```
ok boot network-device:,ipfwd
```

6. Configure one `vnet` interface for each data plane port in `ldg2`.

```
# ifconfig vnet1 12.12.12.13 netmask 255.255.255.0 up
# ifconfig vnet2 11.11.11.12 netmask 255.255.255.0 up
```

7. Configure the Sun Netra DPS TIPC node and Linux TIPC node.

Note that the `tn-tipc-config` tool for Linux must be built from the `SUNWndpsd` package. See [“To Configure the Environment for TIPC”](#) on page 315 for instructions on how to build this tool.

```
# ./tn-tipc-config -addr=10.3.5
# ./tn-tipc-config -be=eth:vnet1/10.3.0
# tipc-config -addr=10.3.4
# tipc-config -be=eth:eth1/10.3.0
```

8. Place the `fibctl` application and the `ifctl` application in the `ldg2` domain.

9. Configure the Sun Netra DPS network interfaces with the `ifctl` application:

```
# ./ifctl port0 0 12.12.12.13 netmask 255.255.255.0 mtu 1500 up
# ./ifctl port1 1 11.11.11.12 netmask 255.255.255.0 mtu 1500 up
```

10. Configure the exception handling vnet interface in ndps.

The name for this interface must be in the form `vnetinstance-number`. Obtain the instance number by executing the `ldm list-bindings -e ndps` command in the primary domain. The number listed under the `DEVICE` column in the output of this command is the instance number. Also, a valid IP address must not be assigned to the vnet interface that is used for exception handling. This device is operated as a pure L2 device.

```
# ./ifctl vnet1 1 0.0.0.0 netmask 255.255.255.0 mtu 1500 up
# ./ifctl vnet2 2 0.0.0.0 netmask 255.255.255.0 mtu 1500 up
```

11. Configure the FIB tables using the fibctl application:

```
# ./fibctl fibctl_tables
```

From this moment, the MAC address learning module will start learning MAC address for the next-hops mentioned in the FIB table. The data plane will start transferring packets to and from the control plane using the vnet interface in ndps.

IPv6 Packet Forwarding Application with Exception Handling

The IPv6 packet forwarder with exception handling consists of:

- Interface management

Interface management is used to set up network interfaces and change their parameters such as address. Based on the interface data the incoming packets are either handed over to the host (control plane) or passed to the protocol exception handling block.

- IPv6 protocol exception handling

The exception handling looks for IPv6 packets that require extra actions and passes them to the control plane for further processing. Such packets are neighbor or router solicitation and advertisement messages.

- FIB management

The rest of the packets that do not need special treatment are passed to the forwarding block that uses the data provided by FIB management to decide where to send the packet or whether encapsulation is needed.

- IP-IP tunneling

IP-IP tunneling takes care of decapsulating the incoming packets or encapsulating the outgoing packets if necessary.

- Data-plane and control-plane synchronization

Data-plane and control-plane synchronization is responsible of keeping the interface and FIB data of the data plane synchronized with the interface, routing, and neighbor data of the control plane.

Interface Management

Interface management is performed by the `ifct1` application in the control plane. It can add and remove interfaces, change the address, physical port, and possible tunnel point. The interface data is transferred to the data plane through IPC or TIPC.

When a packet is received in the data plane, the data plane IPv6 layer determines if the packet is destined to one of the configured local interfaces. If true, then the packet is passed to the fastpath manager that sends the packet to the IPv6 layer of the Oracle Solaris control domain. If the destination interface is a tunnel endpoint then the packet is decapsulated.

When IPC or TIPC is used for exception packet transfers with the control domain, a STREAMS module named `lwmodip6` is used in the Oracle Solaris OS to interface with the Oracle Solaris IPv6 Layer. For each interface enabled in the data plane, a corresponding `vnet` interface is configured in the Oracle Solaris domain. The `lwmodip6` module is inserted into the STREAMS stack of each configured `vnet` interface. This module communicates with the data plane application to receive and transmit IPv6 packets over IPC or TIPC.

When the `vnet` interface is used for exception packet transfers with the control domain, the STREAMS module, `lwmodip6` is not used. Instead, the exception path packets are directly transmitted and received using the `vnet` interfaces.

IPv6 Protocol Exception Handling

Packets not destined to a local interface are checked for possible exceptions. Exceptional packets such as neighbor or router solicitation or advertisement messages are passed to the control plane, using the packet passing mechanism described in the [“Interface Management” on page 207](#).

The control plane uses the network stack of the Oracle Solaris OS to conduct neighbor or router discovery, address configuration, and duplicate address detection. The resulting routing entries and neighbor cache entries are combined into FIB entries and propagated to the data-plane. See [“Data-Plane and Control-Plane Synchronization” on page 209](#) for further details.

Note – Exception handling does not currently include fragmenting of the forwarded packets.

IPv6 Protocol Exception Handling Using `vnet`

Packets not destined to a local interface are checked for possible exceptions. Exceptional packets such as neighbor or router solicitation or advertisement messages are passed to the control plane using the `vnet` interfaces.

Note – Currently, when Neighbor Discovery Protocol packets are handled using `vnet` interfaces for communication with the control plane, the learning mechanism in the data plane learns MAC addresses only for those IP addresses that are present in the `dest-addr` column of the FIB table (that is, the learning mechanism learns MAC addresses only for the gateways in the FIB table). Thus, the user must push a FIB table to the data plane before exception packets and control plane packets can be handled using this method. In addition, if the user requires that learning mechanism learns MAC addresses of any host, even if the host is not a gateway, then the learning mechanism must be extended with this functionality.

The control plane uses the network stack of the Oracle Solaris OS or Linux OS to conduct neighbor or router discovery, address configuration and duplicate address detection. The user pushes a FIB to the data plane. The MAC address learning module in the data plane will learn the MAC address of the next-hop hosts in the FIB using the neighbor or router solicitation or advertisement messages.

Note – Exception handling does not currently include fragmenting of the forwarded packets.

FIB Management

FIB management is performed by the `ipfwd_sync.d` application running in the control plane. The application uses the `fibctl.sh` utility to add, remove, or change FIB entries in the local copy of the database. After the changes are done in the local copy it is transferred to the data-plane using the `fibctl` tool. FIB entries are changed when a new route is added or an existing route is removed in the control plane. FIB entries are also modified when changes in the control plane's neighbor cache require changes.

FIB Management Using `vnet` Exception Handling

The FIB Management is done within the data plane application by the MAC address learning module. The user pushes a FIB to the data plane. The MAC address learning module will update the FIB entries with MAC addresses learnt from neighbor solicitation, neighbor advertisement, router solicitation, router advertisement and router redirect messages that are received from data ports or from the `vnet` interfaces.

Note – When exception handling is done using `vnet`, the `ipfwd_sync.d` is not used.

IP-IP Tunneling

IP-IP tunneling is controlled through the `ifctl` tool. It can set up four types of tunnels:

- `6in6` (IPv6-in-IPv6)
- `6in4` (IPv6-in-IPv4)
- `4in6` (IPv4-in-IPv6)
- `4in4` (IPv4-in-IPv4)

The tunnels are created when an interface is given a second IP address that becomes the tunnel endpoint. Packets received over tunnels are decapsulated and processed as usual. If the forwarding results in the packet being sent over a tunnel than it is encapsulated in the appropriate IP protocol and transmitted.

Data-Plane and Control-Plane Synchronization

The `ipfwd_sync.d` application monitors the control plane (Oracle Solaris OS) for the following events:

- Interface changes (add, remove, up, down, and address change)
- Routing entry changes (add and remove)
- Neighbor cache changes (set address and remove)

Interface changes are propagated to the data plane using the `ifctl` tool.

Routing entry changes are applied to the local copy of the data plane FIB table using `fibctl.sh`. `fibctl.sh` can add, remove, and change FIB entries in the local copy and then load the FIB table to the data plane.

Neighbor cache changes are also applied to the local FIB table copy first. When a neighbor appears, the FIB table is searched for gateways (next hop nodes) with the same IP address as the new neighbor. The MAC address of these entries are updated. When the neighbor disappears the gateway MAC addresses are set to 00:00:00:00:00:00.

Exception Path Components

The exception path framework consists of the following components:

- IPv6 forwarder
- `lwmodip6`
- Fastpath manager
- `vnet` driver

IPv6 Forwarder (`ipfwd` Strand)

The IPv6 forwarder receives Ethernet frames from the Rx strand. The forwarder checks if the frames received contain IP (IPv6 or IPv4) packets. Frames that do not contain IP packets are passed to the fastpath manager.

All frames that contain IPv6 packets are further processed by the IPv6 forwarder thread. While processing the IPv6 packets, if any IPv6 protocol exception is detected, the IPv6 Forwarder thread passes those packets to the fastpath manager thread for processing the exception.

The following IPv6 protocol exceptions will result in an exception condition:

- The destination of the packet is a multicast address.
- The packet is destined to a network or host that does not have an entry in the FIB table.
- The packet must be forwarded to a host or a gateway whose Ethernet address is not resolved.
- The packet is destined to an interface that is owned by the `ipfwd` application (local delivery).

Note – For packets originated from the host (control domain), the fragmentation is taken care of by the Oracle Solaris OS stack, and only IPv6 packets handled internally are not fragmented before forwarding.

IPv6 STREAMS Module (lwmodip6)

This module is used for the processing of IPv6 packets that are destined to the local interfaces. The module passes IPv6 packets to and from the control plane Oracle Solaris IPv6 layer and the data plane `ipfwd` application. It uses IPC or TIPC to communicate with the data plane application.

Note – This module must not be used when `vnet` is used for exception packet transfers.

Fastpath Manager

The fastpath manager performs the following functions related to IPv6 exception handling:

- Interfaces with the control plane components such as `lwmodip6`, `fibctl`, or `ifctl` using IPC, or TIPC.
- Passes egress packets from control plane to transmit strands.
- Receives packets from IP forwarder strands and sends them to control plane.
- Interfaces with the `vnet` driver transmit and receive strands to enqueue and dequeue exception packets to and from the control plane.
- Executes the MAC Address learning algorithm and the FIB management when exception handling is done using `vnet` communication.

Exception Path Tools

The following tools are required to use the `ipfwd` application with exception handling and neighbor discovery (ND) handling:

`ifctl`

See [“Control Plane Components and Utilities”](#) on page 178.

`fibctl`

See [“Control Plane Components and Utilities”](#) on page 178.

fibctl.sh

fibctl.sh is a wrapper for fibctl to allow manipulating individual entries in the FIB table. It keeps a local copy of the table, makes the necessary changes and commits them to the data-plane using fibctl. The following shows the usage:

fibctl.sh add/del/mac *prefix* [*gateway interface*]

```
fibctl.sh add ::/0 fe80::200:ff:fe00:100 vnet1:0
fibctl.sh del fe80::200:ff:fe00:100/64
fibctl.sh mac 3ffe:501:ffff:101:200:ff:fe00:101 00:00:00:00:01:01
```

ipfwd_sync.d

ipfwd_sync.d can be started without parameters. It monitors events in the control plane (Oracle Solaris OS) and interacts with the data plane using the described exception path tools.

Note – With vnet exception handling, fibctl.sh and ipfwd_sync.d are not used.

▼ To Compile the Reference Application

1. Copy the ipfwd reference application from /opt/SUNWndps/src/apps/ipfwd directory to a directory location.
2. Execute the build script in that location.

▼ To Compile the IPv6 Forwarding Application With Exception Handling Using Sun Netra DPS

1. On a system that has /opt/SUNWndps installed, go to the user_workspace/src/apps/ipfwd application directory.
2. Build the application using the build script.
The ldoms and the ipv6 options must be provided.

```
# ./build cmt2 10g_niu ldoms ipv6
```

Compiling the lwmodip6 STREAMS module

The lwmodip6 module is provided in ipfwd/src/solaris/module directory. It is built using the build file in this directory. The following shows the usage:

```
./build ipv4|ipv6 [tipc]
```

The following build options are provided:

- ipv4 – Builds lwmod for IPv4 interface.
- ipv6 – Builds lwmod for IPv6 interface.
- tipc – Uses TIPC to communicate with data plane. Otherwise, it uses TNIPC.

▼ To Build the lwmodip6 Module for IPv6 Exception Handling Using IPC

```
% ./build ipv6
```

▼ To Build the lwmodip6 Module for IPv6 Exception Handling Using TIPC

```
% ./build ipv6 tipc
```

▼ To Run the ipfwd Application With IPv6 Exception Handling

1. Set up logical domains on the target system with one Sun Netra DPS domain and the following Oracle Solaris domains:

- primary – Primary domain for running Logical Domain Manager (ldm).
- ndps – Sun Netra DPS domain for running the Sun Netra DPS ipfwd application.
- ldg2 – Oracle Solaris domain for running the excpd application.
- ldg3 – Oracle Solaris domain for establishing IPC channels.

One vnet interface is needed in ldg2 for each data plane port. These vnet interfaces are connected to isolated vswitches in the primary domain.

2. Add vswitches for each vnet that will be configured:

```
# ldm add-vswitch vsw1 primary  
# ldm add-vswitch vsw1 primary
```

3. Reboot the primary domain for these changes to take effect.

4. Add the vnet interfaces to the control domain (ldg2).

The MAC addresses must be the same as that of Sun Netra DPS domain's interfaces.

```
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet1 vsw1 ldg2
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet2 vsw2 ldg2
```

5. Run the ipfwd application that is compiled with exception handling:

a. Copy the ipfwd binary to the tftpboot server:

```
% cp user-directory/ipfwd/code/ipfwd/ipfwd tftpserver/tftpboot
```

b. At the ok prompt on the target machine, type:

```
ok boot network-device:,ipfwd
```

c. Copy the IPv6 STREAMS module to ldg2, and load it:

```
# modload lwmodip6
```

6. Enable the vnet interface for each data plane port in ldg2, and insert lwmod6 for each interface:

```
# ifconfig vnet1 inet6 plumb
# ifconfig vnet1 inet6 modinsert lwmodip6@1
# ifconfig vnet2 inet6 plumb
# ifconfig vnet2 inet6 modinsert lwmodip6@1
```

7. Copy the ipfwd_sync.d application, the fibctl application, and the ifctl application to the ldg2 domain, and start the synchronization, redirecting the output to a log file:

```
# ./ipfwd_sync.d > ipfwd_sync.log &
```

From this moment the interface or routing table changes of the control plane will be reflected in the data-plane data structures.

8. Synchronize the interfaces by bringing up the IPv6 interfaces.

```
# ifconfig vnet1 inet6 up
# ifconfig vnet2 inet6 up
```

▼ To Compile the IPv6 Forwarding Application With Exceptional Handling Using `vnet`

1. On a system that has `/opt/SUNWndps` installed, go to the `user_workspace/src/apps/ipfwd` application directory.

2. Build the application using the build script.

The `ldoms`, `excp`, `vnet`, and `ipv6` options must be provided.

```
# ./build cmt2 10g_niu ldoms excp vnet ipv6
```

▼ To Run the `ipfwd` Application With IPv6 Exception Handling

1. Set up the logical domains on the target system with one Sun Netra DPS domain and the following Oracle Solaris OS domains:

- `primary` – Primary domain for running logical domain manager (`ldm`)
- `ndps` – Sun Netra DPS domain for running the Sun Netra DPS `ipfwd` application
- `ldg2` – Oracle Solaris domain for handling exceptions
- `ldg3` – Oracle Solaris domain for establishing IPC channels

One `vnet` interface is needed in `ldg2` for each data plane port. One `vnet` interface is needed in `ndps` for each Ethernet port in the data plane. One `vswitch` is needed in the `primary` domain for each data plane port. Add the `vswitch` devices in the `primary` domain for the `vnet` devices in `ldg2` and `ndps` that will be used for exception handling.

```
# ldm add-vswitch vsw1 primary
# ldm add-vswitch vsw2 primary
```

2. Reboot the primary domain for these changes to take effect.

3. Add the `vnet` interfaces to the control domain `ldg2`.

The MAC address must be the same as the interfaces in the Sun Netra DPS domain.

```
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet1 vsw1 ldg2
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet2 vsw2 ldg2
```

4. Add the `vnet` interface that is used for exception handling in `ndps`.

```
# ldm add-vnet vnet1 vsw1 ndps
# ldm add-vnet vnet2 vsw2 ndps
```

▼ Run the `ipfwd` Application That Is Compiled With Exception Handling

1. Place the `ipfwd` binary on the `tftpboot` server:

```
# cp user-dir/ipfwd/code/ipfwd/ipfwd tftpserver-boot/tftpboot
```

2. At the `ok` prompt on the target machine, type:

```
ok boot network-device:,ipfwd
```

3. Configure one `vnet` interface for each data plane port in `ldg2`:

```
# ifconfig vnet1 inet6 plumb
# ifconfig vnet2 inet6 plumb
# ifconfig vnet1 inet6 up
# ifconfig vnet2 inet6 up
```

4. Place the `fibctl` and the `ifctl` application in the `ldg2` domain.
5. Configure the Sun Netra DPS network interfaces with the `ifctl` application.

```
# ./ifctl port0 0 fe80::214:4fff:fe9c:86f4 mtu 1500 up
# ./ifctl port1 1 fe80::214:4fff:fe8:ebec mtu 1500 up
```

6. Configure the `vnet` exception handling in `ndps`.

The name chosen for this interface must be in the form `vnetinstance-number`. Use the `ldm list-bindings -e ndps` command in the primary domain to obtain the instance number. The number listed under the `DEVICE` column in the output of this command is the instance number. Also, a valid IP address must not be assigned to the `vnet` interface that is used for exception handling. This device is operated purely as a L2 device.

```
# ./ifctl vnet1 1 0::0 mtu 1500 up
# ./ifctl vnet2 2 0::0 mtu 1500 up
```

7. Configure the FIB table using `fibctl`.

```
# ./ifctl fibctl_tables
```

The MAC address learning module starts learning MAC address for the next-hops mentioned in the FIB table. The data plane will start transferring packets to and from the control plane using the `vnet` interface in `ndps`.

▼ To Compile the IPv6 Forwarding Application Using `vnet` Exceptional Handling in a Linux Guest Logical Domain

1. **On a system that has `/opt/SUNWndps` installed, go to the `user_workspace/src/apps/ipfwd` application directory.**
2. **Enable the `-DVNET_TIPC_CONFIG` flag in the required makefile.**
For example: `Makefile.nxge`
3. **Build the application using the build script.**
The `ldoms`, `excp`, `vnet`, `tipc`, and `ipv6` options must be provided.

```
# ./build cmt2 10g_niu ldoms excp tipc vnet ipv6
```

▼ To Run the `ipfwd` Application Using IPv6 Exception Handling in a Linux Guest Logical Domain

1. **Set up the logical domains on the target system with one Sun Netra DPS domain and the following guest domains:**
 - `primary` – Primary domain for running logical domain manager (`ldm`)
 - `ndps` – Sun Netra DPS domain for running the Sun Netra DPS `ipfwd` application
 - `ldg2` – Linux OS domain for handling exceptions
 - `ldg3` – Oracle Solaris OS domain for executing the `tnsmctl -P -v` command

2. Add one vnet interface in ldg2 for each data plane port.

One vnet interface is needed in ndps for each Ethernet port in the data plane, and one vswitch is needed in the primary domain for each data plane port. Add the vswitch devices in the primary domain for the vnet devices in ldg2 and ndps for exception handling.

```
# ldm add-vswitch vsw1 primary
# ldm add-vswitch vsw2 primary
```

3. Reboot the primary domain for these changes to take effect.

4. Add the vnet interfaces to the control domain ldg2.

The MAC address must be the same as the interfaces in the Sun Netra DPS domain.

```
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet1 vsw1 ldg2
# ldm add-vnet mac-addr=XX:XX:XX:XX:XX:XX vnet2 vsw2 ldg2
```

5. Add the vnet interface for exception handling in ndps.

```
# ldm add-vnet vnet1 vsw1 ndps
# ldm add-vnet vnet2 vsw2 ndps
```

▼ Run the ipfwd Application That Is Compiled With Exception Handling

1. Place the ipfwd binary in the tftpboot server:

```
# cp user-dir/ipfwd/code/ipfwd/ipfwd tftpserver-boot/tftpboot
```

2. At the ok prompt on the target machine, type:

```
# boot network-device:,ipfwd
```

3. Configure one vnet interface for each data plane port in ldg2:

```
# ifconfig vnet1 inet6 up
# ifconfig vnet2 inet6 up
```

4. Configure the Sun Netra DPS TIPC node and Linux TIPC node.

Note that the `tn-tipc-config` tool for Linux must be built from the `SUNWndpsd` package.

```
# ./tn-tipc-config -addr=10.3.5
# ./tn-tipc-config -be=eth:vnet1/10.3.0
# tipc-config -addr=10.3.4
# tipc-config -be=eth:eth1/10.3.0
```

See “[To Configure the Environment for TIPC](#)” on page 315 for instructions to build this tool.

5. Place the `fibctl` and the `ifctl` application in the `ldg2` domain.

6. Configure the Sun Netra DPS network interfaces with the `ifctl` application.

```
# ./ifctl port0 0 fe80::214:4fff:fe9c:86f4 mtu 1500 up
# ./ifctl port1 1 fe80::214:4fff:fe8:ebec mtu 1500 up
```

7. Configure the exception handling `vnet` interface in `ndps`.

The name chosen for this interface must be in the form `vnetinstance-number`. Use the `ldm list-bindings -e ndps` command in the primary domain to obtain the instance number. The number listed under the `DEVICE` column is the instance number. Also, a valid IP address must not be assigned to the `vnet` interface that is used for exception handling. This device is operated purely as a L2 device.

```
# ./ifctl vnet1 1 0::0 mtu 1500 up
# ./ifctl vnet2 2 0::0 mtu 1500 up
```

8. Configure the FIB table using `fibctl`.

```
./fibctl fibctl_tables
```

The MAC address learning module starts learning MAC address for the next-hops mentioned in the FIB table. The data plane will start transferring packets to and from the control plane using the `vnet` interface in `ndps`.

Differentiated Services Reference Application

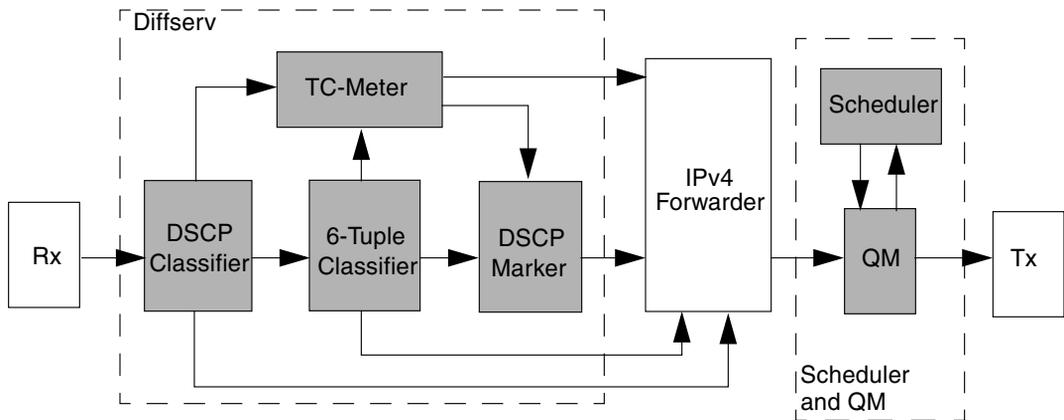
The differentiated Services (DiffServ) reference application is integrated with the IP forwarding application. The DiffServ data path consists of classifier, meter, marker, and policing components. These components provide quality-of-services (QoS) features for traffic entering the node and avoids congestion in the network. These components can be arranged in the pipeline such that each component performs specific task and propagates the result (traffic class and policing information) to the next component.

The following are major features of DiffServ:

- [“Classifiers” on page 221](#)
- [“Policing \(Meter\)” on page 222](#)
- [“DSCP Marker” on page 222](#)
- [“Shaping” on page 222](#)
- [“Building the DiffServ Application” on page 223](#)
- [“DiffServ Command-Line Interface Implementation” on page 224](#)
- [“Command-Line Interface for the IPv4-DiffServ Application” on page 224](#)

FIGURE 11-4 shows the arrangement of the components in the data path. The scheduler and queue manager are executed in a separate thread, whereas the other components are located in the forwarding thread. The following sections describe the functions of the different parts.

FIGURE 11-4 IPv4 DiffServ Internal Data Path



Classifiers

This section describes the *DiffServ* classifiers.

Differentiated Services Code Point Classifier

The differentiated services code point (DSCP) classifier (RFC 2474) fast path component sets QoS variables (flow and color) based on the DSCP value extracted from the IPv4 packet header and directs packets to the proper next component (meter, marker, and IPv4) for further processing. The DSCP classifier always remain enabled.

6-Tuple Classifier

The 6-tuple classifier fast path component performs an exact-match lookup on the IPv4 header. The classifier maintains a hash table with exact-match rules. Thus, a table lookup can fail only if there is no static rule defined. An empty rule corresponds to best-effort traffic. As a result, on a lookup failure a packet is assigned to the best-effort service (default rule) and passed on for further processing. The classifier slow path component configures the hash table used by the classifier fast path component. 6-tuple classifier can be enabled or disabled at run time.

Policing (Meter)

The three-color (TC) meter implements two metering algorithms: single-rate three-color meter (SRTCM) and two-rate three-color meter (TRTCM).

Single-Rate Three-Color Marker

The single-rate three-color marker (SRTCM) meters an IP packet stream and marks its packets green, yellow, or red. Marking is based on a committed information rate (CIR) and two-associated burst sizes, a committed burst size (CBS) and an excess burst size (EBS). A packet is marked green if it does not exceed the CIR. The packet is marked yellow if it does exceed the CBS, but not the EBS. Otherwise, the packet is marked red.

Two-Rate Three-Color Marker

The two-rate three-color marker (TRTCM) meters an IP packet stream and marks its packets green, yellow, or red. A packet is marked red if it exceeds the peak information rate (PIR). Otherwise, it is marked either yellow or green depending on whether it exceeds or does not exceed the committed information rate (CIR).

DSCP Marker

The DSCP marker updates the type-of-service (TOS) field in the IPv4 header and recomputes the IPv4 header checksum

Shaping

This section includes the deficit round robin scheduler and queue manager.

Deficit Round Robin Scheduler

The deficit round robin (DRR) scheduler schedules packets in a flexible queuing policy with priority concept. With this scheduler, the parameters are the number of sequential service slots that each queue can get during its service turn. The number of services for each queue depends on the value of its parameter called *deficit factor*. The deficit of queue is reduced as the scheduler schedules packets from that queue. The maximum deficit of each queue can be configured and is called weight of that

queue. The DRR scheduler will schedule the packets by considering the packet size of the packet at the top of the queue. Queues are still served in round robin fashion (cyclically) in a preassigned order.

Queue Manager

The queue manager performs enqueue and dequeue operations on the queues. The queue manager manages an array of queues, with each queue corresponding to a particular per hop behavior (PHB), for queuing packets per port. The queue manager receives enqueue requests from the IPv4-DiffServ pipeline. On receiving the enqueue request, the queue manager places the packet into the queue corresponding to the PHB indicated by the DSCP value in the packet. The queue manager maintains the state for each queue and uses the tail drop mechanism in case of congestion.

The queue manager receives the dequeue requests from the scheduler. The dequeue request consists of the PHB and the output port. Packets from the queue corresponding to this PHB and output port is dequeued and the dequeued packet is placed on the transmit queue for the output port.

Building the DiffServ Application

To build the DiffServ application, specify the `diffserv` keyword on the build script command line. All files of the DiffServ data path implementation are located in the `diffserv` subdirectory of `stc/app` in the IP forwarding application. The DiffServ application requires an logical domain environment, as all configuration is through an application running on an Oracle Solaris control domain that communicates with the data plane application through IPC.

For example, to build the DiffServ application to make use of both NIU ports on an UltraSPARC T2-based system, use the following command:

```
% ./build cmt2 10g_niu ldoms diffserv no_freeq 2port
```

DiffServ Command-Line Interface Implementation

The IPv4 Forwarding Information Base (FIB) table configuration (`fibctl`) command-line interface (CLI) has been extended to support configuration of DiffServ tables. This support behavior is the same as the FIB table configuration protocol over IPC between the control plane and data plane logical domains. Support is provided for configuring (choosing) the following DiffServ tables:

- DSCP classifier table
- Classifier 6-tuple table
- STRCM and TRTCM table
- Queue manager configuration table
- Scheduler configuration table

▼ To Build the Extended Control Utility

- **Type the following command in the `src/solaris` subdirectory of the IP forwarding reference application:**

```
% gmake DIFFSERV=on
```

Command-Line Interface for the IPv4-DiffServ Application

This section contains descriptions of the CLI commands for the IPv4-DiffServ application.

DSCP Classifier

The DSCP classifier supports the following commands.

`add`

Adds the DSCP classifier entry in the DSCP table.

Syntax

```
diffserv dscp add DSCP-value port-number flow-id color-id class-id next-block
```

Parameters

- *DSCP-value* – DSCP value should be greater than 0 and less than 64.
- *port-number* – Port number should be less than NUM_PORTS.
- *flow-id* – ID used to identify the traffic flow to which the packet belongs.
- *color-id* – ID should be green, yellow, or red.
- *class-id* – ID used to identify the queue number within an output port.
- *next-block* – Next block should be meter, marker, or fwder.

Example

```
fibctl> diffserv dscp add 1 0 1 green 1 meter
```

delete

Deletes DSCP classifier entry from DSCP table.

Syntax

```
diffserv dscp delete DSCP-value port-number
```

Parameters

- *DSCP-value* – DSCP value should be greater than 0 and less than 64.
- *port-number* – Port number should be less than NUM_PORTS.

Example

```
fibctl> diffserv dscp delete 1 0
```

update

Updates the existing DSCP classifier entry in DSCP table.

Syntax

```
diffserv dscp update DSCP-value port-number flow-id color-id class-id next-block
```

Parameters

- *DSCP-value* – DSCP value should be greater than 0 and less than 64.
- *port-number* – Port number should be less than NUM_PORTS.
- *flow-id* – ID used to identify the traffic flow to which the packet belongs.
- *color-id* – ID should be green, yellow, or red.
- *class-id* – ID used to identify the queue number within an output port.
- *next-block* – Next block should be meter, marker, or fwder.

Example

```
fibctl> diffserv dscp update 1 0 1 yellow 1 fwder
```

purge

Purges the DSCP table.

Syntax

```
diffserv dscp purge
```

display

Displays the DSCP table.

Syntax

```
diffserv dscp display
```

6-Tuple Classifier

The 6-tuple classifier supports the following commands:

add

Adds classifier 6-tuple entry in 6-tuple table.

Syntax

```
diffserv class6tuple add SrcIp DestIp Proto Tos SrcPrt DestPrt IfNum flow-id  
color-id next-block class-id
```

Parameters

- *SrcIp* – Source IP address (for example, 192.168.1.5) in the IP header of packet.
- *DestIp* – Destination IP address (for example, 192.168.1.5) in the IP header of packet.
- *Proto* – IP protocol field in the IP header of packet.
- *Tos* – Differentiated services code point (6 bits of TOS field).
- *SrcPrt* – Source port number in the TCP/UDP header packet.
- *DestPrt* – Destination port number in the TCP/UDP header packet.
- *IfNum* – Input port starting from port 0, on which the packet is received.
- *flow-id* – ID used to identify the traffic flow to which the packet belongs.
- *color-id* – ID used to identify the packet drop precedence level (green, yellow, or red).
- *next-block* – Used to identify the next packet processing block meter, marker, and fwder.
- *class-id* – ID used to identify the queue number within an output port (for example: ef, af0, af1, af2, af3, be).

Example

```
fibctl1> diffserv class6tuple add 211.2.9.195 192.168.115.76 17  
16 61897 2354 0 50 green meter 44
```

delete

Deletes 6-tuple classifier entry from 6-tuple table.

Syntax

```
diffserv class6tuple delete SrcIp DestIp Proto Tos SrcPrt DestPrt IfNum
```

Parameters

- *SrcIp* – Source IP address (for example, 192.168.1.5) in the IP header of packet.
- *DestIp* – Destination IP address (for example, 192.168.1.5) in the IP header of packet.
- *Proto* – IP protocol field in the IP header of packet.
- *Tos* – Differentiated services code point (6 bits of TOS field).
- *SrcPrt* – Source port number in the TCP/UDP header packet.
- *DestPrt* – Destination port number in the TCP/UDP header packet.
- *IfNum* – Input port starting from port 0, on which the packet is received.

Example

```
fibctl> diffserv class6tuple delete 211.2.9.195 192.168.115.76  
17 16 61897 2354 0
```

update

Updates the existing 6-tuple classifier entry in 6-tuple table.

Syntax

```
diffserv class6tuple update SrcIp DestIp Proto Tos SrcPrt DestPrt IfNum  
flow-id color-id next-block class-id
```

Parameters

- *SrcIp* – Source IP address (for example, 192.168.1.5) in the IP header of packet.
- *DestIp* – Destination IP address (for example, 192.168.1.5) in the IP header of packet.
- *Proto* – IP protocol field in the IP header of packet.
- *Tos* – Differentiated services code point (6 bits of TOS field).
- *SrcPrt* – Source port number in the TCP/UDP header packet.
- *DestPrt* – Destination port number in the TCP/UDP header packet.

- *IfNum* – Input port starting from port 0, on which the packet is received.
- *flow-id* – ID used to identify the traffic flow to which the packet belongs.
- *color-id* – ID used to identify the packet drop precedence level (green, yellow, or red).
- *next-block* – Used to identify the next packet processing block meter, marker, and fwder.
- *class-id* – ID used to identify the queue number within an output port (for example: ef, af0, af1, af2, af3, be).

Example

```
fibctl> diffserv class6tuple update 211.2.9.195 192.168.115.76
17 16 61897 2354 0 50 red marker 44
```

purge

Purges the 6-tuple table.

Syntax

```
diffserv class6tuple purge
```

display

Displays the 6-tuple table.

Syntax

```
diffserv class6tuple display
```

enable or disable

Enables or disables the 6-tuple table.

Syntax

```
diffserv class6tuple enable|disable
```

Example

```
fibctl> diffserv class6tuple enable  
fibctl> class6tuple disable
```

TC Meter

The TC meter supports the following commands:

add

Adds a meter instance in TC meter table.

Syntax

```
diffserv meter add flow-id CBS EBS CIR EIR green-dscp green-action yellow-dscp  
yellow-action red-dscp red-action meter-type stat-flag
```

Parameters

- *flow-id* – ID used to identify the traffic flow to which the packet belongs.
- *CBS* – The value of the committed burst size (CBS) is larger than 0, it is larger than or equal to the size of the largest possible IP packet in the stream. *cbs* is measured in bytes.
- *EBS* – The value of the excess burst size (EBS) is larger than 0. It is larger than or equal to the size of the largest possible IP packet in the stream. EBS is measured in bytes.
- *CIR* – Committed information rate (CIR) at which a traffic source is signed up to send packets to the meter instance. It is measured in bytes-per-second. The *cir* should be in M-bytes per seconds.
- *EIR* – Excess information rate (EIR) at which a traffic source is signed up to send packets to the meter instance. It is measured in bytes-per-second. This is used only when TRTCM is enabled. The *eir* should be in megabytes-per-second.
- *green-dscp* – DSCP packet mark value for green packets.
- *green-action* – Select the next packet processing block for green packets (drop, fwder, and marker).
- *yellow-dscp* – DSCP packet mark value for yellow packets.
- *yellow-action* – Select the next packet processing block for yellow packets (drop, fwder, and marker).

- *red-dscp* – DSCP packet mark value for red packets.
- *red-action* – Select the next packet processing block for red packets (drop, fwder, and marker).
- *meter-type*
 - 0 – TRTCM color aware
 - 1 – TRTCM color blind
 - 2 – SRTCM color aware
 - 3 – SRTCM color blind
- *stat-flag*
 - 0 – Statistics disable
 - 1 – Statistics enable

Example

```
fibctl> diffserv meter add 1 1500 1500 1 1 12 marker 13 drop 14  
drop 1 1
```

delete

Deletes a meter instance in TC meter table.

Syntax

```
diffserv meter delete flow-id
```

Parameter

- *flow-id* – ID used to identify the traffic flow to which the packet belongs.

Example

```
fibctl> diffserv meter delete 1
```

update

Updates a meter instance in TC meter table.

Syntax

`diffserv meter update flow-id CBS EBS CIR EIR green-dscp green-action
yellow-dscp yellow-action red-dscp red-action meter-type stat-flag`

Parameters

- *flow-id* – ID used to identify the traffic flow to which the packet belongs.
- *CBS* – The value of the committed burst size (CBS) is larger than 0, it is larger than or equal to the size of the largest possible IP packet in the stream. *cbs* is measured in bytes.
- *EBS* – The value of the excess burst size (EBS) is larger than 0, it is larger than or equal to the size of the largest possible IP packet in the stream. EBS is measured in bytes.
- *CIR* – committed information rate (CIR) at which a traffic source is signed up to send packets to the meter instance. It is measured in bytes-per-second. The *cir* should be in megabytes-per-second.
- *EIR* – excess information rate (EIR) at which a traffic source is signed up to send packets to the meter instance. It is measured in bytes-per-second. This is used only when TRTCM is enabled. The *eir* should be in megabytes-per-second.
- *green-dscp* – DSCP packet mark value for green packets.
- *green-action* – Select the next packet processing block for green packets (`drop`, `fwder`, and `marker`).
- *yellow-dscp* – DSCP packet mark value for yellow packets.
- *yellow-action* – Select the next packet processing block for yellow packets (`drop`, `fwder`, and `marker`).
- *red-dscp* – DSCP packet mark value for red packets.
- *red-action* – Select the next packet processing block for red packets (`drop`, `fwder`, and `marker`).
- *meter-type*
 - 0 – TRTCM color aware
 - 1 – TRTCM color blind
 - 2 – SRTCM color aware
 - 3 – SRTCM color blind
- *stat-flag*
 - 0 – Statistics disable
 - 1 – Statistics enable

Example

```
fibctl> diffserv meter update 1 1500 1500 1 1 12 marker 13 drop  
14 drop 0 0
```

purge

Purges meter table.

Syntax

```
diffserv meter purge
```

display

Displays the TC meter table.

Syntax

```
diffserv meter display
```

stats

Displays the TC meter statistics.

Syntax

```
diffserv meter stats flow-id
```

Parameter

- *flow-id* – ID used to identify the traffic flow to which the packet belongs.

Example

```
fibctl> diffserv meter stats 1
```

Scheduler

The scheduler supports the following commands:

add

Configures weight for all AF classes and maximum rate limit for EF class.

Syntax

```
diffserv scheduler add output-port class-id weight
```

Parameters

- *output-port* – Port number should be less than NUM_PORTS.
- *class-id* – ID used to identify the queue number within an output port (for example: ef, af0, af1, af2, af3, be).
- *weight* – Maximum number of bytes to be scheduled. If class is ef, the weight will be bytes-per-seconds. Otherwise, the weight will be number of bytes.

Example

```
fibctl> diffserv scheduler add 1 af1 128
```

update

Updates weight for all AF classes and maximum rate limit for EF class.

Syntax

```
diffserv scheduler update output-port class-id weight
```

Parameters

- *output-port* – Port number should be less than NUM_PORTS.
- *class-id* – ID used to identify the queue number within an output port (for example: ef, af0, af1, af2, af3, be).
- *weight* – Maximum number of bytes to be scheduled. If class is ef, the weight will be bytes-per-seconds. Otherwise, the weight will be number of bytes.

Example

```
fibctl> diffserv scheduler update 1 af1 256
```

display

Displays scheduler table entries.

Syntax

```
diffserv scheduler display output-port
```

Parameter

output-port – Port number should be less than NUM-PORTS.

Example

```
fibctl> scheduler display 1
```

DiffServ References

TABLE 11-3 lists DiffServ references.

TABLE 11-3 DiffServ References

Reference	Document Descriptions
RFC 2474	Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers
RFC 2475	An Architecture for Differentiated Services
RFC 2597	Assured Forwarding PHB Group
RFC 2697	A Single-Rate Three-Color Marker
RFC 3246	An Expedited Forwarding PHB (Per-Hop Behavior)
RFC 3260	New Terminology and Clarifications for DiffServ
RFC 4115	A Differentiated Service Two-Rate, Three-Color Marker with Efficient Handling of in-Profile Traffic

Generic Routing Encapsulation Reference Application

The generic routing encapsulation (GRE) reference application is integrated with the IP forwarding application. Topics include:

- [“Generic Routing Encapsulation Introduction” on page 237](#)
- [“References” on page 237](#)
- [“Data Plane Architecture” on page 237](#)
- [“GRE Command-Line Interface Implementation” on page 241](#)
- [“Directory Structure” on page 241](#)
- [“To Compile the GRE Code” on page 241](#)
- [“To Run the IPv4 and GRE Application” on page 242](#)
- [“CLI for the IPv4-GRE Application” on page 243](#)

Generic Routing Encapsulation Introduction

Generic routing encapsulation (GRE) is a protocol for encapsulating a network layer protocol within another network layer protocol.

GRE is generally used as a tunneling protocol to encapsulate a wide variety of network layer packets inside IPv4 tunneling packets. The original network layer packet becomes the payload for the final packet.

For example, a node has a packet that needs to be encapsulated and sent to another node. This packet is then encapsulated using the generic routing encapsulation protocol. A delivery IPv4 header is added to the GRE encapsulated packet and this packet is forwarded to its destination over the public IPv4 network. At the destination, the GRE header and the delivery header are decapsulated, and the payload packet is forwarded in the local network.

References

TABLE 11-4 lists references for the GRE protocol.

TABLE 11-4 GRE Reference Documentation

Reference Number	Description
RFC 2784	This document specifies a protocol for performing encapsulation of an arbitrary network layer protocol over another arbitrary network layer protocol.
RFC 2890	This document describes extensions by which two fields, key and sequence number, can be optionally carried in the GRE header.

Data Plane Architecture

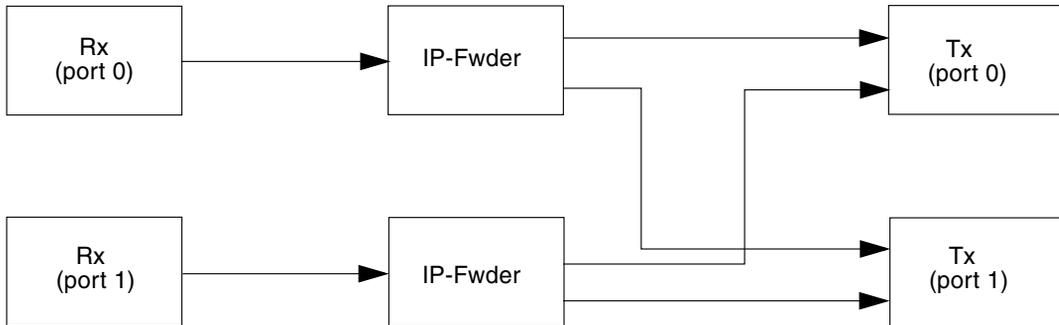
The data plane architecture for the GRE implementation on Sun UltraSPARC T1 and T2 boards is described in this section.

The GRE encapsulator and GRE decapsulator components are included in the data plane. The GRE encapsulator adds the GRE header and the delivery header to the payload packet. The GRE decapsulator removes the delivery header and GRE header from the encapsulated packet.

IPv4 Forwarding Data Plane

FIGURE 11-5 shows a diagram of the IPv4 forwarding.

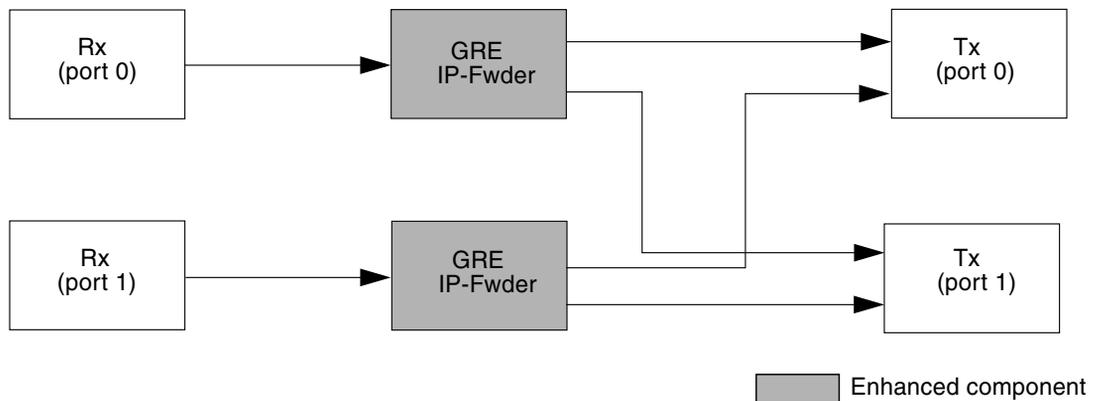
FIGURE 11-5 IPv4 Forwarding



GRE Over IPv4 Data Plane

FIGURE 11-6 shows a diagram of the GRE over IPv4 data plane.

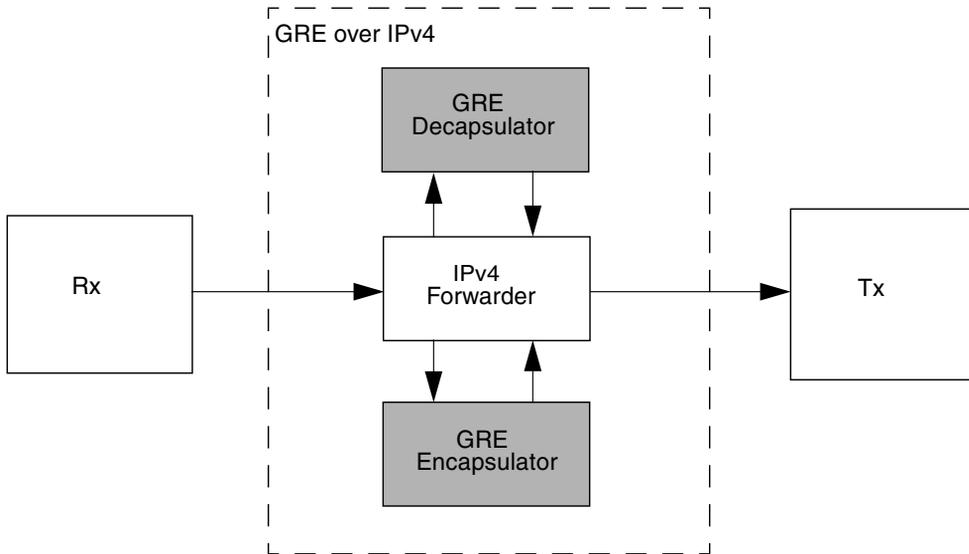
FIGURE 11-6 GRE Over IPv4 Data Plane



GRE Over IPv4 Data Plane Internal Block Diagram

FIGURE 11-7 shows the GRE over IPv4 data plane internal block diagram.

FIGURE 11-7 GRE Over IPv4 Data Plane Internal Block Diagram



GRE Over IPv4 Application

The following describes the GRE over IPv4 application.

IPv4 Forwarder

When a tunnel endpoint decapsulates a GRE packet that has an IPv4 packet as the payload, the destination address in the IPv4 payload packet header is used to forwards the packet and the TTL of the payload packet is decremented. Take care while forwarding such a packet, because if the destination address of the payload packet is the encapsulator of the packet (that is, the other end of the tunnel), looping can occur. In this case, the packet must be discarded.

GRE Encapsulator

When a node has a packet that needs to be encapsulated and forwarded, this packet is called the payload packet. The payload is first encapsulated in the GRE header. The resulting GRE packet is then encapsulated in the IPv4 protocol. GRE packets that are encapsulated within IPv4 use IPv4 protocol type 47.

The GRE encapsulator inserts the key field in the GRE header as according to the RFC 2890 document. The GRE encapsulator also inserts the Sequence Number field in the GRE header as according to the RFC 2890 document. See [“GRE Reference Documentation” on page 237](#).

GRE Decapsulator

When a node receives GRE encapsulated packet for local delivery, the node checks if the IPv4 protocol type is set to 47. If the IPv4 protocol type is set to 47, then the packet is given to the GRE decapsulator. The GRE decapsulator removes the GRE header, and the packet is given to the IPv4 forwarder to forward the packet in the local network. The GRE decapsulator uses the Sequence Number field in the GRE header to establish the order in which packets have been transmitted from the GRE encapsulator to the GRE decapsulator.

Key and Sequence Number Extensions to GRE

The RFC 2890 document (see [“GRE Reference Documentation” on page 237](#)) describes enhancements by which two fields, key and sequence number, can be optionally carried in the GRE header. The key field identifies an individual traffic flow within a tunnel. The sequence number field maintains the sequence of packets within the GRE tunnel.

When the decapsulator receives an out-of-sequence packet, the decapsulator discards the packet. A packet is considered out-of-sequence if the sequence number of the received packet is less than or equal to the sequence number of the last successfully decapsulated packet.

GRE decapsulator maintains a buffer per flow (flow is identified by the key number). This buffer holds the packets with the sequence number gap. When the GRE decapsulator receives an in-sequence packet, the decapsulator checks the sequence number of the packet at the head of the buffer. If the next in-sequence packet has been received, the receiver decapsulates it as well as the following in-sequence packets that may be present in the buffer.

The packets do not remain in the buffer indefinitely but they are decapsulated once they remain in the buffer for `OUTOFORDER_TIMER` mini-seconds.

GRE Command-Line Interface Implementation

The IPv4 forwarding information base (FIB) table configuration (`fibctl`) command-line interface (CLI) has been extended to support configuration of GRE tables. GRE related configuration commands are added to the existing FIB table configuration protocol over IPC between the control plane and the data plane logical domains. The following parameters are provided for configuring the GRE table:

- GRE configuration table
Configuration contains the source IP and destination IP of tunnel end points. The IP addresses of the tunnel end points must be public IP addresses.
- GRE key number
The GRE key number is configured through the CLI.
- IPv4 forwarding table is modified to accommodate next hop type and tunnel ID.

Directory Structure

TABLE 11-5 lists the GRE directory structure.

TABLE 11-5 GRE Directory Structure

Directory	Description
<code>ipfwd/src/app/gre</code>	Source code for GRE components
<code>ipfwd/src/solaris</code>	Control plane CLI code
<code>ipfwd/code</code>	Generated code
<code>ipfwd/code/ipfwd</code>	Binary

▼ To Compile the GRE Code

1. Copy the `ipfwd` reference application from the `/opt/SUNWndps/src/apps/ipfwd` directory to a desired directory location.
2. Execute the build script in that location.

▼ To Compile the IPv4 and GRE Application Using Sun Netra DPS

1. On a system that has `/opt/SUNWndps` installed, go to the `user-workspace/src/apps/ipfwd` application directory.
2. To enable GRE, execute the build script:

```
% ./build cmt2 10g_niu ldoms gre
```

▼ To Compile the Command-Line Interface Application

- Go to the `src/apps/ipfwd/src/solaris` directory, and type the following:

```
% gmake clean  
% gmake GRE=on
```

▼ To Run the IPv4 and GRE Application

1. Copy the `ipfwd` binary to the `tftpbboot` server:

```
% cp user-directory/ipfwd/code/ipfwd/ipfwd tftpbboot-server/tftpbboot/
```

Note – You might need to use `ftp` or other applications to transfer this binary file.

2. At the `ok` prompt on the target machine, type:

```
ok boot network_device:, ipfwd
```

▼ To Run the CLI Application

1. Set up logical domains on the target system with one Sun Netra DPS domain and the following Oracle Solaris domains:
 - `primary` – Primary domain for running the Logical Domain Manager (`ldm`)
 - `ndps` – Sun Netra DPS domain for running the Sun Netra DPS data plane application
 - `ldg2` – Oracle Solaris domain for running the `fibctl` application
 - `ldg3` – Oracle Solaris domain for establishing IPC channelsSee “To Build the `ifctl` and `fibctl` Utility” on page 183, for building the `fibctl` utility in the Oracle Solaris subtree.
2. Place the `fibctl` Oracle Solaris OS executable file into the `ldg2` domain.

```
% fibctl
```

CLI for the IPv4-GRE Application

The following commands are supported.

`add`

Adds the GRE entry in the GRE encapsulation table.

Syntax

```
gre add local-dest-addr local-dst-mask local-src-addr local-src-mask global-src-addr global-dst-addr
```

Parameters

- `local-dest-addr` – Destination network IPv4 address
- `local-dst-mask` – Destination network mask
- `local-src-addr` – Source network IPv4 address
- `local-src-mask` – Source network mask
- `global-src-addr` – Source IPv4 address of encapsulated packet
- `global-dst-addr` – Destination IPv4 address of encapsulated packet

Example

```
fibctl> gre add 192.168.115.0 255.255.255.0 211.2.9.0  
255.255.255.0 10.10.10.10 10.11.12.13
```

update

Updates the GRE entry in the GRE encapsulation table.

Syntax

```
gre update local-dest-addr local-dst-mask local-src-addr local-src-mask global-src-addr  
global-dst-addr
```

Parameters

- *local-dest-addr* – Destination network IPv4 address
- *local-dst-mask* – Destination network mask
- *local-src-addr* – Source network IPv4 address
- *local_src-mask* – Source network mask
- *global-src-addr* – Source IPv4 address of encapsulated packet
- *global-dst-addr* – Destination IPv4 address of encapsulated packet

Example

```
fibctl> gre update 192.168.115.0 255.255.255.0 211.2.9.0  
255.255.255.0 1.1.1.1 10.1.1.1
```

delete

Deletes the GRE entry in the GRE encapsulation table.

Syntax

```
gre delete local-dest-addr local-dst-mask local-src-addr local-src-mask
```

Parameters

- *local-dest-addr* – Destination network IPv4 address
- *local-dst-mask* – Destination network mask
- *local-src-addr* – Source network IPv4 address
- *local-src-mask* – Source network mask

Example

```
fibctl> gre delete 192.168.115.0 255.255.255.0 211.2.9.0  
255.255.255.0
```

purge

Purges the GRE encapsulation table.

Syntax

```
gre purge
```

Parameters

No parameters are required.

display

Displays the GRE encapsulation table.

Syntax

```
gre display
```

Parameters

No parameters are required.

GRE Reference Application Example

This GRE reference application example is run on an UltraSPARC T2 system. See [“Supported Systems” on page 2](#) for Sun systems supported by this application.

Required equipment:

- One UltraSPARC T2-based system
- One traffic generator port
- One NIU 10-Gbps Ethernet port (one XAUI card)
- One straight connect fiber cable

▼ To Build the GRE Reference Application

- Execute the following command:

```
% ./build cmt2 10g_niu ldoms gre -hash hash-policy
```

Traffic Generator Configuration

To run the encapsulation path:

- Frame data – select EthernetII, IPv4 + UDP/IP
- DA MAC – MAC ID of the target system’s port that receives the traffic from this traffic generator.
- IPv4 – If *hash-policy* is *ip-addr*:
SA=211.2.9.0
DA=192.168.115.0 ~ 192.168.115.255 (continue increment by 1)
- IPv4 – If *hash-policy* is *tcam-classify*:
SA=211.2.9.0
DA=192.168.115.1 ~ 192.168.115.8 (increment by 1 and repeat 8 counts)
- UDP – No action required.
- Payload – No action required.

To run the decapsulation path:

- Frame data – select EthernetII, IPv4 + GRE/IP
- Minimum packet size – 128-B
- DA MAC – MAC ID of the target system’s port that receives the traffic from this traffic generator.

- IPv4 (delivery header) – SA=x.x.x.x and DA=16.0.0.1
- IPv4 (inner header) – SA=x.x.x.x and DA=16.0.0.2

Note that the following fields must be present in the GRE header:

- Key field (this is a required field)
- Sequence number (this is a required field)
- Checksum/Reserve1 (valid checksum)
- fibctl application

On the Oracle Solaris domain (ldg2), run the following commands:

```
fibctl> connect
fibctl> write-table 1
fibctl> use-table 1
```

To run the encapsulation path, the following command is also required:

```
fibctl> gre add 192.168.115.0 255.255.255.0 211.2.9.0
255.255.255.0 1.1.1.1 10.1.1.1
```

Access Control List Reference Application

The access control list (ACL) reference application is integrated with the IP forwarding application. The ACL component classifies IPv4 packets using a set of rules. The classification can be done using the source and destination addresses and ports, as well as the protocol and the priority of the packet.

The algorithms (*trie*, *bspl*, and *hicut*) are used in the ACL library trade memory for speed. The rules are preprocessed to achieve a high lookup rate while using a lot of memory.

The ACL application can be built for using the following mechanism to transfer data between the control plane application (*acltool*) and data plane IP Forwarding application:

1. Use LDC to communicate
2. Use TIPC with IPC bearer
3. Use TIPC with vnet bearer

▼ To Build the ACL Application

ACL application can be build to use LDC or TIPC as medium to communicate with the control domain.

- **To build ACL to use LDC as medium, specify the `acl` keyword on the build script command line.**

For example:

```
% ./build cmt2 10g_niu ldoms acl
```

- **To build ACL to use TIPC as medium, specify the `acl` and `tipc` keywords on the build script command line.**

For example:

```
% ./build cmt2 10g_niu ldoms acl tipc
```

▼ To Run the ACL Application

The `ipfwd` application with ACL requires an logical domain environment because all configurations are done through an application running on an Oracle Solaris OS or Linux OS control domain. Both LDC and TIPC media are supported for Oracle Solaris OS domains. To use Linux as a control domain, use TIPC with `vnet` as TIPC bearer. The Sun Netra DPS domain needs to be configured with at least 16 Gbytes of memory, which is a requirement for the ACL application.

▼ To Configure the ACL Application Environment Using LDC

1. **Enable shared memory by adding the following line to the `/etc/system` file:**

```
set ldc:ldc_shmem_enabled = 1
```

2. Enable the ACL communication channel between the Sun Netra DPS domain and the Oracle Solaris OS control domain.

A special configuration channel must be set up between these domains. The channel is established as follows:

```
# ldm add-vdpcs shmem-server Netra-DPS-domain-name
# ldm add-vdpcc shmem-client shmem-server Solaris-control-domain-name
```

3. Add `/opt/SUNWndpsd/lib` to `LD_LIBRARY_PATH`.

▼ To Configure the ACL Application Environment Using TIPC

- See “[To Configure the Environment for TIPC](#)” on page 315 for instructions on how to configure the TIPC environment.

Command-Line Interface for the ACL Application

The `acltool` is a command-line tool that sends commands to the ACL engine running in the Sun Netra DPS domain. The interface is similar to `iptables(8)`. The major difference is that it does not take a chain as a parameter. There are three `acltool` binaries in the `SUNWndpsd` package:

- `/opt/SUNWndpsd/bin` – This directory contains the `acltool` for Oracle Solaris OS control domains.
 - `/opt/SUNWndpsd/bin/acltool` – This binary uses LDC as media to communicate with Sun Netra Data plane application.
 - `/opt/SUNWndpsd/bin/acltool.tipc` – This binary uses TIPC as media to communicate with Sun Netra Data plane application.
- `/opt/SUNWndpsd/linux/bin` – This directory contains the `acltool` for Linux control domain:
 - `/opt/SUNWndpsd/bin/acltool.tipc` – This binary uses TIPC as media to communicate with Sun Netra Data plane application.

The command options for `acltool` and `acltool.tipc` are the same in Oracle Solaris OS and Linux OS logical domains.

Following is a description of the various `acltool` commands and options.

```
% acltool --help
```

Usage

`acltool command [options]`

Help Command

- `-h` or `--help`
Prints usage help.

Control Commands

- `--init algorithm`
Initializes ACL engine using algorithm for packet lookup.
- `--start`
Starts the packet classification.
- `--stop`
Stops the packet classification.
- `--status`
Prints the status of the ACL engine.
- `-c` or `--config-file filename`
Reads rule commands from the configuration file.

Rule Commands

- `-A` or `--append rule`
Appends a rule.
- `-D` or `--delete rule`
Removes the matching rule.
- `-L` or `--list`
Lists all rules.

- `-F` or `--flush`
Flushes (removes) all rules.

Rule Specification Options

- `-p` or `--protocol num`
Protocol (`tcp`, `udp`, `icmp`) or protocol number.
- `-s` or `--source ip[/mask]`
Source ip prefix.
- `-d` or `--destination ip[/mask]`
Destination ip prefix.
- `-j` or `--jump num`
Specifies where to jump (action).
- `-g` or `--goto num`
Same as `--jump`.
- `--sport num[:num]`
Source protocol port.
- `--source-port num[:num]`
Source protocol port.
- `--dport num[:num]`
Destination protocol port.
- `--destination-port num[:num]`
Destination protocol port.
- `-v` or `--ipv4|6`
List rules with given IP version.
- `-o` or `--offset num`
Start listing from `num` offset.

▼ To Use `acltool` in a Linux OS Control Domain

1. **Copy** `libtnacltipc.so` **from** `/opt/SUNWndpsd/linux/lib` **to** `/usr/lib64` **directory in the Linux OS guest logical domain.**
2. **Copy** `acltool.tipc` **from** `/opt/SUNWndpsd/linux/bin` **to your working directory in the Linux OS guest logical domain.**

3. Execute the `acltool.tipc` tool.

For example:

```
# /working-dir/acltool.tipc options
```

Radio Link Protocol Reference 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 that can generate NAKs (negative acknowledges). The application must be modified to process reverse traffic.

▼ To Compile the RLP Application

1. Copy the `rlp` reference application from the `/opt/SUNWndps/src/apps/rlp` directory to a desired directory location.
2. Create the build script in that location.

Build Script

TABLE 11-6 shows the radio link protocol (`rlp`) application build script.

TABLE 11-6 `rlp` Application Build Script

Build Script	Usage
<code>./build</code> (See “Argument Descriptions” on page 253.)	Build <code>rlp</code> application to run on an Ethernet interface.

Usage

```
./build cmt type [ldoms] [arp] [profiler][-hash FLOW_POLICY]
```

Argument Descriptions

The following arguments are supported:

- *cmt*

Specifies whether to build the `ipfwd` application to run on the CMT1 (UltraSPARC T1) platform or CMT2 (UltraSPARC T2) platform.

 - `cmt1` – Build for CMT1 (UltraSPARC T1)
 - `cmt2` – Build for CMT2 (UltraSPARC T2)
- *type*
 - `4g` – Build `rlp` application to run on QGC (quad 1-Gbps `nxge` Ethernet interface).
 - `10g` – Build `rlp` application to run on 10-Gbps Ethernet (dual 10-Gbps `nxge` Ethernet interface).
 - `10g_niu` – Build `rlp` application to run on NIU (dual 10-Gbps UltraSPARC T2 Ethernet interface) on a CMT2-based system.
- [`ldoms`]

This is an optional argument specifying whether to build the `rlp` application to run on the logical domain environment. When this flag is specified, the `rlp` logical domain reference application will be compiled. If this argument is not specified, then the non-logical domain (standalone) application will be compiled. See [“How Do I Calculate the Base PA Address for NIU or Logical Domains to Use with the `tnsmct1` Command?”](#) on page 388.
- [`arp`]

This is an optional argument to enable `arp` and can run only on the logical domain environment.
- [`profiler`]

This is an optional argument that generate code with profiling enabled.
- [`-hash FLOW_POLICY`]

This is an optional argument used to enable flow policies. For more information, see [“Other RLP Options”](#) on page 255.

▼ To Build the RLP Application

1. In `/src/apps/rlp`, pick the correct build script, and run it.

For example, to build for 10-Gbps Ethernet on a Sun Netra or Sun Fire T2000 system, type the following at your shell window:

```
% ./build cmt1 10g
```

In this example, the `10g` option is used to build the RLP application to run on the Sun multithreaded 10-Gbps Ethernet. The `cmt` argument is specified as `cmt1` to build the application to run on UltraSPARC T1-based Sun 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
```

3. At the `ok` prompt on the target machine, type:

```
ok boot network-device: ,rlp
```

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

Default System Configuration

The following table shows the default system configuration.

TABLE 11-7 Default System Configuration

	NDPS domain (strand IDs)	IPC Polling Statistics (strand IDs)	Other domain (strand IDs)
CMT1 non-logical domain	0 to 31	31	N/A
CMT1 logical domain	0 to 19	18 and 19	20 to 31
CMT2 non-logical domain	0 to 63	63	N/A
CMT2 logical domain	0 to 39	38 and 39	40 to 63

The main files that control the system configurations are:

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

Default RLP Application Configuration

The following table shows the default RLP application configuration:

TABLE 11-8 Default RLP Application Configuration

Applications Runs On	Number of Ports Used	Number of Channels per Port	Total Number of Q Instances	Total Number of Strands Used
4-Gbps PCIE (nxge QGC)	4	1	4	12
10-Gbps PCIE (nxge 10-Gbps)	1	4	4	12
10-Gbps NIU (niu 10-Gbps):	1	8	8	24

The main files that control the application configurations are:

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

Other RLP Options

This sections includes instructions on how to use additional RLP options.

▼ To Bypass the rlp Operation

- To bypass the rlp operation (that is, receive --> transmit without rlp_process operation), uncomment the following line from Makefile.nxge for Sun multithreaded 10-Gbps and 4x1-Gbps PCIe Ethernet adapter:

```
-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.

Flow Policy for Spreading Traffic to Multiple DMA Channels

The user can specify a policy for spreading traffic into multiple DMA flows by hardware hashing or by hardware TCAM lookup (classification). See [TABLE 11-2](#) for flow policy options.

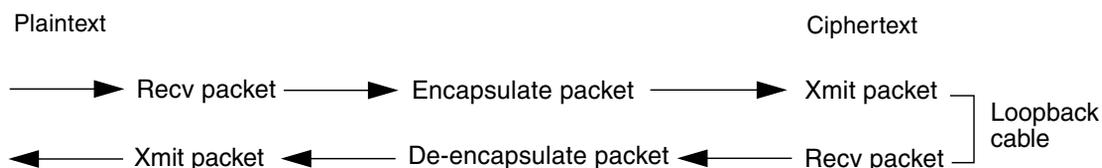
IPSec Gateway Reference Application

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 Application Architecture

The design calls for six Sun 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 11-8](#).

FIGURE 11-8 IPSec Gateway Application Architecture



Refer to the following RFC documents for a description of IPSec and the ESP protocol:

- 4301 – *Security Architecture for the Internet Protocol*
- 4303 – *IP 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 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 cache is used to quickly look up 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 or decrypt traffic

- Supported algorithms:

- AES (ECB/CBC/CTR) with 128/192/256 bits

- DES/3DES (ECB/CBC/FCB) with 128/192/256 bits

- RC4

- Integrity service

- Authenticate through optional hashing

- Supported algorithms: HMAC-SHA1, HMAC-SHA256, and HMAC-MD5

High-Level Packet Processing

The following describes functions of outbound and inbound packet processing.

Outbound Packets

The following list contains descriptions of the outbound packet processing:

- Receive packets from an ingress network port.
- Hash the source or destination IP address and port numbers.

- 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.
 - Transmit ciphertext packet from an egress network port.

Inbound Packets

The following list contains descriptions of the inbound packet processing:

- Receive Packets from an ingress network port.
- 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.
 - Transmit plain-text packets from an egress network port.

Security Association Database and Security Policy Database

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

- Crypto algorithm to use (AES, 3DES, and others)
- Key length
- Key
- Initial vector (IV)
- Type of service to apply (privacy-only or privacy + integrity)
- Hash algorithm (SHA1, SHA256, and so on)
- Hash length
- Hash key
- Sequence number

Refer to the `sadb.h` header file (`/opt/SUNWndpsec/src/libs/ipsec/sadb.h`) for all other fields in the SA database.

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 or inbound). The user must use software hashing and hash table lookups for every packet. Note that when this is ported to Sun multithreaded 10-Gbps Ethernet on PCIe, the packet classification features speed-up this hashing.

Outbound Packets and Inbound Packets

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

Outbound Packets

The user 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 others).

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, and protocol), the rule states one of the following:

- Single value (for example, matches on source address of 129.1.2.3)
- List of values (for example, matches either 129.1.2.3, 129.1.2.5, or 129.1.2.10)
- Range of values (for example, 129.1.1.3 to 129.1.1.10)
- Match-all (for example, any source port)
- Mask (for example, matches any source address 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. Due to the loopback configuration (refer to later sections for loopback configurations), the receive bound is receiving ciphertext packets from the transmit. Therefore, an SPI of the outbound packet plus 1 should be used as the SPI.

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 named 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 packet that uses the exact combination of selectors comes in, it needs to be looked up in the SPDC hash table using the `SPDC_HASH()` function. If found, immediate access to the SA is made.

The definitions of this SPDC and the function can be found in `sadb.h` and `sadb.c`, respectively.

This application does not hash on the protocol type because UDP or TCP protocols types are assumed due to the presence of the source and destination ports in the packets.

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. The application then compares the entry for a match, and if there is not a match, the function will walk the chain until one is found.

Inbound Packets

Inbound IPSec packets contain an ESP header with an SPI. The application parses the SPI, hashes it using `SPI_HASH_FROM_SPI()`, looks it up in the SPI hash table, and accesses the SA pointer from there. The application cannot use the same hashing as done for outbound packets because the selectors (source and destination IP address and ports) have been encapsulated and encrypted. Decryption cannot be done until the SA is looked up.

The SPI hash table is defined as:

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

Static Security Policy Database and Security Association Database

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 since the inbound SA is just a copy of the outbound SA, except for the SPI.

To perform various encryption and hashing scenarios, this SA entry is where the user needs 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 */
#ifdef INTEGRITY
#endif
#ifdef INTEGRITY
    IPSEC_SVC_ESP_PLUS_INT, /* service type */
#else
    IPSEC_SVC_ESP, /* service type */
#endif
#ifdef
    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 the user wants to test privacy (encryption), leave `INTEGRITY` commented out. No hashing will be done. If the user wants 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.

It is important to ensure the proper key lengths and IV lengths in the table.

You might 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 logical domain IPC mechanism to interface with Sun 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 Sun Netra DPS and informs the Sun Netra DPS application of updates through the logical domain 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 out of a Sun Netra DPS fast queue and calls:

```
IPSEC_Process_Plaintext_Pkt(mblk)
```

The main ciphertext packet processing thread is called `CiphertextRcvProcessLoop()`. The thread takes a packet off a fast queue and calls `IPSEC_Process_Ciphertext_Pkt(mblk)`.

Find the `IPSEC_Process_Plaintext_Pkt()` and `IPSEC_Process_Ciphertext_Pkt()` functions in `ipsec_proc.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, the user application should 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 Sun Netra DPS memory pool. The memory pool size is 2-Kbytes and the memory pool function 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).

The following shows how to calculate the location of the outbound packet:

```
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)

```

The following shows how to calculate the inbound packet:

```
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 Sun Netra DPS memory pools shown in [TABLE 11-9](#). The names and sizes are defined in `ipsecgw_config.h`:

TABLE 11-9 Sun 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 two threads: one to perform most of the packet encapsulation and de-encapsulation, and the other to perform the encryption and decryption and optional hashing.

An extra fast queue is inserted in each path. For example, the pipeline for the eight threads configurations is shown as follows:

```
PlaintextRcvPacket ->
  PlaintextRcvProcessLoop ->
    EncryptAndHash ->
      CiphertextXmitPacket -> Network port 1 ---->
                                     LOOPBACK
      <- CiphertextRcvPacket <- Network port 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` (message block), this data is piggy-backed 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-nxge`
This directory consists of IPsec code that supports the Sun multithreaded 10-Gbps Ethernet on PCI-E or on-chip NIU in UltraSPARC T2.
- `/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.

Build Script

This section contains descriptions of the usage and arguments supported by the build script.

Usage

```
./build cmt type [auth] [-hash FLOW_POLICY]
```

Argument Descriptions

- *cmt*

Specifies whether to build the IPSec Gateway application to run on the CMT1 platform or CMT2 platform.

- *cmt1* – Build for CMT1 (UltraSPARC T1)
- *cmt2* – Build for CMT2 (UltraSPARC T2)

- *type*

Specifies the application type. Available application types are shown as follows:

- *ipcrypto* – Build the *ipsecgw* application to run crypto on IP packets only (no IPSec protocol). This application configuration can be used to measure raw crypto overheads.
- *qgc* – Build the *ipsecgw* application to run on the Sun Multithreaded Quad Gigabit Ethernet.
- *10g_niu* – Build the *ipsecgw* application to run one application instance on the UltraSPARC T2 10-Gbps Ethernet (NIU).
- *niu_multi* – Build the *ipsecgw* application to run up to four application instances on the UltraSPARC T2 10-Gbps Ethernet.
- *niu_tunnel_in* – Build the *ipsecgw* application to run up to eight application instances on the UltraSPARC T2 10-Gbps Ethernet.
- *niu_tunnel_out* – Build the *ipsecgw* application to run up to eight application instances on the UltraSPARC T2 10-Gbps Ethernet.

- [*auth*]

This is an optional argument to apply authentication (hashing protocol) to the packet stream along with crypto. The hash algorithm is specified in the *sa_init_static_data.c* source file.

- [-hash *FLOW_POLICY*]

This is an optional argument used to enable flow policies. See [TABLE 11-2](#) for the flow policies for all flow policy options.

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

TABLE 11-10 lists the source files.

TABLE 11-10 Source Files

Source File	Description
<code>common.h</code>	Header file consists of common information.
<code>config.h</code>	Consists of receive buffer configuration information.
<code>debug.c</code>	Used when compiling in <code>DEBUG</code> mode (see <code>IPSEC_DEBUG</code> in the Makefile to turn on IPsec debugs). This file contains the debug thread that calls <code>teja_debugger_check_ctrl_c()</code> .
<code>init.c</code>	Main initialization code called by Sun Netra DPS runtime for setting up fast queues and initializing the Crypto library and the IPsec code.
<code>init_multi.c</code>	Main initialization code called by Sun Netra DPS runtime for setting up fast queues used by the IPsec multiple instances code.
<code>ip_crypto.c</code>	Location of the main application threads for the IPsec crypto (crypto only, no IPsec overhead).
<code>ipsec_niu_config.c</code>	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.
<code>ipsecgw.c</code>	Contains the main application threads.
<code>ipsecgw_config.c</code>	Assists user to map application tasks to CPU core and hardware strands.
<code>ipsecgw_flow.c</code>	Contains the classification flow entries.
<code>ipsecgw_flow.h</code>	Contains the definitions of the classification flow.
<code>ipsecgw_impl_config.h</code>	Contains the information related to <code>mb1k</code> , receive buffer sizes, number of channels, SA, SPDC.
<code>ipsecgw_niu.c</code>	Main application thread for the NIU configuration.
<code>ipsecgw_niu_multi.c</code>	Main application thread for the NIU multi-instances configuration.
<code>lb_objects.h</code>	Contains memory pool definitions.
<code>mymalloc.c</code>	Used by the low-level crypto-code.
<code>mymalloc.h</code>	Memory pool definitions used by the crypto library.
<code>perf_tools.c</code>	Used for profiling (not available on UltraSPARC T2).
<code>perf_tools.h</code>	Used for profiling (not available on UltraSPARC T2).
<code>rx.c</code>	Packet receive code which uses Ethernet API.

TABLE 11-10 Source Files (Continued)

Source File	Description
tx.c	Packet xmit code which uses Ethernet API encryption and hashing algorithms.
user_common.c	Contains the callback functions used by the Sun Netra DPS Ethernet APIs.
user_common.h	Contains fast queue definitions and function prototypes.
util.c	Contains IPsec utility functions.

TABLE 11-11 lists the IPsec library files.

TABLE 11-11 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.
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 11-12 lists the crypto library files.

TABLE 11-12 Crypto Library Files

Crypto Library File	Description
crypt_consts.h	Contains various crypto constants.
ndpscript.c	Contains crypto API implementations.
ndpscript.h	Contains data structures and function prototypes.
ndpscript_impl.h	Contains crypto context structure.

Reference Application Configurations

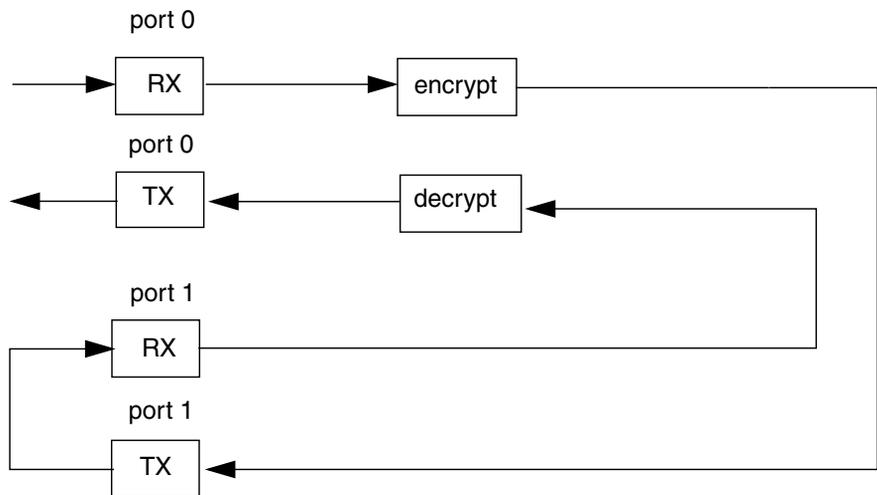
IPSec and crypto have five reference application configurations:

- “IP with Encryption and Decryption” on page 273
- “IPSec Gateway on Quad GE” on page 274
- “IPSec Gateway on NIU 10-Gbps Interface (One Instance)” on page 275
- “IPSec Gateway on NIU 10-Gbps Interface (Up to Four Instances)” on page 277
- “Multiple Instances (Up to Eight Instances) Back-to-Back Tunneling Configuration” on page 279

IP with Encryption and Decryption

This configuration can be used to evaluate the raw performance of the crypto engine. Two UltraSPARC T2 crypto engines are used: one for encryption and one for decryption.

FIGURE 11-9 IP With Encryption and Decryption Default Configuration



The following list includes the configuration requirements:

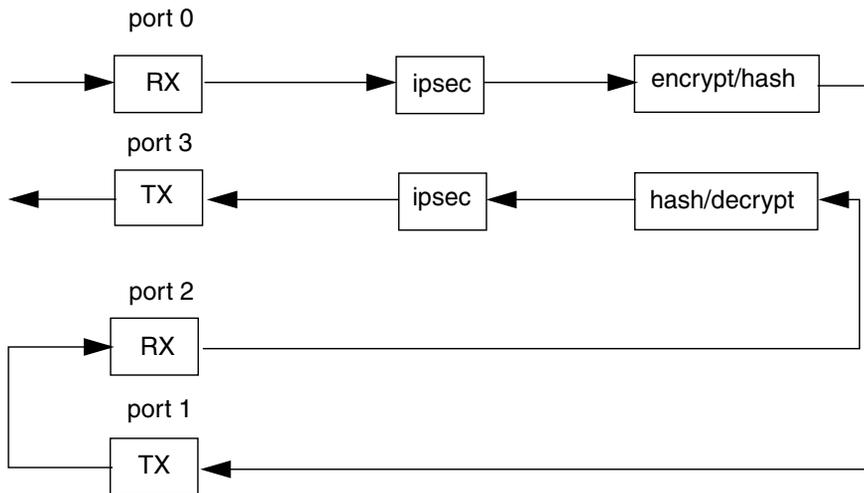
- Required equipment:
 - One UltraSPARC T2-based system
 - One traffic generator port

- Two NIU 10-Gbps Ethernet ports (two XAUI cards)
- One pair of straight connect copper cable, one cross-over copper cable
- Build method:
 - `./build cmt2 ipcrypto`
- Traffic generator configuration:
 - Frame Data – Select EthernetII, IPv4 + UDP/IP
 - DA MAC – MAC ID of port 0 (the recipient of plaintext)
 - 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 = 0x55
- Static data (`sa_init_static_data.c`) configuration (use default)

IPSec Gateway on Quad GE

This configuration implements one traffic flow on the PCIE Quad Gigabit Ethernet card.

FIGURE 11-10 IPSec Gateway on Quad GE Default Configuration



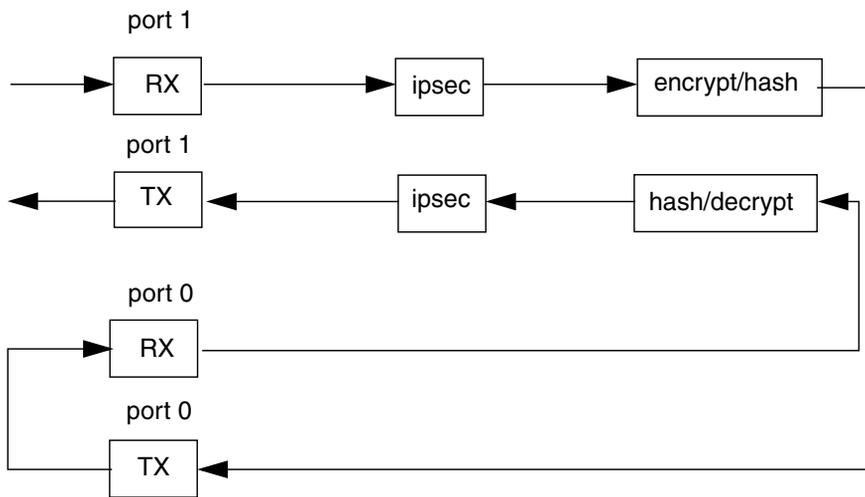
The following list includes the configuration requirements:

- Required equipment:
 - One UltraSPARC T2-based system
 - One Traffic Generator port
 - One PCIE Quad Gigabit Ethernet card
 - One pair of straight connect copper cable, one cross-over copper cable
- Build method:
 - `./build cmt2 qgc`
- Traffic generator configuration:
 - Frame Data – Select EthernetII, IPv4 + UDP/IP
 - DA MAC – MAC ID of port 0 shown in the above diagram
 - 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 = 0x55
- Static data (`sa_init_static_data.c`) configuration:
 - Must specify Remote Gateway MAC ID (port 2) in the MAC ID entry of `sa_outb1`.

IPSec Gateway on NIU 10-Gbps Interface (One Instance)

This configuration runs one instance of IPSec gateway application on the NIU 10-Gbps Ethernet interface. Two UltraSPARC T2 crypto engines are used: one for encrypt-hash and one for hash-decrypt. This configuration is not yet supported on the Sun Netra CP3260 platform.

FIGURE 11-11 IPSec Gateway on NIU 10-Gbps Interface (One Instance) Default Configuration



The following list includes the configuration requirements:

- Required equipment:
 - One UltraSPARC T2-based system
 - One traffic generator port
 - One PCIE 10-Gbps Ethernet card
 - One pair of straight connect copper cable and one cross-over copper cable
- Build method:
 - For crypto only:

```
./build cmt2 10g_niu -hash FLOW_POLICY
```
 - For crypto and authentication:

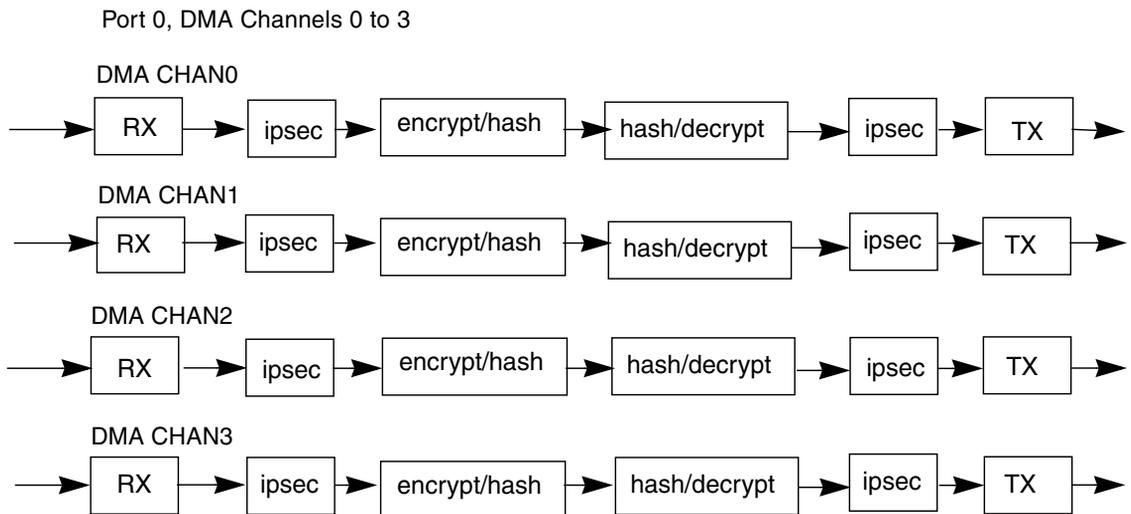
```
./build cmt2 10g_niu auth -hash FLOW_POLICY
```
 - Policy TCAM_CLASSIFY is recommended for both configurations.
- Traffic generator configuration:
 - Frame Data – select EthernetII, IPv4 + UDP/IP
 - DA MAC – MAC ID of port 1
 - IPv4 – SA=69.235.4.0 DA=69.235.4.1

- 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 = 0x55
- Static data (sa_init_static_data.c) configuration:
 - Must specify remote gateway MAC ID (port 0) in the MAC ID entry of sa_outb1.

IPSec Gateway on NIU 10-Gbps Interface (Up to Four Instances)

This configuration implements multiple instances of IPSEC gateway application on the NIU interface through internal loopback. Eight UltraSPARC T2 crypto engines are used: four to perform encrypt-hash and four to perform decrypt-hash.

FIGURE 11-12 IPSec Gateway on NIU 10-Gbps Interface (Up to Four Instances) Default Configuration



The following list includes the configuration requirements:

- Required equipment:
 - One UltraSPARC T2-based system
 - One traffic generator port
 - One NIU 10-Gbps Ethernet port (one XAUI card)
 - One straight connect fiber cable

- Build method:
 - For crypto only:

```
./build cmt2 niu_multi -hash FLOW_POLICY
```
 - For crypto and authentication:

```
./build cmt2 niu_multi auth -hash FLOW_POLICY
```

Note – To build for running on Sun Netra ATCA CP3260 systems, `HASH_POLICY` options are limited to the following policies: `IP_ADDR`, `IP_DA`, and `IP_SA`.

- Traffic generator configuration:
 - Frame data – Select EthernetII, IPv4 + UDP/IP
 - DA MAC – MAC ID of port 0
 - IPv4 – If *flow_policy* is *IP-address* (default), then:

```
SA=69.235.4.0
```

```
DA=69.235.0.0 ~ 69.235.255.255 (continue increment by 1)
```

If `FLOW_POLICY` is `TCAM_CLASSIFY`, then:

```
SA=69.235.4.0
```

```
DA=69.235.4.1 ~ 69.235.4.4 (increment by 1 and repeat every 4 counts)
```
 - 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 = `0x55`

Note – This setting of the traffic generator applies to the Sun SPARC Enterprise T5120 and T5220 systems. For Sun Netra ATCA CP3260 systems, see [“Flow Policy for Spreading Traffic to Multiple DMA Channels” on page 282](#).

Note – To build for Sun Netra CP3260, in `src/libs/ipsec/sa_init_static_data.c`, the `sa_outb1 remote_gw_mac` must be set to the port address of the outgoing Ethernet port.

- Static data (`sa_init_static_data.c`) configuration (use default)

Note – In the application configuration file (for example, `ipsecgw_niu_config.c`), if `port0` is used, no action is required. If `port1` is used, add: `..., OPEN_OPEN, NXGE_10G_START_PORT+1, ...`

Multiple Instances (Up to Eight Instances) Back-to-Back Tunneling Configuration

This configuration implements multiple instances of the IPsec gateway application on the NIU interfaces through back-to-back between two systems.

FIGURE 11-13 Default Configuration for System1 (Tunnel in)

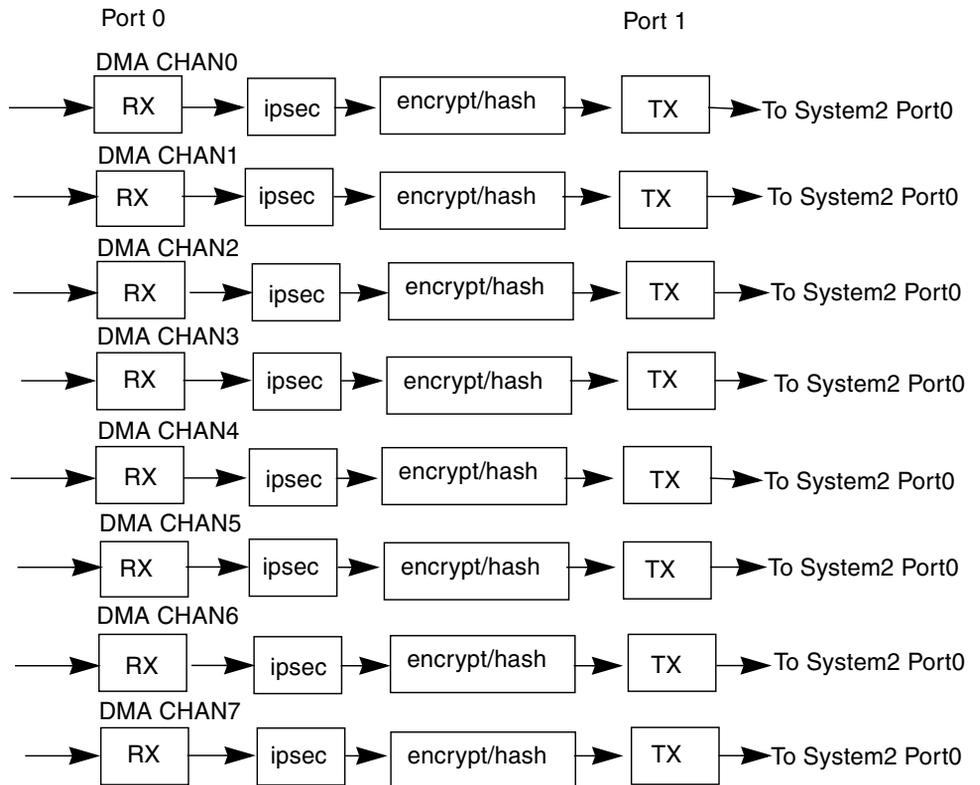
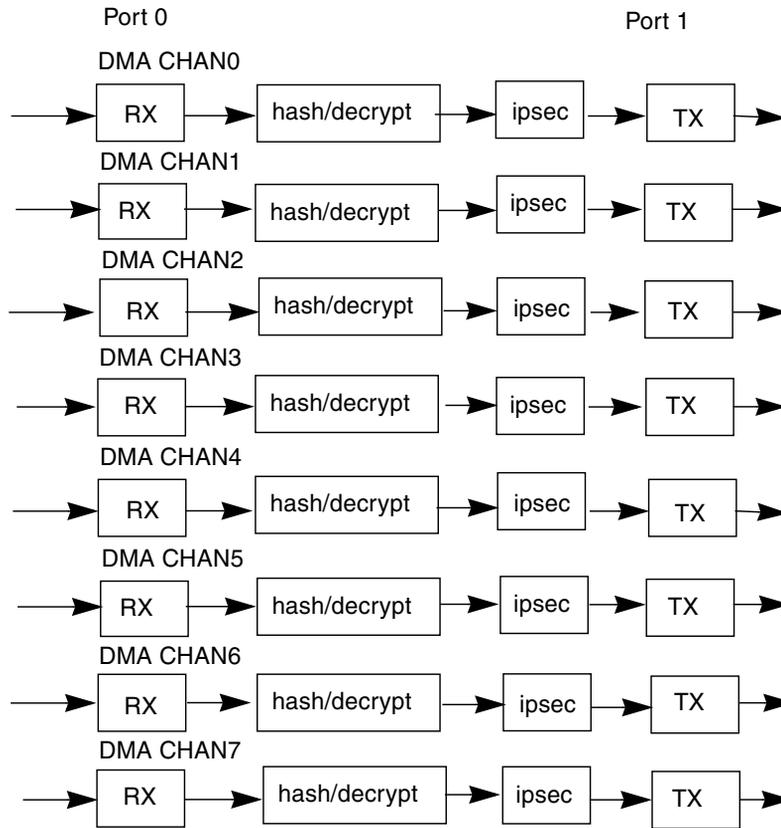


FIGURE 11-14 Default Configuration for System1 (Tunnel Out)



The following list includes the configuration requirements:

- Required equipment:
 - Two UltraSPARC T2-based systems
 - Two traffic generator ports
 - Four NIU 10-Gbps Ethernet ports (four XAUI cards, two for each system)
 - Two pair of straight connect fiber cables and one pair of cross-over fiber cable

- Build method

Two different binaries are required to run the back-to-back tunneling configuration. The following shows the two different methods generating the binaries for the corresponding system.

- System1

For crypto only:

```
./build cmt2 niu_tunnel_in -hash FLOW_POLICY
```

For crypto and authentication:

```
./build cmt2 niu_tunnel_in auth -hash FLOW_POLICY
```

- System2

For crypto only:

```
./build cmt2 niu_tunnel_out -hash TCAM_CLASSIFY
```

For crypto and authentication:

```
./build cmt2 niu_tunnel_out auth -hash TCAM_CLASSIFY
```

Note – Although other hash policies may still be used to generate binary for System2, traffic might not spread evenly on the System2 Rx input. TCAM_CLASSIFY policy will guarantee that traffic will spread evenly among the 8 DMA channels for this particular configuration.

- Traffic generator configuration:

- Frame data – Select EthernetII, IPv4 + UDP/IP

- DA MAC – MAC ID of System1 port0 shown in the diagram in [“Default Configuration for System1 \(Tunnel in\)”](#) on page 279

- IPv4

If FLOW_POLICY is IP_ADDR (default), then:

SA=69.235.4.0

DA=69.235.0.0 ~ 69.235.255.255 (continue increment by 1)

If FLOW_POLICY is TCAM_CLASSIFY, then:

SA=69.235.4.0

DA=69.235.4.1 ~ 69.235.4.8 (increment by 1 and repeat every 8 counts)

- 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 = 0x55

- Static data (`sa_init_static_data.c`) configuration:
 - Must specify remote gateway MAC ID (System2 `port0`) in the MAC ID entry of `sa_outb1`.

Note – In the application configuration file (for example, `ipsecgw_niu_config.c`), if `port0` is used, no action is required. If `port1` is used, add: `..., OPEN_OPEN, NXGE_10G_START_PORT+1, ...`

Flow Policy for Spreading Traffic to Multiple DMA Channels

The user can specify a policy for spreading traffic into multiple DMA flows by hardware hashing or by hardware TCAM lookup (classification). See [TABLE 11-2](#) for flow policy options.

▼ To Enable a Flow Policy

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

```
FLOW_POLICY=policy
```

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

For example, to enable hash on an IP destination and source address, run the build script with the following arguments:

```
% ./build cmt2 niu_multi -hash FLOW_POLICY=HASH_IP_ADDR
```

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

If none of the policies in [TABLE 11-2](#) are specified do not specify the `FLOW_POLICY` in the above `gmake` line. For example, if `#FLOW_POLICY=HASH_IP_ADDR`, a default policy will be given. When the default policy is used, all level (L2, L3, and L4) header fields are used for spreading traffic.

Traffic Generator Reference Application

This section explains how to compile Sun Netra DPS traffic generator tool (`ntgen`), how to use the tool, and the options provided by this tool.

The traffic generator (`ntgen`) is a tool that allows the generation of packets that are encapsulated in Ethernet. The Ethernet header might or might not have VLAN tags, but only Ethernet headers that use type encapsulation are supported. The `ntgen` tool provides options to modify the Ethernet header fields for all packet types. The tool also provides options to modify header fields of IPv4, UDP and GRE packets. The `ntgen` tool is capable of generating packets that have fixed or random sizes.

The traffic generator operates only with logical domains enabled. The user interface application runs in the Oracle VM Server for SPARC software and the `ntgen` tool runs in the Sun Netra DPS domain.

The user interface application provides a template packet to `ntgen` with user-provided options for modifications. The traffic generator creates new packets using the template packet, applies the modifications specified by the user options, and transmits the packets. The template packets are read by the user interface application from a snoop capture file (see the `templates/` directory in the `ntgen` application directory).

Note the following requirements:

- `tnsmctl -P -v` is required to start the traffic generator on systems that use NIU.
- The user interface application *must* be run as superuser in the Oracle Solaris OS logical domain.
- On Sun SPARC Enterprise T5120 and T5220 systems, 4-Gbytes of memory are required.

Using the User Interface

This section contains instructions for using the user interface.

▼ To Start the `ntgen` User Interface

The `ntgen` control plane application is represented by the binary `ntgen`.

● **Type:**

```
% ./ntgen
```

Usage

```
./ntgen [options ...] filename
```

See [TABLE 11-13](#) for the list of options.

Parameter

- *filename* – Snoop file

See “[ntgen Parameter Description](#)” on [page 294](#) for further descriptions and examples.

ntgen Option Descriptions

[TABLE 11-13](#) lists the options for the `ntgen` control plane application. See `-I` for further descriptions and examples.

TABLE 11-13 Traffic Generator Control Plane Application Options

Option	Description
-h	Prints this message.
-D	Sets destination MAC address.
-S	Sets source MAC address.
-A	Sets source and destination IPv4 addresses.
-P	Sets payload size.
-p	Sets UDP source and destination ports.
-V	Sets VLAN ID range.
-k	Sets GRE key range.
-iD	Destination MAC address increment mask.
-iS	Increments source IP address, destination IP address host or network.
-iA	Increments SIP or DIPs host or network.
-ip	Increments UDP source or destination port.

TABLE 11-13 Traffic Generator Control Plane Application Options (*Continued*)

Option	Description
-iV	Increments or decrements VLAN ID.
-ik	Increments or decrements GRE key.
-dD	Destination MAC address decrement mask.
-dS	Source MAC address decrement mask.
-dA	Decrements source IP address, destination IP address host, or network.
-dp	Decrements UDP source or destination port.
-c	Continuous generation.
-n	Generate number of packets specified.
-I	Ingress or receive only mode.
-R	Generates random packet sizes.
-N	Sets source or destination IPv6 addresses.
-iN	Increments IPv6 addresses.
-dN	Decrements IPv6 addresses.

Option Descriptions

The following options are supported:

■ -h

Prints displayed message.

Example:

```
ntgen -h
```

■ -D *xx:xx:xx:xx:xx:xx*

Changes the destination MAC address of a packet. Specify the destination MAC address in the colon format.

Example:

```
ntgen -D aa:bb:cc:dd:ee:00 filename
```

■ -S *xx:xx:xx:xx:xx:xx*

Changes the source MAC address of a packet. Specify the destination MAC address in the colon format.

Example:

```
ntgen -S 11:22:33:44:55:00 filename
```

- `-A xx.xx.xx.xx, yy.yy.yy.yy`

Changes the source and destination IP addresses in the packet. Specify the IP addresses in the dotted decimal notation.

The first argument in the option is the source IP address. The second argument in the option is the destination IP address. You can use an asterisk (*) for either the source IP address or the destination IP address to imply that no change needs to occur for that parameter.

Examples:

- `ntgen -A 192.168.1.1,192.168.2.1 filename`

The source IP address is changed to 192.168.1.1 and the destination IP address is changed to 192.168.2.1.

- `ntgen -A 192.168.1.10, * filename`

The source IP is changed to 192.168.1.10 and the destination IP is unchanged. The destination IP is retained as it is in the template packet.

- `-p xx,yy`

Changes the UDP source port and destination port numbers.

The first argument is the UDP source port number and the second argument is the UDP destination port number. You can use an asterisk (*) for either the source port or the destination port to imply that no change needs to occur to that parameter. In that case, the value present in the template packet is retained.

Examples:

- `ntgen -p 1111,2222 filename`

The source port number is changed to 1111 and the destination port number is changed to 2222.

- `ntgen -p *,2222 filename`

The source port number remains unchanged from its value in the template packet. The destination port number is changed to 2222 in the packets generated.

- `-P x`

Increases the UDP payload size. The value specified must be between 1 and 65536. The value denotes the number of bytes that need to be added to the payload.

Example:

```
ntgen -P 1024 filename
```

The UDP packet payload size is incremented by 1024 bytes (that is, the new payload size is the original size plus 1024 bytes).

- `-V VLAN-ID-start-value, VLAN-ID-end-value`

Creates Ethernet frames with 802.1Q VLAN tags in the traffic packets. The Ethernet header of each packet that is generated is appended with a VLAN tag. The VLAN Identifier (VLAN ID) in the VLAN tags of the outgoing frames vary between

VLAN-ID-start-value and *VLAN-ID-end-value*. Two methods of VLAN ID variation are provided through the `-iV` option. When the `-iV` option is used with an argument of 1, the VLAN IDs are incremented. When the `-iV` option is used with an argument of 0, the VLAN IDs are decremented. Refer to “`-iV 1/0`” on [page 289](#) for further details and examples.

Examples:

- `ntgen -V 100,4094 filename`

Ethernet frames with VLAN tags are generated where the VLAN IDs in the VLAN tags of all frames are set to 100 (that is, the VLAN ID start value). The VLAN IDs do not vary in this example since the `-iV` option is not used.

- `ntgen -V 1,4094 -iV 1 filename`

Ethernet frames with VLAN tags are generated where the VLAN IDs in the VLAN tags vary from 1 to 4094 in an incremental fashion.

- `ntgen -V 1,4094 -iV 0 filename`

Ethernet frames with VLAN tags are generated where the VLAN IDs in the VLAN tags vary from 1 to 4094 in a decremental fashion.

- `-k GRE-key-start-value, GRE-key-end-value`

Changes the GRE key of GRE encapsulated packets in the range specified. The GRE key field in the generated packets will vary between the *GRE-key-start* value and the *GRE-key-end* value. Two methods of the GRE key variation are provided with the

`-ik` option. When the `-ik` option is used with value 1, GRE keys are incremented. When the `-ik` option is used with value 0, the GRE keys are decremented. Refer to “`-ik 1/0`” on [page 290](#) for further details.

Examples:

- `ntgen -k 1,1000 -ik 1 filename`

GRE keys in the generated traffic start from 1 and increase to 1000.

- `ntgen -k 1,1000 -ik 0 filename`

GRE keys in the generated traffic start from 1000 and decrease to 1.

Note – Only the `file_gre_novlan` template file can be used with this option.

- `-iD xx:xx:xx:xx:xx:xx`

Increments the bytes in the destination MAC address that is specified using the `-D` option. The option is followed by the byte mask. `ff` increments the byte. `0` does not increment the byte.

Examples:

- `ntgen -D aa:bb:cc:dd:ee:00 -iD 00:00:00:00:00:ff filename`

Only byte 0 is incremented.

- `ntgen -D aa:bb:cc:dd:ee:00 -iD ff:ff:ff:ff:ff:ff filename`

All bytes are incremented.

- `-iS xx:xx:xx:xx:xx:xx`

Increments the bytes in the source MAC address that is specified using the `-S` option. The option is followed by the byte mask. `ff` increments the byte. `0` does not increment the byte.

Examples:

- `ntgen -S aa:bb:cc:dd:ee:00 -iS 00:00:00:00:00:ff filename`

Only byte 0 is incremented.

- `ntgen -S aa:bb:cc:dd:ee:00 -iS ff:ff:ff:ff:ff:ff filename`

All bytes are incremented.

- `-iA host/net/pfx/*, host/net/pfx/*`

Increments the source IP address and destination IP address (that were specified using the `-A` option) based on the IP address class or on a prefix. The first argument corresponds to the source IP address of a packet. The second argument corresponds to the destination IP address of a packet.

To perform a class-based increment, specify the `host` or `net` arguments with the `-iA` option. `ntgen` determines the class of IP address (class A, class B, class C, or class D) that is specified with the `-A` option. From the class, the option determines the length of the host part and the network part of the IP address. Based on the parameters passed through the `-iA` option, either the host part or the network part of the IP address is incremented. If an asterisk (*) is passed, then the IP address is not incremented.

The string `net` denotes that the network portion of the corresponding IP address must be incremented. The string `host` denotes that the host part of the IP address must be incremented.

To perform a prefix-based increment, provide the prefix length argument with the `-iA` option. Provide a prefix length for each IP address (source and destination) as arguments to the `-iA` option. These values are used to calculate the portion of the IP address that must be incremented. If an asterisk (*) is passed, then the corresponding IP address is not incremented.

Note – Currently, only 16 bits of an IP address can be incremented using either class-based or prefix-based methods.

Examples:

- `ntgen -A 192.168.1.1,192.168.2.1 -iA net,host filename`

The network portion of the source IP address and the host portion of the destination IP address are incremented.

- `ntgen -A 192.168.1.1,192.168.2.1 -iA host,host filename`

The host portion of both the source and destination IP addresses are incremented.

- `ntgen -A 192.168.10.10,192.168.10.20 -iA host,* filename`

The host portion of the source IP address is incremented. The destination IP address is not incremented.

- `ntgen -A 10.10.10.10,10.10.10.11 -iA 10,12 filename`

The source IP address is incremented with a prefix length of 10. The destination IP address is incremented with a prefix length of 12.

- `ntgen -A 10.10.10.10,10.10.10.11 -iA 10,* filename`

The source IP address is incremented with a prefix length of 10. The destination IP address is not incremented.

- `-ip 0/1, 0/1`

Increments the UDP source port and destination port numbers. The first argument corresponds to the UDP source port. The second argument corresponds to the UDP destination port. 0 does not increment the port numbers. 1 increments the port numbers.

Examples:

- `ntgen -p 1111,2222 -ip 0,1 filename`

The source port is not incremented, but the destination port is incremented.

- `ntgen -p 1111,2222 -ip 1,1 filename`

Both the source and destination ports are incremented.

- `-iV 1/0`

Increments or decrements VLAN IDs in the VLAN tags of the generated Ethernet frames. 1 denotes an increment operation. 0 denotes a decrement operations.

The VLAN IDs are provided by the user using the `-v` option. For the increment operation, the first VLAN ID is the *VLAN-ID-start-value* that is provided in the `-v` option. The VLAN ID is incremented for each subsequent frame until the *VLAN-ID-end-value* provided with the `-v` option is reached. Then the VLAN ID returns to the *VLAN-ID-start-value* and the sequence is repeated.

For the decrement operation, the first VLAN ID is the *VLAN-ID-end-value* that is provided with the `-V` option. The VLAN ID is decremented for each subsequent frame until *VLAN-ID-start-value* provided with the `-V` option is reached. Then the VLAN ID returns to the *VLAN-ID-start-value* and the sequence is repeated.

Examples:

- `ntgen -V 100,200 -iV 1 filename`

Ethernet frames are appended with a VLAN tag that contain VLAN ID in the range 100 to 200. Starting at 100, the VLAN IDs are incremented for each frame starting until 200.

- `ntgen -V 100,200 -iV 0 filename`

Ethernet frames are appended with a VLAN tag that contain VLAN ID in the range 200 to 100. Starting at 200, the VLAN IDs are decremented for each frame starting until 100.

- `-ik 1/0`

Increments or decrements GRE keys in the GRE header of the generated GRE packets. An argument of 1 denotes an increment operation. 0 denotes a decrement operation. Provide the GRE keys using the `-k` option.

For the increment operation, the first GRE key is the *GRE-key-start-value* provided with the `-k` option. The GRE key is incremented for each subsequent packet until the *GRE-key-end-value* provided with the `-k` option is reached. The GRE Key then returns to the *GRE-key-start-value* and the sequence is repeated.

For the decrement operation, the first GRE key is the *GRE-key-end-value* provided with the `-k` option. The GRE key is decremented for each subsequent packet until the *GRE-key-start-value* provided with the `-k` option is reached. The GRE key then returns to the *GRE-key-end-value* and the sequence is repeated.

Examples:

- `ntgen -k 1,100 -ik 1 filename`

GRE packets with key values in the range 1 to 100 are generated. Starting at 1, the key value is incremented for each packet until 100.

- `ntgen -k 1,100 -ik 0 filename`

GRE packets with key values in the range 100 to 1 are generated. Starting at 100, the key value is decremented for each packet until 1.

- `-dD xx:xx:xx:xx:xx:xx`

Decrements the bytes in the destination MAC address that is specified using the `-D` option. The option is followed by a byte mask. `ff` decrements the byte. `00` does not decrement the byte.

Examples:

- `ntgen -D aa:bb:cc:dd:ee:00 -dD 00:00:00:00:00:00 filename`

Only byte 0 of the MAC address is decremented.

- `ntgen -D aa:bb:cc:dd:ee:00 -dD ff:ff:ff:ff:ff:ff filename`

All bytes of the MAC address are decremented.

- `-dS xx:xx:xx:xx:xx:xx`

Decrements the bytes in the source MAC address that is specified using the `-S` option. The option is followed by a byte mask. `ff` decrements the byte. `00` does not decrement the byte.

Examples:

- `ntgen -S aa:bb:cc:dd:ee:00 -dS 00:00:00:00:00:00 filename`

Only byte 0 of the MAC address is decremented.

- `ntgen -S aa:bb:cc:dd:ee:00 -dS ff:ff:ff:ff:ff:ff filename`

All bytes of the MAC address are decremented.

- `-dA host/net/pfx/*, host/net/pfx/*`

Decrements the source IP address and destination IP address (that were specified using the `-A` option) based on the IP address class or on a prefix. The first argument corresponds to the source IP address of a packet. The second argument denotes the destination IP address of a packet.

To perform a class-based decrement, specify the `host` or `net` arguments with the `-dA` option. `ntgen` determines the class of the IP address (class A, class B, class C or class D) that is specified using the `-A` option. From the class, the option determines the length of the host part and the network part of the IP address. Based on the parameters passed through the `-iA` option, either the host part or the network part of the IP address is decremented. If an asterisk (*) is passed, then the IP address is not decremented.

The string `net` denotes that the network portion of the corresponding IP address must be decremented. The string `host` denotes that the host part of the corresponding IP address must be decremented.

To perform a prefix-based decrement, provide the prefix length argument with the `-dA` option. Provide a prefix length for each IP address (source and destination) as arguments to the `-dA` option. These values are used to calculate the portion of the IP address that needs to be decremented. If an asterisk (*) is passed, then the corresponding IP address is not decremented.

Note – Currently, only 16 bits of an IP address can be decremented using either class-based or prefix-based methods.

Examples:

- `ntgen -A 192.168.1.1,192.168.2.1 -dA net,host filename`

The network portion of the source IP address and the host portion of the destination IP address are decremented.

- `ntgen -A 192.168.1.1,192.168.2.1 -dA host,host filename`

The host portion of both the source and destination IP addresses are decremented.

- `ntgen -A 192.168.10.10,192.168.10.20 -iA host,* filename`

The host portion of the source IP address is decremented. The destination IP address is not decremented.

- `ntgen -A 10.10.10.10,10.10.10.11 -dA 10,12 filename`

The source IP address is decremented using a prefix length of 10. The destination IP address is decremented using a prefix length of 12.

- `ntgen -A 10.10.10.10,10.10.10.11 -dA 10,* filename`

The source IP address is decremented using a prefix length of 10. The destination IP address is not decremented.

- `-dp 0/1,0/1`

Decrements the UDP source port and destination port numbers. The first argument corresponds to the UDP source port. The second argument corresponds to the UDP destination port. 0 does not decrement. 1 decrements the port numbers.

Examples:

- `ntgen -p 1111,2222 -dp 0,1 filename`

The UDP source port is not decremented, but the destination port is decremented.

- `ntgen -p 1111,2222 -dp 1,1 filename`

Both the source and destination ports are decremented.

- `-c`

Generates packets continuously.

Examples:

- `ntgen -c filename`

The packets in the file are generated continuously without applying any modifications.

- `ntgen -D aa:bb:cc:dd:ee:00 -S 11:22:33:44:55:00
-A 192.168.10.10,192.168.10.11 -p 9999,8888
-iD ff:ff:ff:ff:ff:ff -iS ff:ff:ff:ff:ff:ff -iA host,host
-ip 1,1 -c filename`

All the modifications pertaining to the options specified are applied and the packets are generated continuously.

- *-n number of packets*

Specifies the number of packets that need to be generated.

Example:

- `ntgen -n 1000000 filename`

In this example, a million packets are generated.

- *-I*

Runs the traffic generator in ingress mode. In this mode the traffic generator only receives packets, displays statistics about the ingress traffic, and discards the received traffic. This option takes no arguments.

- *-R*

When used with a UDP/IPv4 template packet or a GRE template packet with a UDP/IPv4 payload, this option generates random packet sizes. The resulting frame sizes vary between 64 bytes (or 68 bytes with VLAN tag) and 1518 bytes (1522 bytes with VLAN tag).

If other packet types are used, this option has no effect.

- *-N*

Changes the source and destination IPv6 addresses in a packet. The IP addresses are specified in a colon separated format, `x:x:x:x:x:x:x:x`. In this format, each `x` is a hexadecimal 16-bit value of the address part. In all, eight such values are present.

The first argument in the option is the source IPv6 address and the second argument is the destination IPv6 address. You can use an asterisk (*) to specify either the source or the destination address to imply that no change needs to be done for that parameter.

Examples:

- `ntgen -N 1:1:1:1:1:1:1:1,2:2:2:2:2:2:2:2 -n 10 filename`

The source IPv6 address is set to `1:1:1:1:1:1:1:1` and the destination IPv6 address is set to `2:2:2:2:2:2:2:2`.

- `ntgen -N 1:1:1:1:1:1:1:1,* -n 10 filename`

The source IPv6 address is set to `1:1:1:1:1:1:1:1`. The destination IPv6 address is not changed and is retained since it is in the template packet.

- *-iN*

Increments the IPv6 addresses in the packet generated. The user provides a mask in the option for each address that needs to be incremented. The mask is provided in a colon separated format, `x:x:x:x:x:x:x:x`. This format consists of eight 16-

bit parts similar to the IPv6 address. Each *x* in the mask is either the hexadecimal value `0x0000` or `0xffff` and maps to the corresponding 16-bit value in the IPv6 address supplied with the `-N` option.

A value of `0x0000` in the mask implies that the corresponding 16-bit IPv6 address part is not incremented. A value of `0xffff` in the mask implies that the corresponding 16-bit IPv6 address part is incremented.

Examples:

- `ntgen -N a:b:c:d:e:f:0:1,* -iN 0000:0000:0000:0000:0000:0000:0000:ffff,* -n 10 filename`

Only the first 16-bit part of the source IPv6 address is incremented. The remaining parts are unchanged.

- `ntgen -N *,a:b:c:d:e:f:0:1 -iN *,ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff -n 10 filename`

All parts of the IPv6 destination address are incremented.

- `-dN`

Decrements the IPv6 addresses in packets generated. The user provides a mask in the option for each address that needs to be decremented. The mask is provided in a colon separated format, `x:x:x:x:x:x:x:x`. This format consists of eight 16-bit parts similar to the IPv6 address. Each *x* in the mask is either the hexadecimal value `0x0000` or `0xffff` and maps to the corresponding 16-bit value in the IPv6 address supplied with the `-N` option.

A value of `0x0000` in the mask implies that the corresponding 16-bit IPv6 address part is not decremented. A value of `0xffff` in the mask implies that the corresponding 16-bit IPv6 address part is decremented.

Examples:

- `ntgen -N a:b:c:d:e:f:0:1,* -dN 0000:0000:0000:0000:0000:0000:0000:ffff,* -n 10 filename`

Only the first 16-bit part of the source IPv6 address is decremented. The remaining parts are unchanged.

- `ntgen -N *,a:b:c:d:e:f:0:1 -iN *,ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff -n 10 filename`

All parts of the IPv6 destination address are decremented.

ntgen Parameter Description

The snoop input file option, *filename*, specifies a snoop file that contains the template packet to be used for creating the traffic packets. You can use one of the files in the `templates/` directory in the `ntgen` application directory. These files contain packets whose fields can be modified with the `ntgen` tool options. You can analyze

these snoop files by using the snoop program in the Oracle Solaris OS. Use the ntgen options to modify the protocol header files. A detailed explanation of the template snoop files is provided in [“Template Files” on page 296](#).

Note – Only the first packet from the snoop command is used by ntgen for generating traffic.

Note – The -A, -iA and -dA options are applied only to the delivery IPv4 header (outer IPv4 header) of a GRE packet.

Notes

The increment options (-iD, -iS, -iA and -ip) and the decrement options (-dD, -dS, -dA and -dp) have effect only when the values that need to be incremented or decremented are also being modified.

For example, the following commands have no effect:

- `ntgen -iD ff:ff:ff:ff:ff:ff filename`

This command has no effect. The destination MAC address will not be incremented.

- `ntgen -iA host,host filename`

This command has no effect. The source and destination IP addresses will not be incremented.

- `ntgen -ip 1,1 filename`

This command has no effect. The port numbers will not be incremented.

The following commands *will* have effect:

- `ntgen -D aa:bb:cc:dd:ee:00 -iD ff:ff:ff:ff:ff:ff filename`

This command increments the destination MAC address after changing it to aa:bb:cc:dd:ee:00. Because -D option is being used, the -iD option takes effect.

- `ntgen -A 192.168.1.1,192.168.1.2 -iA host,host filename`

This command increments the source and destination IP addresses. Because the -A option is being used, the -iA option takes effect.

- `ntgen -p 1234,6789 -ip 1,1 filename`

This command increments the source and destination UDP ports. Because the -p option is being used, the -ip option takes effect.

Traffic Generator Output

TABLE 11-14 shows an example of the traffic generator output.

TABLE 11-14 Traffic Generator Output Example

Port,Chan	Tx Rate (pps)	Tx Rate (mbps)	Rx Rate (pps)	Rx Rate (mbps)
0, 0	947550.5506	485.1459	32224.4898	386.6939
1, 0	947550.5506	485.1459	32224.4898	386.6939
2, 0	947550.5506	485.1459	32224.4898	386.6939
3, 0	947550.5506	485.1459	32224.4898	386.6939

TABLE 11-15 describes the traffic generator output.

TABLE 11-15 Traffic Generator Output Description

Column	Description
Port,Chan	Port is the port number and Chan is the channel number for which the statistics are displayed. In the example output shown in TABLE 11-14 for NxGE QGC, Port varies from 0 to 3 and Chan is 0 for all ports.
Tx Rate (pps)	Transmission rate in packets per second.
Tx Rate (mbps)	Transmission rate in megabits per second.
Rx Rate (pps)	Receive rate in packets per second.
Rx Rate (mbps)	Receive rate in megabits per second.

Template Files

The following template files are provided with the application to be used with `ntgen`.

- `file_64B_novlan`
Snoop file that contains a single 64-byte Ethernet frame that has no VLAN tag. This file has a UDP/IPv4 payload.
- `file_256B_novlan`
Snoop file that contains a single 256 bytes Ethernet frame that has no VLAN tag. The file has a UDP/IPv4 payload.
- `file_1514B_novlan`

Snoop file that contains a single 1514 bytes Ethernet frame that has no VLAN tag. This file has a UDP/IPv4 payload.

- `file_gre_novlan`

Snoop file that contains a GRE packet with an IPv4 as the delivery protocol and IPv4 as the payload protocol. The payload is a UDP datagram. The UDP datagram has a payload of 22 bytes. Both IPv4 headers have no IP options. GRE header consists of GRE key and GRE checksum values.

Using the Traffic Generator

This section describes configuring, starting, and stopping the `ntgen` tool.

Configuring Logical Domains for the Traffic Generator

[TABLE 11-16](#) shows the domain role in the configuration.

TABLE 11-16 Logical Domain Configuration

Domain	Operating System	Role
<code>primary</code>	Solaris	Owns one of the PCI buses and uses the physical disks and networking interfaces to provide virtual I/O to the Oracle Solaris OS guest domains.
<code>ldg1</code>	LWRTE	Owns the other PCI bus (<code>bus_b</code>) with its two network interfaces and runs an LWRTE application.
<code>ldg2</code>	Solaris	Runs control plane application (<code>ntgen</code>) and <code>add_drv tnsn</code> (SUNWndpsd package) and uses <code>ntgen</code> to control traffic generation.
<code>ldg3</code>	Solaris	Controls <code>lwrt</code> (global configuration channel) and <code>add_drv tnsn</code> (SUNWndpsd package) and uses <code>tnsmctl</code> to set up configuration.

[TABLE 11-17](#) shows the LDC channels configured.

TABLE 11-17 LDC Channels Configured

Server	Client
<code>ldg1 primary-gc</code>	<code>ldg3 tnsn-gc0</code>

TABLE 11-17 LDC Channels Configured (Continued)

Server	Client
ldg1 config-tnsm-ldg2	ldg2 config-tnsm0
ldg1 ldg2-vdpcs0	ldg2 vdpcc0
ldg1 ldg2-vdpcs1	ldg2 vdpcc1

These LDC channels can be added with the following Oracle VM Server for SPARC software manager commands:

```
ldm add-vdpcs primary-gc ldg1
ldm add-vdpcc tnsn-gc0 primary-gc ldg3
ldm add-vdpcs config-tnsm-ldg2 ldg1
ldm add-vdpcc config-tnsm0 config-tnsm-ldg2 ldg2

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

In the Oracle Solaris domains, you must add the `tnsm` driver.

▼ To Add the `tnsm` Driver

1. Install the `SUNWndpsd` package.
2. Install the driver:

```
add_drv tnsn
```

The `primary-gc` and `tnsm-gc0` combination is the global configuration channel. `LWRTE` accepts configuration messages on this channel.

The `config-tnsm-ldgx` and `config-tnsm0` combination is for setup messages between `LWRTE` and the control plane domain.

To find out what the LDC IDs are on both sides, use the following:

- For logical domains 1.0, use `ldm list-bindings`
- For logical domains 1.0.1, use `ldm list-bindings -e`

Example output from logical domain 1.0:

```
ldm list-bindings
In ldg1:
Vdpcs:  config-tnsm-ldg2
        [LDom ldg2, name: config-tnsm0]
        [LDC: 0x6]
In ldg2:
Vdpcc:  config-tnsm0      service: config-tnsm-ldg2 @ ldg1
        [LDC: 0x5]
```

Example output from logical domain 1.0.1:

```
ldm list-bindings -e
In ldg1:
VDPCS
  NAME
  config-tnsm-ldg2
  CLIENT
  config-tnsm0@ldg2
  LDC
  6
In ldg2:
VDPCC
  NAME
  config-tnsm0
  SERVICE
  config-tnsm-ldg2@ldg1
  LDC
  5
```

3. Pick a channel number to be used for the control IPC channel that uses this LDC channel (for example, 3).

4. Bring up the control channel with the following command:

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

Description of parameters:

- -S – Set up a channel.
- -C *n1* – Channel ID for new channel.
- -L *n2* – LDC ID local to *LWRTE*.
- -R *n3* – LDC ID remote to *LWRTE* (local to link partner logical domain).
- -F *n4* – Channel ID of the control channel between the two link partners. Because this command brings up the control channel, *n1* == *n4*.

In the previous `tnsmctl` command example:

- *n1* = 3 – Channel ID chosen for this configuration channel.

- $n2 = 6$ – LDC ID shown by `ldm list-bindings` for `config-tnsm-ldg2` in `ldg1`.
- $n3 = 5$ – LDC ID shown by `ldm list-bindings` for `config-tnsm0` in `ldg2`.
- $n4 = 3$ – Same channel ID as $n1$, because the config channel is being initialized.

5. Use control channel 3 to set up general purpose IPC channels between LWRTE and the Oracle Solaris OS.

For example, set up channel ID 4 for use by the `ntgen` to `ndpstgen` communication.

To do so, look up the LDC IDs on both ends.

Example output from logical domain 1.0:

```
ldg1:
Vdpcs:  ldg2-vdpcs0
        [LDom ldg2, name: vdpcc0]
        [LDC: 0x7]

ldg2:
Vdpcc:  vdpcc0  service: ldg2-vdpcs0 @ ldg1
        [LDC: 0x6]
```

Example output from logical domain 1.0.1:

```
ldg1:
VDPCS
  NAME
  ldg2-vdpcs0
  CLIENT          LDC
  vdpcc0@ldg2    7

ldg2:
VDPCC
  NAME          SERVICE          LDC
  vdpcc0 `     ldg2-vdpcs0@ldg1    6
```

6. Type the following in `ldg3`:

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

The `-C 4` parameter is the ID for the new channel. The `-F 3` has the channel set up before.

The global configuration channel between `ldg3` and `LWRTE` comes up automatically as soon as the application is started in `LWRTE` and the `tnsm` device driver is added in `ldg3`.

7. Build the `ntgen` utility in the Oracle Solaris OS subtree.
8. After the channel to be used is initialized using `tnsmctl` (must be channel ID 4 that is hard coded into the `ndpstgen` application), use `ntgen` to generate traffic (refer to the *NTGEN User's Manual*).

▼ To Prepare Building the `ntgen` Utility

1. Build the Sun Netra DPS image.
2. Build the `ntgen` user interface application (in the `src/solaris` subdirectory).

▼ To Set Up and Use Logical Domains for the Traffic Generator

1. Configure the primary domain.
2. Save the configuration (`ldm add-spconfig`) and reboot.
3. Configure the Sun Netra DPS domain (including the `vdpcs` services).
4. Configure the Oracle Solaris OS domains (including `vdpc` clients).
5. Bind the Sun Netra DPS domain (`ldg1`).
6. Bind the Oracle Solaris OS domains (`ldg2` and `ldg3`).
7. Start and boot all domains (can be in any order).
8. Install the `SUNWndpsd` package in the Oracle Solaris OS domains.
9. Load the `tnsm` driver in the Oracle Solaris OS domains (`add_drv tns`).
10. In the global configuration Oracle Solaris OS domain (`ldg3`), use `/opt/SUNWndpsd/bin/tnsmctl` to set up the control channel between the Sun Netra DPS domain (`ldg1`) and the control domain (`ldg2`).
11. In the global configuration Oracle Solaris OS domain (`ldg3`), use `/opt/SUNWndpsd/bin/tnsmctl` to set up the `ntgen` control channel (channel ID 4).
12. In the control domain (`ldg2`), use the `ntgen` utility to start traffic generation.

▼ To Start the Traffic Generation

- Use the `ntgen` binary tool.

For example:

```
% ./ntgen -c file_64B_novlan
```

▼ To Stop Traffic Generation

- Pressing Ctrl-C at any time.

▼ To Compile the Traffic Generator

1. Copy the `ntgen` reference application from the `/opt/SUNWndps/src/apps/ntgen` directory to a desired directory location
2. Run the build script in that location.

Build Script

TABLE 11-18 shows the traffic generator (`ntgen`) application build script.

TABLE 11-18 `ntgen` Application Build Script

Build Script	Usage
<code>./build</code> (See “Argument Descriptions” on page 302.)	Build <code>ntgen</code> application to run on an Ethernet interface.
	Build <code>ntgen</code> application to run on Sun QGC (quad 1-Gbps <code>nxge</code> Ethernet interface).
	Build <code>ntgen</code> application to run on Sun multithreaded 10-Gbps (dual 10 Gbps <code>nxge</code> Ethernet interface).
	Build <code>ntgen</code> application to run on NIU (dual 10-Gbps UltraSPARC T2 Ethernet interface) on a CMT2-based system.

Usage

```
./build cmt app [profiler] [2port]
```

Argument Descriptions

The build script supports the following optional arguments:

- *cmt*
Specifies whether to build the traffic generator application to run on the CMT1 platform or CMT2 platform.
 - *cmt1* – Build for CMT1 (UltraSPARC T1)
 - *cmt2* – Build for CMT2 (UltraSPARC T2)
- *app*
 - *4g* – Builds the traffic generator application to run on QGC (quad 1-Gbps nxge Ethernet interface).
 - *10g* – Builds the traffic generator application to run on 10-Gbps Ethernet (dual 10-Gbps nxge Ethernet interface).
 - *10g_niu* – Builds the traffic generator application to run on NIU (dual 10-Gbps UltraSPARC T2 Ethernet interface) on a CMT2 based system.
- [*profiler*]
Generates code with profiling enabled.
- [*2port*]
This is an optional argument to compile dual ports on the 10-Gbps Ethernet card or the UltraSPARC T2 network interface unit (NIU).
For example, to build for 10-Gbps Ethernet on the Sun Netra T2000 system, type:

```
% ./build cmt1 10g
```

In this example, the build script is used to build the traffic generator application to run on the 10-Gbps Ethernet. The *cmt* argument is specified as *cmt1* to build the application to run on the Sun Netra T2000 system that is an UltraSPARC T1-based system. The *app* argument is specified as *10g* to run on 10-Gbps Ethernet.

▼ To Run ndpstgen

1. On a tftpboot server, type:

```
% cp your-workspace/ntgen/code/ndpstgen/ndpstgen /tftpboot/ndpstgen
```

2. At the ok prompt on the target machine, type:

```
ok boot network-device: ,ndpstgen
```

Default Configurations

The following table shows the default system configuration.

TABLE 11-19 Default System Configuration

	NDPS Domain (strand IDs)	Statistics (strand ID)	Other Domains (strand IDs)
CMT1 logical domain	0 to 19	N/A	20 to 31
CMT2 logical domain	0 to 39	N/A	40 to 63

The main files that control the system configuration are:

- `ntgen/src/apps/config/tgen_swarch.c`
- `ntgen/src/apps/config/tgen_map.c`

The following table shows the default `ntgen` application configuration.

TABLE 11-20 Default `ntgen` Application Configuration

Applications Runs On	Number of Ports Used	Number of Channels per Port	Total Number of Q Instances	Total Number of Strands Used
4-Gbps PCE (nxge QGC)	4	1	4	12
10-Gbps PCIE (nxge 10-Gbps)	1	4	4	12
10-Gbps NIU (niu 10-Gbps)	1	8	8	40

The main files that control the application configurations are:

- `ntgen/src/apps/tgen_config.c`
- `ntgen/src/apps/tgen_config.h`

Interprocess Communication Reference Application

The IPC reference application showcases the programming interfaces of the IPC framework (see [“Interprocess Communication Software” on page 89](#) and the *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*).

The IPC reference application consists of the following three components:

- Sun Netra DPS application that receives and transmits test data messages.
- Oracle Solaris test utility that transmits and receives messages from user space.
- STREAMS module that intercepts network traffic from an interface to send it to the Sun Netra DPS domain, and transmits packets it receives through IPC on this network interface.

The application runs in an logical domain environment similar to the environment described in [“Example Environment for UltraSPARC T1 Based Servers”](#) on page 94 and [“Example Environment for UltraSPARC T2 Based Servers”](#) on page 98.

IPC Reference Application Content

The complete source code for the IPC reference application is in the `SUNWndps` package in the `/opt/SUNWndps/src/apps/ipc_test` directory.

The source code files include the following:

- Build script and makefiles for the application:
 - `Makefile`
 - `build`
- Common header file describing the communications protocol used between the components:
 - `src/common/include/ipctest.h`
- System configuration for the Sun Netra DPS application in the `src/config` directory:
 - `src/config/ipc_test_hwarch.c`
 - `src/config/ipc_test_swarch.c`
 - `src/config/ipc_test_map.c`
- Sun Netra DPS application files in the `src/app` directory:
 - `src/app/common.h`
 - `src/app/init.c`
 - `src/app/ipc_test_config.h`
 - `src/app/ipc_test.c`
 - `src/app/lb_objects.h`
 - `src/app/ldc_malloc_config.h`
 - `src/app/ldc_malloc.c`
- Oracle Solaris OS user space application in `src/solaris/cmd`:
 - `src/solaris/cmd/ipctest.c`

- `src/solaris/cmd/Makefile`
- Oracle Solaris STREAMS module in the `src/solaris/module`:
 - `src/solaris/module/include/lwmod.h`
 - `src/solaris/module/lwmod.c`
 - `src/solaris/module/Makefile`

Building the IPC Reference Application

This section includes descriptions of how to build the IPC reference application.

Usage

```
build cmt [single_thread] | solaris
```

Argument Descriptions

The build script supports the following arguments:

- *cmt*

Specifies whether to build the `ipc_test` application to run on the CMT1 (UltraSPARC T1) platform or CMT2 (UltraSPARC T2) platform.

 - `cmt1` – Build for CMT1 (UltraSPARC T1)
 - `cmt2` – Build for CMT2 (UltraSPARC T2)

This argument is required to build the Sun Netra DPS application.
- [*single_thread*]

With this option, two data IPC channels are polled by the same thread. In the default case, three channels are polled, each one on its own thread. The interfaces and usage for the Oracle Solaris side remain unchanged.
- *solaris*

Build the Oracle Solaris OS user space application and the STREAMS module in their respective source directories.

Example

The following commands below build the Sun Netra DPS application for single thread polling on an UltraSPARC T2 processor and the Oracle Solaris components, respectively.

```
% ./build cmt2 single_thread
% ./build solaris
```

Running the IPC Application

In addition to the channels described in “[Example Environment for UltraSPARC T1 Based Servers](#)” on page 94, two IPC channels with IDs 5 and 6, respectively, need to be set up using the `ldm` and `tnsmct1` commands.

The Sun Netra DPS application is booted from either a physical or a virtual network interface assigned to its domain. For example, if a `tftp` server has been set up in the subnet, and there is a `vnet` interface for the Sun Netra DPS domain, the IPC test application can be booted with the following command at the OpenBoot PROM:

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

▼ To Use the `ipctest` Utility

1. Boot the `ipc_test` application in the Sun Netra DPS domain
2. Use the `tnsmct1` utility from the control domain to set up the IPC channels.
3. Copy the `ipctest` binary from the `src/solaris/cmd` directory to the Oracle Solaris domain.

For example, `ldg2` as shown in the Oracle Solaris OS user space application in `src/solaris/cmd`.

The `ipctest` utility drives a single IPC channel, which is selected by the `connect` command (see “[ipctest Commands](#)” on page 308). Multiple channels can be driven by separate instances of the utility. The utility can be used at the same time as the `STREAMS` module (see “[To Install the `lwmod` STREAMS Module](#)” on page 308). In this case, however, the IPC channel with ID 5 is not available for this utility. For example, the utility can be used on channel 4 to read statistics of the traffic between the Sun Netra DPS application and the Solaris module on channel 5.

ipctest Commands

The ipctest utility opens the tnsn 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.
- `stats`

Requests statistics from the `ipc_test` application and displays them.
- `perf-stats iterations`

Requests statistics from the `ipc_test` application for iterations times and displays the time used.
- `perf-pkts-rx num_messages message_size`

Sends request to the Sun Netra DPS to send `num_messages` messages with a data size of `message_size` and to receive the messages.
- `perf-pkts-tx num_messages message_size`

Sends `num_messages` messages with a data size of `message_size` to the Sun Netra DPS domain.
- `perf-pkts-rx-tx num_messages message_size`

Sends request to the Sun Netra DPS to send `num_messages` messages with a data size of `message_size` and to receive the messages. Also, spawns a thread that sends as many messages of the same size to the Sun Netra DPS domain.
- `exit, x, quit, or q`

Exits the program.
- `help`

Contains program help information.

▼ To Install the lwmod STREAMS Module

1. **Copy the lwmod module from the `src/solaris/module/sparcv9` directory to the Oracle Solaris domain.**

For example, `ldg2` as shown in the Solaris OS STREAMS module in `src/solaris/module`.

2. Load and insert the module just above the driver for either a virtual or a physical networking device.

To use a physical device, modify the configuration such that the primary domain is connected through IPC channel 5, or, on an UltraSPARC T1-based system, assign the second PCI bus to `ldg2`.

Note – Before inserting the module, the `ipc_test` application must have been booted in the Sun Netra DPS domain, and the IPC channels must have been set up.

3. Set up the module on a secondary `vnet` interface:

```
# modload lwmod
# ifconfig vnet1 modinsert lwmod@2
```

4. Display the position of the module:

```
# ifconfig vnet1 modlist
0 arp
1 ip
2 lwmod
3 vnet
```

With the module installed, all packets sent to `vnet1` will be diverted to the Sun Netra DPS domain, where the application will reverse the MAC addresses and echo the packets back to the Oracle Solaris module. The module will transmit the packet on the same interface.

Note – No packet will be delivered to the stack above the module. If networking to the domain is needed, the module should not be inserted in the primary interface.

▼ To Remove the `lwmod` STREAMS Module

- **Type:**

```
# ifconfig vnet1 modremove lwmod@2
```

Transparent Interprocess Communication Reference Application

The TIPC reference application contained in the Sun Netra DPS package is similar to example applications available with the Oracle Solaris OS TIPC package. The functionalities provided by this reference application are:

- HelloWorld – Demonstrates message exchange between server and client in connection less mode. The application can be compiled with following sub functionality:
 - Loopback mode – The server and client run on the same TIPC node, without requiring sending the messages on wire.
 - Server mode – Only the server runs on the Sun Netra DPS domain. The client must be run on another TIPC node.
 - Client mode – Only the client runs on the Sun Netra DPS domain. The server must be run on another TIPC node.
- Connection demo – Demonstrates message exchange between server and client in connection oriented mode. The application can be compiled with following sub functionality:
 - Loopback mode – The server and client run on the same TIPC node, without requiring sending the messages on wire.
 - Server mode – Only the server runs on the Sun Netra DPS domain. The client must be run on another TIPC node.
 - Client mode – Only the client runs on the Sun Netra DPS domain. The server must be run on another TIPC node.

The Loopback functions, HelloWorld loopback, and connection demo loopback can be run in TIPC standalone mode, as the server and client run on the same TIPC node.

The reference 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 applications. The files for this implementation are located in the `src/app` subdirectory.

Source Files

All TIPC example source files are located in the following package directory:
`/opt/SUNWndps/src/apps/tipc.`

The contents include:

- The makefile used for building:
 - `./Makefile.nxge`
- Information file for TIPC examples:
 - `./README`
- Build script for one step build include:
 - `./build`
- System configuration for the application:
 - `./src/config/hwarch.c`
 - `./src/config/map.c`
 - `./src/config/swarch.c`

The hardware architecture is similar to the ones used for other reference applications.

The mapping file contains a mapping for each strand of the target domain:

- `tipc_eth.c` file contains simple functions that use the Ethernet driver to receive and transmit a packet.
- `tipc_util.c` file contains the memory allocation provided for the Ethernet driver.
- `init.c` file contains the initialization code for the application. First, the queues are initialized. The initialization of the Oracle VM Server for SPARC software framework is accomplished using calls to the functions `mach_descrip_init()`, `lwrtw_cnex_init()`, `lwrtw_init_ldc()`, and `tnipc_init()`. After this initialization, the TIPC is initialized by a call of `tipc_init()`. The first four functions must be called in this specific order.
- `tipc_app.c` file contains 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 based on the application that is built.
- `hello_world_client.c` file contains implementations of a connectionless TIPC client similar to the client available in the TIPC examples package.
- `hello_world_server.c` file contains implementations of connectionless TIPC server similar to the server available in TIPC examples package.
- `conn_demo_client.c` file contains implementations of connection-oriented TIPC client similar to the client available in the TIPC examples package.

- `conn_demo_server.c` file contains implementations of connection-oriented TIPC server similar to the server available in the TIPC examples package.

Default Configurations

TABLE 11-21 shows the default system configurations:

TABLE 11-21 TIPC Default System Configurations

	Sun Netra DPS Domain (strand IDs)	Statistics (strand ID)
CMT1 logical domain	0 to 7	7
CMT2 logical domain	0 to 7	7

The main files that control the system configurations are:

- `./src/config/swarch.c`
- `./src/config/map.c`

▼ To Compile the TIPC Application

1. **Copy the TIPC reference application from the `/opt/SUNWndps/src/apps/tipc` directory to a desired directory.**
2. **Create the build script in that location.**

Build Script

TABLE 11-22 shows the TIPC application build script.

TABLE 11-22 TIPC Application Build Script

Build Script	Usage
<code>./build</code> (See “Argument Descriptions” on page 313.)	Build TIPC HelloWorld application to run in loopback mode.

TABLE 11-22 TIPC Application Build Script

Build Script	Usage
	Build TIPC HelloWorld application (HelloWorld client and HelloWorld server) application to run in network mode.
	Build TIPC connection demo application to run in loopback mode.
	Build TIPC connection demo application (connection demo client and connection demo server) to run in network mode.

Usage

```
./build cmt type app
```

Argument Descriptions

The build script supports the following arguments:

- *cmt*

Specifies whether to build the TIPC application to run on the CMT1 (UltraSPARC T1) platform or CMT2 (UltraSPARC T2) platform.

- *cmt1* – Build for CMT1 (UltraSPARC T1)
- *cmt2* – Build for CMT2 (UltraSPARC T2)

- *type*

- *4g* – Build TIPC application to use on 4-Gbps Ethernet QGC (quad 1-Gbps *nxge* Ethernet interface).
- *10g* – Build TIPC application to use on 10-Gbps Ethernet (dual 10-Gbps Multithreaded Ethernet PCI-E interface).
- *10g_niu* – Build TIPC application to use on NIU (dual 10-Gbps UltraSPARC T2 on-chip Ethernet interface) on a CMT2-based system.
- *vnet* – Build TIPC application to use *vnet* interfaces.

- *app*
 - `helloworld_server` – Build HelloWorld server similar to HelloWorld server available in TIPC example package.
 - `helloworld_client` – Build HelloWorld client similar to HelloWorld client available in TIPC example package.
 - `helloworld_loopback` – Build HelloWorld server and client to run in Sun Netra DPS in standalone or loopback mode.
 - `conn_demo_loopback` – Build connection demo server and client to run in Sun Netra DPS in standalone or loopback mode.
 - `conn_demo_client` – Build connection demo client similar to connection demo client available in TIPC example package.
 - `conn_demo_server` – Build Connection demo server similar to connection demo server available in TIPC example package.
- `VNET_TIPC_CONFIG`

This option enables the TIPC stack in the Sun Netra DPS application to be configured using the `tn-tipc-config` tool for the Linux platform. The Linux `tn-tipc-config` tool uses `vnet` for exchanging commands and data. When the Linux `tn-tipc-config` tool is used, the Sun Netra DPS application must be compiled with the `-DTIPC_VNET_CONFIG` flag enabled in the `makefile` (for example, `Makefile.nxge`).

▼ To Run the TIPC Application

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

```
% cp your-workspace/tipc/code/main/main /tftpboot/tipc_app
```

3. At the `ok` prompt on the target machine, type:

```
ok boot network-device: ,tipc_app
```

4. Configure the TIPC stack using the `tipc-config` tool as described in “Configuring Environment for TIPC” on page 158.

IP Forward Reference Application Using TIPC

TIPC is integrated with the IP forwarding application. IP forwarding application uses TIPC to communicate with the control plane applications (`fibctl`, `ifctl`, and `excpd`). In the IP forward application, the TIPC stack runs in the fast path manager strand.

The `ipfwd` application with TIPC requires a logical domain environment because all configurations are set up through an application running on a Oracle Solaris OS control domain.

▼ To Build the IP Packet Forward (`ipfwd`) Application

- Specify the `tipc` keyword on the build script command line.

For example:

```
% ./build cmt2 10g_niu ldoms tipc
```

▼ To Configure the Environment for TIPC

1. Set up an IPC channel ID 10 to configure the TIPC stack.

For example:

```
# tnsnmctl -S -C 10 -L 7 -R 6 -F 3
```

To use IPC Channel as TIPC medium-bearer, set up an IPC channel for IPC medium. Note that channel ID 10 cannot be used as IPC bearer.

The following example shows how to configure IPC channel ID 6:

```
# tnsnmctl -S -C 6 -L 8 -R 7 -F 3
```

2. Set the TIPC address to the TIPC stack.

For example:

```
# /opt/SUNWndpsd/bin/tn-tipc-config -addr=10.3.4
```

3. Enable the medium of communication.

TIPC supports IPC channel or the Ethernet interface as the medium of communication.

The following example shows how to enable bearer on IPC channel ID 6 with proto 200.

```
# /opt/SUNWndpsd/bin/tn-tipc-config -be=ipc:6.200/10.3.0
```

To support Ethernet as the TIPC medium in the IP forward application, the application must be build with the `excp` option. The following example enables bearer on Ethernet port0:

```
# /opt/SUNWndpsd/bin/tn-tipc-config -be=eth:port0/10.3.0
```

▼ To Configure Oracle Solaris OS TIPC Stack in Oracle Solaris Domain (1dg2)

1. Set up environment variables `LD_PRELOAD_32` and `LD_PRELOAD_64` before running any Oracle Solaris OS TIPC applications (for instance, `tipc-config`, `fibctl`, `ifctl`, or `excpd`).

```
# LD_PRELOAD_32=/opt/SUNWndps-tipc/lib/libtipsocket.so
# LD_PRELOAD_64=/opt/SUNWndps-tipc/lib/sparcv9/libtipsocket.so
# export LD_PRELOAD_32 LD_PRELOAD_64
```

2. Enable the medium of communication.

TIPC supports IPC channel or the Ethernet interface as the medium of communication.

The following example shows how to enable the bearer on IPC channel ID 6 with proto 200:

```
# /opt/SUNWndps-tipc/sbin/tipc-config -be=ipc:6.200/10.3.0
```

The following example shows how to enable the bearer on Ethernet interface `nxge0`:

```
# /opt/SUNWndps-tipc/sbin/tipc-config -be=eth:nxge0/10.3.0
```

Command-Line Interface Application using TIPC

The IPv4 forwarding information base (FIB) table configuration (`fibctl`) command-line interface (CLI), interface configuration tool (`ifctl`), and IPV4 exception process (`excpd`) have been extended to support TIPC.

▼ To Build the Extended Control Utility

1. To build `fibctl` and `ifctl`, issue the following command in the `src/solaris` subdirectory of the IP forwarding reference application:

```
% gmake TIPC=on
```

2. To build `excpd`, see [“Compiling the `excpd` Application” on page 195](#).
3. To build `lwmodip4`, see [“Compiling the `lwmodip4` STREAMS Module” on page 197](#).
4. To build `lwmodarp`, see [“Compiling the `lwmodarp` STREAMS Module” on page 197](#).
5. To build `lwmodip6`, see [“Compiling the `lwmodip6` STREAMS module” on page 213](#).

FIB Table Configuration Command Line Interface (fibctl)

When IP forward application TIPC address is given, `fibctl` connects to the corresponding IP forward application with the given TIPC address.

```
fibctl> connect IP-forward-TIPC-application-TIPC-address
```

If no TIPC address is given, then `fibctl` tries to discover available IP forward application(s). If only one IP forward application is found, then `fibctl` connects to the found `IpFwd` application. If multiple IP forward applications are found, then it prompts the user to choose the IP forward applications and connects to the selected IP forward applications.

You can use the `status` command to obtain the status of connectivity with the IP forward application:

```
fibctl> status
```

The `status` command prints the status of connectivity:

- `CONNECTED` – `fibctl` is connected to an IP forward application.
- `NOT CONNECTED` – `fibctl` is not connected to an IP forward application.

Interface Configuration Command Line Interface (ifctl)

The `ifctl` commands are the same as explained in the `ifctl` commands list. The tools establish connection with the first available IP forward application.

IPv4 Exception Process (excpd)

The `excpd` process runs as the TIPC server, and the IP forward application runs as a TIPC client. When the IPv4 exception process is up, the IP forward application connects to the `excpd` process and starts communicating with each other.

vnet Reference Application

The vnet reference application illustrates the usage of the vnet Driver API, and it can be used to measure the performance of the Sun Netra DPS vnet driver. The vnet reference application consists of the following components:

- The Sun Netra DPS application that receives and transmits frames
- The Oracle Solaris OS or Linux OS test utility that receives and transmits packets from user space

The application runs in a logical domain environment. To use the application, the user must have the following logical domain setup:

TABLE 11-2 Logical Domain configuration for vnet Reference Application

Domain	Environment	Description
Primary	Solaris OS	Owns one of the PCI buses and uses the physical disks and networking interfaces to provide virtual I/O to the guest domains.
ldg1	LWRITE (ndps)	Owns the other PCI bus (in case of UltraSPARC T1 platform) or the NIU (in case of UltraSPARC T2 platform) and runs the Sun Netra DPS vnet application.
ldg2	Solaris or Linux OS	Runs the control plane applications.
ldg3	Solaris or Linux OS	Controls Sun Netra DPS domain through global control channel.

UltraSPARC T2 Platform

The Sun Netra DPS logical domain (ldg1) must be assigned 40 strands. The guest logical domain (ldg2) must be assigned at least 16 strands.

UltraSPARC T1 Platform

The Sun Netra DPS logical domain (ldg1) must be assigned 20 strands. The guest logical domain (ldg2) must be assigned at least 4 strands.

Supported Tests

The Sun Netra DPS binary for the `vnet` reference application is called `vnettest`, and the guest logical domain application is called `testvnet`.

The `vnet` reference application supports the following tests:

1. Transmit packets from guest logical domain to Sun Netra DPS logical domain
2. Transmit packets from Sun Netra DPS logical domain to guest logical domain
3. Loop-back packets transmitted from guest logical domain to Sun Netra DPS logical domain
4. Loop-back packets transmitted from guest logical domain to Sun Netra DPS logical domain
Performs data integrity check on the loop-backed packets in guest logical domain. This does not support the use of more than one `vnet` interface.
5. Transmit packets from Sun Netra DPS logical domain to Sun Netra DPS logical domain.

`testvnet` Commands

The `testvnet` utility offers the following commands:

- `tx`
Transmits frames to Sun Netra DPS logical domain application from the guest logical domain test application using the specified `vnet` interfaces.
- `rx`
Receives packets that are transmitted from Sun Netra DPS logical domain application in the guest logical domain test application over the specified `vnet` interfaces.
- `lpbk`
Loops back packets sent from the guest logical domain test application over the specified `vnet` interfaces.
- `lpbk-di`
Sun Netra DPS logical domain application loops back packets sent from guest logical domain test application over the specified interface. Test application in guest logical domain verifies data received with data sent for each `vnet` interface specified. Currently, more than one interface cannot be specified for this test.
- `dp-tx`

Transmits frames to itself using two `vnet` interfaces: one for transmitting the frames and another one for receiving the frames. Currently, this test supports only one interface (that is, one interface to transmit and another interface to receive).

- `pkt-sz`

Specifies the frame size to be used for the test (that is, it includes the size of the Ethernet, IP, and UDP headers).

- `pkt-cnt`

Specifies the number of frames to be used for the test. A value of 0 implies infinite count.

- `thd-cnt`

Specifies the number of threads to be used in the guest logical domain for the test. The value provided is for each interface specified.

- `intf-cnt`

Specifies the number of `vnet` interfaces to be used for the test.

Test Setup

The `vnet` reference application uses `vnet` interfaces and UDP sockets to perform the tests. The guest logical domain application, `testvnet`, and the Sun Netra DPS application, `vnettest`, behave as the UDP client or server depending on the test. During the test, the client transmits UDP packets to the server. The packets are destined to UDP port numbers that are determined appropriate.

Two types of UDP sockets are used: control sockets and data sockets. The guest logical domain application uses a single UDP control socket bound to UDP port number 1111 and the Sun Netra DPS application uses a single UDP control socket bound to UDP port 2222. The control sockets are used to exchange commands and responses during the test setup. The data sockets are used to exchange the test packets. The Sun Netra DPS uses data sockets with UDP port numbers starting from 8888. The guest logical domain uses data sockets with UDP port numbers starting from 4444.

Any number of `vnet` devices can be used for the tests. The test applications expect the instance numbers of the `vnet` devices used in the Sun Netra DPS and the guest logical domain to be consecutive. The first `vnet` device in the guest logical domain and the first `vnet` interface in the Sun Netra DPS logical domain is used for exchanging control packets. When using multiple interfaces for a test, interfaces starting from the lowest instance must be used. For example, if `vnet1`, `vnet2`, `vnet3`, and `vnet4` are enabled and a test is run with two interfaces, then `vnet1` and `vnet2` must be used. If the test is run with three interfaces, then `vnet1`, `vnet2`, and `vnet3` must be used.

The testvnet application uses one or more Light Weight Processes (LWP) to perform the tests. The number of LWPs to use is specified by the user in the command line. For each LWP created, a distinct socket end-point is used for the transmit or the receive. The following illustrates the UDP port number mappings for various tests:

TABLE 11-3 vnet Test Configuration 1

Test	thd- cnt	intf- cnt	Guest Logical Domain (source port, destination port)	Sun Netra DPS Logical Domain (source port, destination port)
tx	1	1	(4444, 8888)	(8888, any)
rx	1	1	(4444, any)	(8888, 4444)
lpbk	1	1	(4444, 8888)	(8888, 4444)
lpbk-di	1	1	(4444, 8888)	(8888, 4444)
dp-tx	1	1	N/A	N/A

TABLE 11-4 vnet Test Configuration 2

Test	thd- cnt	intf- cnt	Guest Logical Domain (source port, destination port)	Sun Netra DPS Logical Domain (source port, destination port)
tx	2	1	(4444, 8888), (4445, 8888)	(8888, any)
rx	2	1	(4444, any), (4445, any)	(8888, 4444), (8888, 4445)
lpbk	2	1	Rx: (4444, any), (4445, any) Tx: (4446, 8888), (4447, 8888)	(8888, 4444), (8888, 4445)
lpbk-di	2	1	Rx: (4444, any) Tx: (4445, 8888)	(8888, 4444)

TABLE 11-5 vnet Test Configuration 3

Test	thd- cnt	intf- cnt	Guest Logical Domain (source port, destination port)	Sun Netra DPS Logical Domain (source port, destination port)
tx	2	2	vnet1: (4444, 8888), (4445, 8888) vnet2: (4446, 8889), (4447, 8889)	vnet1: (8888, any) vnet2: (8889, any)
rx	2	2	vnet1: (4444, any), (4445, any) vnet2: (4446, any), (4447, any)	vnet1: (8888, 4444), (8888, 4445) vnet2: (8889, 4446), (8889, 4447)
lpbk	2	2	vnet1: Rx: (4444, any), (4445, any) Tx: (4448, 8888), (4449, 8888) vnet2: Rx: (4446, any), (4447, any) Tx: (4450, 8889), (4451, 8889)	vnet1: (8888, 4444), (8888, 4445) vnet2: (8889, 4446), (8889, 4447)

Virtual Network Setup

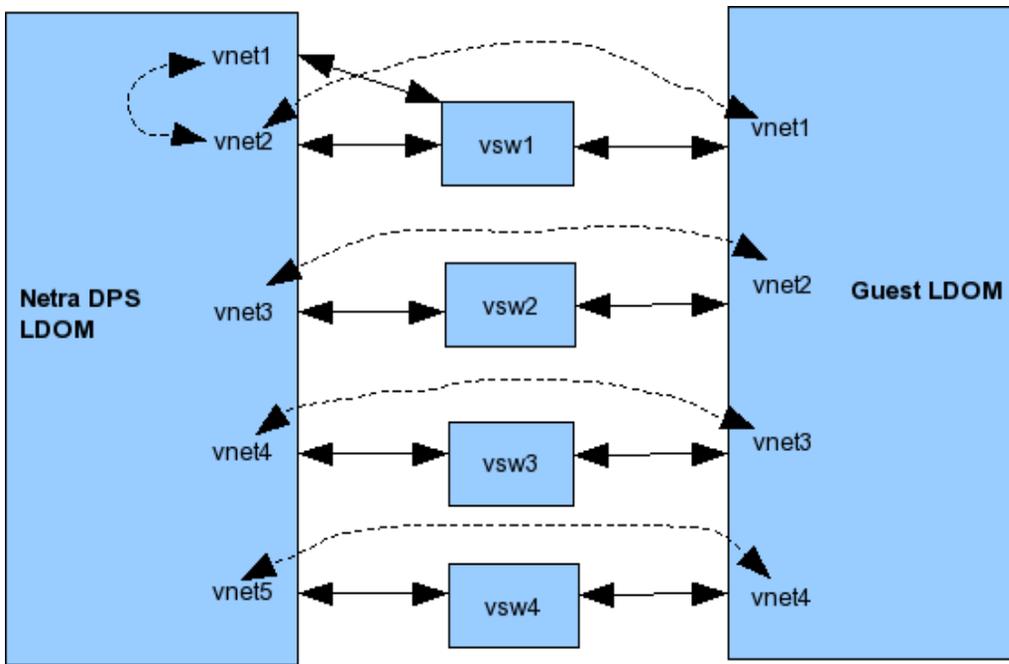
The number of interfaces to be used is determined by the user. Each Sun Netra DPS vnet interface must be directly connected to a guest logical domain vnet interface. This is achieved by linking a Sun Netra DPS vnet and a guest vnet to the same virtual switch. No more than one vnet interface in a logical domain must be attached to the same vswitch. The exception to this requirement is one of the vnet interfaces in the Sun Netra DPS logical domain that is used for dp-tx test. This vnet device is connected to the same vswitch as the another Sun Netra DPS vnet interface.

The following table and illustration show the setup of a virtual network with four vnet interfaces.

TABLE 11-23 Virtual Network Setup

Guest Logical Domain	Sun Netra DPS Logical Domain	Primary	Function
vnet1	vnet2	vsw1	Used for control packets and for data packets between vnet2 and vnet1
vnet2	vnet3	vsw2	Used for data packets between vnet3 and vnet2
vnet3	vnet4	vsw3	Used for data packets between vnet4 and vnet3
vnet4	vnet5	vsw4	Used for data packets between vnet5 and vnet4
	vnet1	vsw1	Data packets for dp-tx between vnet2 and vnet1.

FIGURE 11-15 vnet Test Configuration



In this example, the dotted lines illustrate the direct connection between `vnet` interfaces that are connected to the same `vswitch`.

The `vnet` interfaces must be assigned IP addresses. Also, the ARP must be disabled on the `vnet` devices used for the test. The IP addresses for the Sun Netra DPS `vnet` interfaces are assigned during the test setup.

When testing with VLANs, the `vnettest` application expects the VLAN ID to start from 11 and continue upwards. For example, in the illustration above, the following are VLAN IDs that must be assigned to the interfaces:

- Sun Netra DPS `vnet` interfaces: `vnet1` (11), `vnet2` (11 and 12), `vnet3` (13), `vnet4` (14), `vnet5` (15)
- Guest `vnet` interfaces: `vnet1` (12), `vnet2` (13), `vnet3` (14), `vnet4` (15)

vnet Reference Application Content

The source code for the `vnet` reference application is in the `SUNWndps` package in the `/opt/SUNWndps/src/apps/vnet_sample` directory. The source code includes the following:

- `makefile` for the application
- Common header file, `src/common/vnet_cmd.h`, describing the commands sent from the test utility to the Sun Netra DPS application
- System configuration for the Sun Netra DPS application in the `src/config` directory:
 - `src/config/hwarch.c`
 - `src/config/swarch.c`
 - `src/config/map.c`
- Sun Netra DPS application in the `src/app` directory:
 - `src/app/ldc_malloc.c`
 - `src/app/vnet_test_config.c`
 - `src/app/appln.c`
 - `src/app/init.c`
 - `src/app/user_common.c`
 - `src/app/vnet_ipc.c`
 - `src/app/ldc_malloc_config.h`
 - `src/app/vnet_test_config.h`
 - `src/app/lb_objects.h`
- Oracle Solaris application in `src/solaris` directory:
 - `src/solaris/makefile`

- `src/solaris/vnet_txx.c`
- `src/solaris/vnet_txx_ipc.c`
- `src/solaris/vnet_txx.h`
- Linux application in `src/linux` directory
 - `src/linux/makefile`
 - `src/linux/vnet_txx.c`
 - `src/linux/vnet_txx_ipc.c`
 - `src/linux/vnet_txx.h`

Building the Sun Netra DPS vnet Reference Application

This section includes descriptions of how to build the vnet reference application.

Usage

```
build cmt1 | cmt2 10g | 10g_niu | 4g [2port] [profiler] [vlan]
```

Argument Descriptions

The build script supports the following arguments:

- `cmt1` – Builds the Sun Netra DPS application to run on CMT1 (UltraSPARC T1) platform
- `cmt2` – Builds the Sun Netra DPS application to run on CMT2 (UltraSPARC T2) platform
- `10g` – Builds the Sun Netra DPS application to simulate the message block (`mblk`) and data buffer offset settings for Neptune 10g card.
- `10g_niu` – Builds the Sun Netra DPS application to simulate the message block (`mblk`) and data buffer offset settings for NIU.
- `4g` – Builds the Sun Netra DPS application to simulate the message block (`mblk`) and data buffer offset settings for Neptune QGC card.
- [`2port`] – Builds the Sun Netra DPS application to simulate the message block (`mblk`) and data buffer offset settings for Neptune 10g or NIU with 2 ports.
- `profiler` – Builds the reference application with profiling enabled
- `vlan` – Enables VLAN Tagging for frames used in the tests

▼ To Build the vnet Reference Application

- Execute the following build command:

```
# ./build cmt2 10g vlan
```

This command builds the Sun Netra DPS vnet application for the UltraSPARC T2 platform with VLAN tagging enabled for the test frames.

▼ To Run the vnet Sun Netra DPS Application, vnettest

The Sun Netra DPS application is booted from a virtual network interface assigned to its domain.

- Boot the application.

For example:

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

▼ To Build the vnet Guest Logical Domain Application for the Oracle Solaris OS

1. Change directories to:

```
/opt/SUNWndps/src/apps/vnet_sample/src/solaris
```

2. Run the following command:

```
% gmake
```

▼ Building the vnet Guest Logical Domain Application for the Linux OS

1. Change directories to: /opt/SUNWndps/src/apps/vnet_sample/src

2. Create a TAR file of the common and linux directories:

```
% tar -cvf testvnet-srscs.tar common/linux/
```

3. Copy the TAR file onto a system that has a cross-compiler for UltraSPARC T2.
4. Untar the file into a directory.

```
% mkdir testvnet-lnx
% cp testvnet-srcs.tar testvnet-lnx
% cd testvnet-lnx
% tar -xvf testvnet-srcs.tar
```

5. Change directories to the `linux` directory, and execute the `make` command.

```
% cd linux
% make
```

▼ To Run the `vnet` Guest Logical Domain Application on a Oracle Solaris OS Guest Logical Domain

1. Copy the `testvnet` binary into the guest logical domain.
2. Create a permanent, static ARP entry for the control `vnet`:

```
ok arp -s Netra-DPS-control-vnet-ip Netra-DPS-control-vnet-mac-address permanent
```

3. Start the `testvnet` application:

```
# ./testvnet tx 64 1000000 4 2
% make
```

The application prompts you to enter the IP addresses for the Sun Netra DPS `vnet` interfaces and the guest logical domain `vnet` interfaces to be used in the test

4. Enter IP address for the local interface to be used:

```
Enter IP address for the local interface to be used:
192.168.20.200
Enter IP address for the connected lwrte interface:
192.168.20.201
Enter IP address for the local interface to be used:
192.168.30.200
Enter IP address for the connected lwrte interface:
192.168.30.201
```

After you enter all of the IP addresses, the test starts. The `testvnet` application prints statistical information to the console. The Sun Netra DPS application also prints statistical information on its console. The statistics correspond to the measurements made by each end.

The statistics on the guest logical domain are on a LWP basis. An example is shown below. If more than one interface is used and if `n`-threads are specified as the thread count, then threads 0 to `n-1` are used for interface 0, threads `n` to $(2 * n - 1)$ are used for interface 1, and so on.

```
TRANSMIT STATISTICS - Thread 0
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 60197.255870, 10.594717

TRANSMIT STATISTICS - Thread 3
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 58018.923256, 10.211330

TRANSMIT STATISTICS - Thread 1
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 57842.894969, 10.180350

TRANSMIT STATISTICS - Thread 2
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 57516.098952, 10.122833
```

The statistics on the Sun Netra DPS console are on a per-port basis. An example is shown below:

```
RECEIVE STATISTICS: vnet3
-----
Rx-Cnt: 1048576 Rx-Bytes: 1587544064 Perf(pps, mbps): 117185.516316,
1419.350974 Rx-Retries: 82633548

RECEIVE STATISTICS: vnet2
-----
Rx-Cnt: 1048576 Rx-Bytes: 1587544064 Perf(pps, mbps): 118617.194570,
1436.691461 Rx-Retries: 81623147
```

▼ To Run the vnet Guest Logical Domain Application on a Linux OS Guest Logical Domain

1. Copy the `testvnet` binary onto the guest logical domain.
2. Create a permanent, static ARP entry for the control vnet:

```
# arp -s Netra-DPS-control-vnet-ip Netra-DPS-control-vnet-mac-address
```

3. Start the `testvnet` application:

```
# ./testvnet tx 64 1000000 4 2
```

The application prompts you to enter the IP addresses for the Sun Netra DPS vnet interfaces and also the guest logical domain vnet interfaces to be used in the test.

4. Enter the IP addresses:

```
Enter IP address for the local interface to be used:
192.168.20.200
Enter IP address for the connected lwrte interface:
192.168.20.201
Enter IP address for the local interface to be used:
192.168.30.200
Enter IP address for the connected lwrte interface:
192.168.30.201
```

After you have entered all of the IP addresses, the test starts. The `testvnet` application prints statistical information to the console. The Sun Netra DPS application also prints statistical information to its console. The statistics correspond to the measurements made by each end.

The statistics printed on the guest logical domain are on a LWP basis. An example is shown below. If more than one interface is used and if `n`-threads are specified as the thread count, then threads 0 to `n - 1` are used for interface 0, threads `n` to `(2 * n - 1)` are used for interface 1, and so on.

```
TRANSMIT STATISTICS - Thread 0
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 60197.255870, 10.594717

TRANSMIT STATISTICS - Thread 3
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 58018.923256, 10.211330

TRANSMIT STATISTICS - Thread 1
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 57842.894969, 10.180350

TRANSMIT STATISTICS - Thread 2
-----
Tx-Cnt: 1048576 Tx-Bytes: 23068672 Perf(pps, mbps): 57516.098952, 10.122833
```

The statistics on the Sun Netra DPS console are on a per-port basis. An example is shown below:

```
RECEIVE STATISTICS: vnet3
```

```
-----
```

```
Rx-Cnt: 1048576 Rx-Bytes: 1587544064 Perf(pps, mbps): 117185.516316,  
1419.350974 Rx-Retries: 82633548
```

```
RECEIVE STATISTICS: vnet2
```

```
-----
```

```
Rx-Cnt: 1048576 Rx-Bytes: 1587544064 Perf(pps, mbps): 118617.194570,  
1436.691461 Rx-Retries: 81623147
```


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” on page 333](#)
- [“UltraSPARC T1 Processor Overview” on page 334](#)
- [“UltraSPARC T2 Processor Overview” on page 336](#)
- [“Identifying Performance Issues” on page 338](#)
- [“Optimization Techniques” on page 343](#)
- [“Tuning Troubleshooting” on page 348](#)
- [“Example RLP Exercise” on page 350](#)

Performance Tuning Introduction

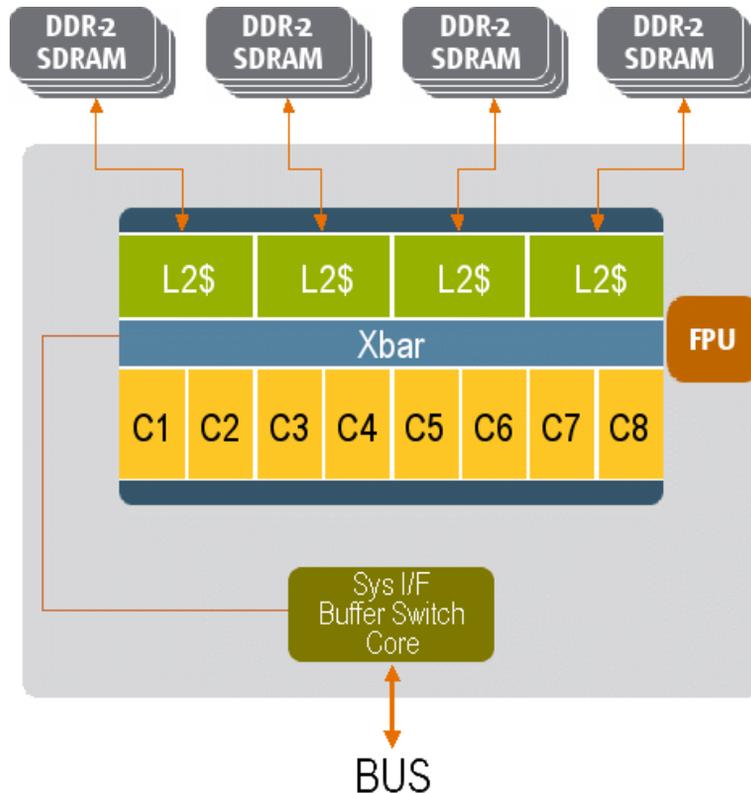
The UltraSPARC T series CMT systems deliver a strand-rich environment with performance and power efficiency that are unmatched by other processors. From a programming point of view, the UltraSPARC T1 and UltraSPARC T2 processor strand-rich environment can be thought of as symmetric multiprocessing on 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 12-1](#) shows the UltraSPARC T1 architecture.

FIGURE 12-1 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 Oracle 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 12-1](#) summarizes the key performance limits and latencies.

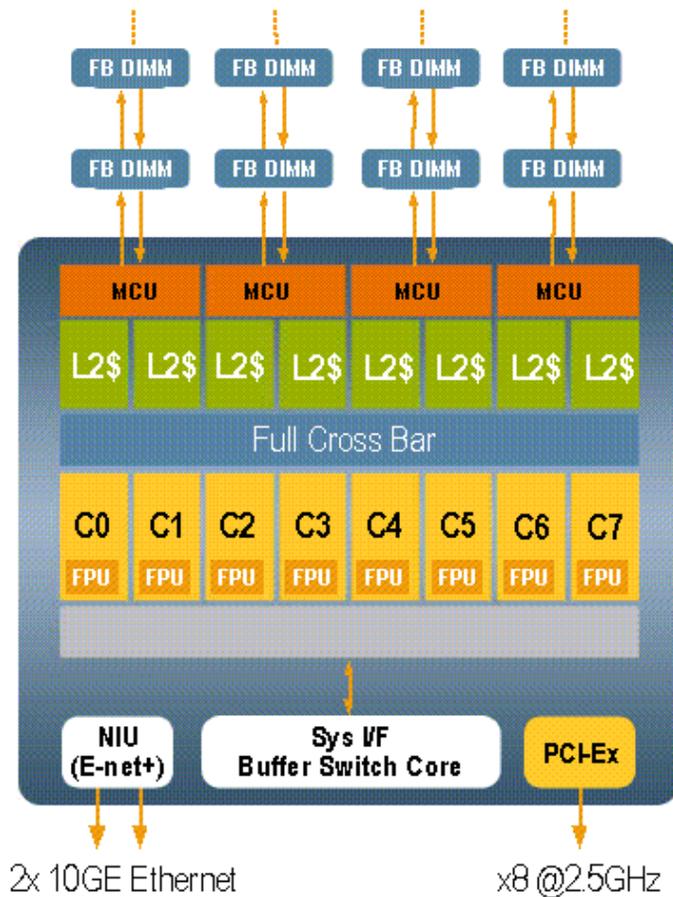
TABLE 12-1 UltraSPARC T1 Key Performance Limits and Latencies

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 12-2](#) shows the UltraSPARC T2 system architecture.

FIGURE 12-2 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 12-2 summarizes the key performance limits and latencies.

TABLE 12-2 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 12-2 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)

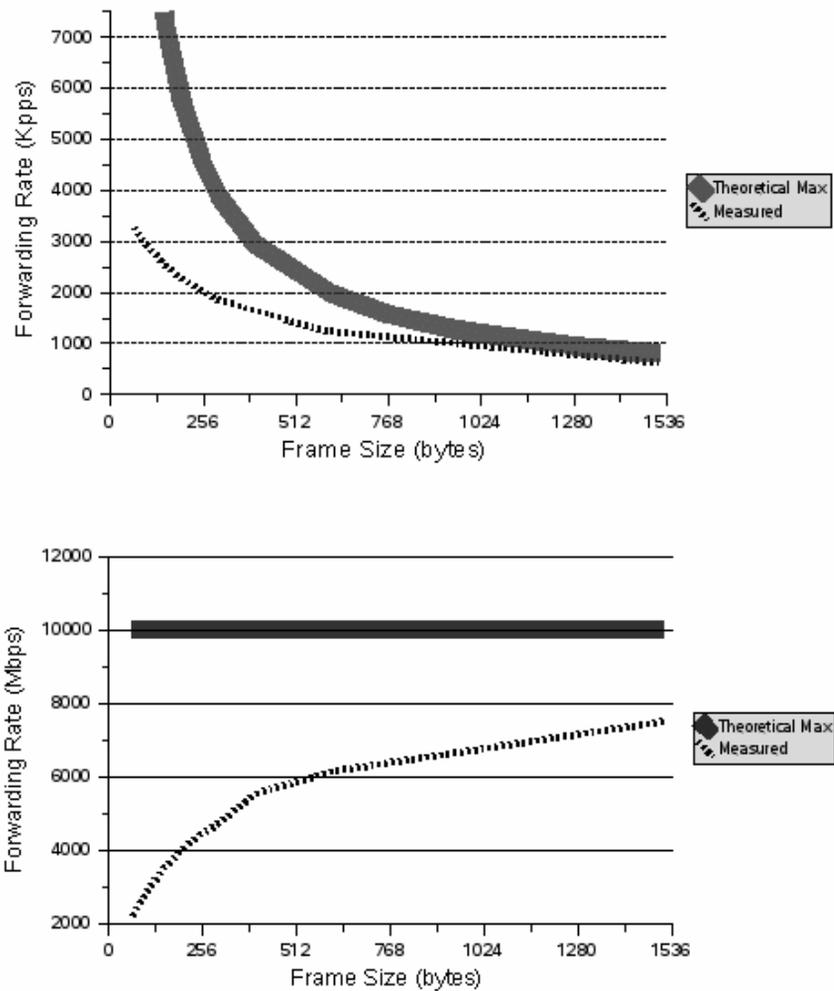
Identifying Performance Issues

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 12-3](#) shows the packet forwarding rate limited by this I/O bottleneck.

FIGURE 12-3 UltraSPARC T1 Forwarding Packet Rate Limited by I/O Throughput

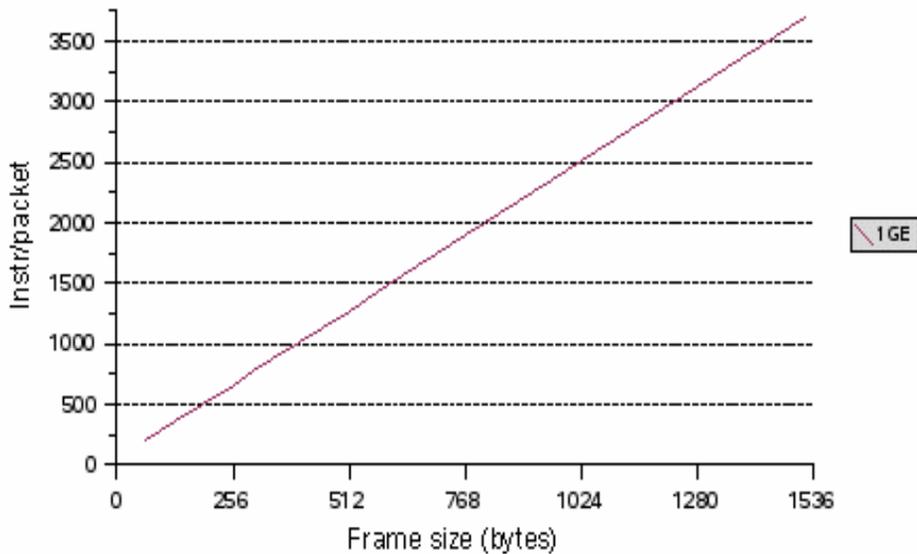


The theoretical maximum represents the throughput of 10 Gbytes per second. The measured results show that the achievable forwarding throughput is a function of packet size. For 64-byte packets, the measured throughput is 2.2 Gbyte per second or 3300 kilo packets per second.

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 12-4 shows the average maximum number of instructions the processor can execute per packet while maintaining line rate.

FIGURE 12-4 Instructions per Packet Versus Frame Size



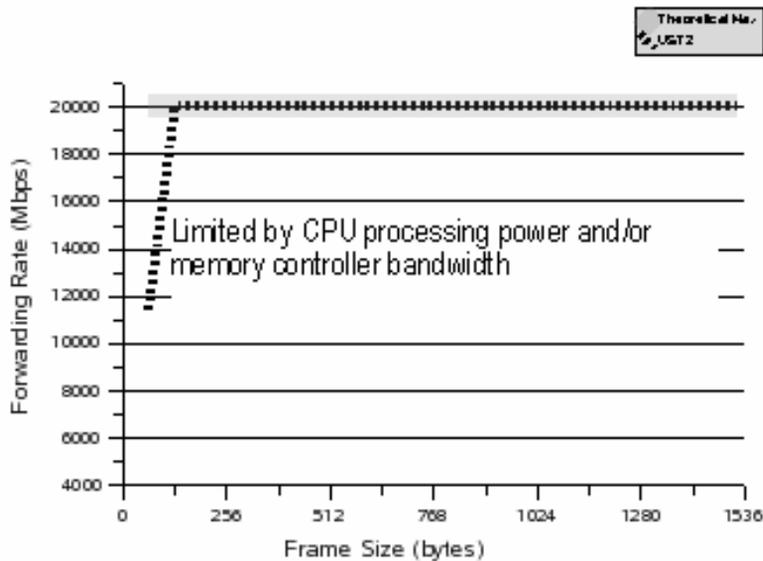
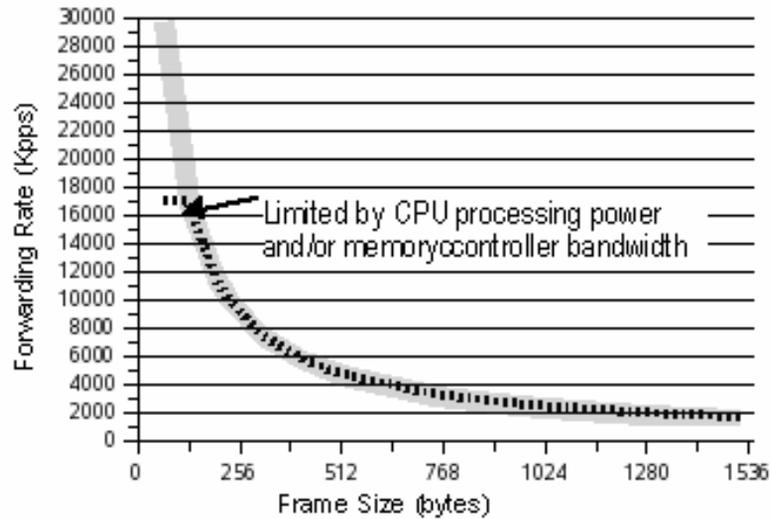
The inter-arrival time increases with packet size, so that more processing can be accomplished.

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 12-5 shows the forwarding packet rate limited by CPU processing power or memory controller bandwidth.

FIGURE 12-5 UltraSPARC T2 Forwarding Packet Rate



Theoretical Max.
U_{GT2}

Optimization Techniques

This section includes optimization techniques that you can use to improve tuning. The techniques include:

- [“Code Optimization” on page 343](#)
- [“Pipelining” on page 344](#)
- [“Parallelization” on page 345](#)
- [“Mapping” on page 346](#)
- [“Parking Idle Strands” on page 346](#)
- [“Slowing Down Polling” on page 347](#)

Code Optimization

Writing efficient code and using the appropriate compiler option is the primary step in obtaining optimal performance for an application. GCCfss compilers provide many optimization flags to tune your application. Refer to the *GNU C Compiler User’s Guide and GCC for SPARC Systems-Additional command line option flags* for the complete list of optimization flags available. See [“Reference Documentation” on page xxx](#). 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 enables the user 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. The user can program the split functions to communicate through Sun 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 12-6 shows how to split and map an application using fast queues and CMT processor to three strands.

FIGURE 12-6 Example of Pipelining

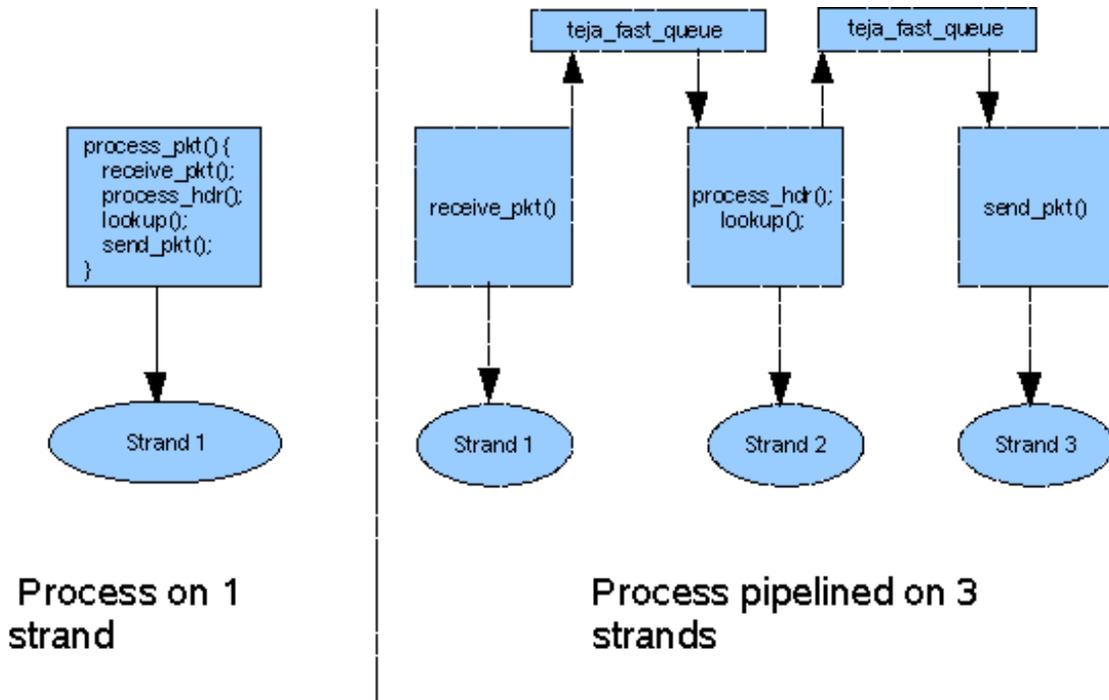
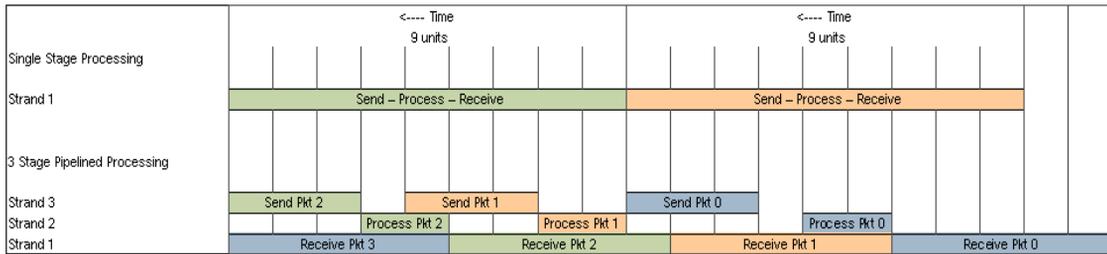


FIGURE 12-7 shows how pipelining improves the throughput.

FIGURE 12-7 Pipelining Effect on Throughput

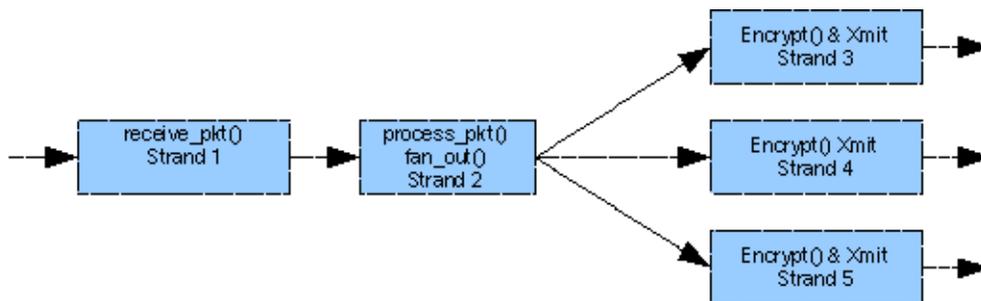


In this example, a single-strand application takes nine units of time to complete processing of a packet. The same application split into three functions and mapped to three different strands takes longer to complete the same processing, but is able to process more packets in the same time.

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, the user 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 12-8 is allocated three strands for encryption.

FIGURE 12-8 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 assign multiple instances of the same function on the same core. On UltraSPARC T1 processors, the user 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. The user 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 the user wants 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 346](#), 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 selected depends on your application. For instance, if the application is using the floating-point unit, the user might not want a useless floating-point instruction to slow down polling because that might stall useful floating-point instructions. Likewise, if the application is memory bound, using a memory instruction to slow polling might add memory latency to other memory instructions.

Tuning Troubleshooting

This section includes descriptions of troubleshooting techniques that you can perform to improve tuning. The troubleshooting techniques include:

- [“What Is a Compute-Bound Versus a Memory-Bound Thread?”](#) on page 348
- [“Cannot Reach Line Rate for Packets Smaller Than 300 Bytes”](#) on page 348
- [“Cannot Scale Throughput to Multiple Ports”](#) on page 349
- [“How Do I Achieve Line Rate for 64-byte Packets?”](#) on page 349
- [“When Should I Consider Thread Placement?”](#) on page 350

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. The user 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. The user 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. 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 (202×3) 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, 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.

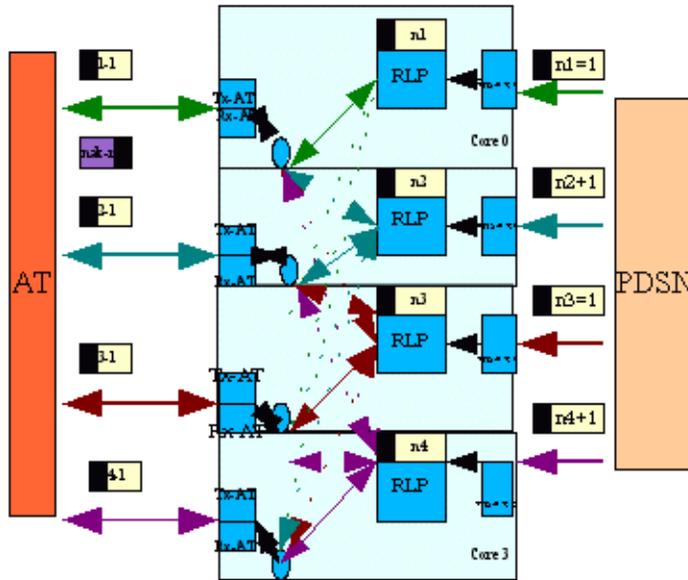
Application Configuration

The RLP reference application has three basic components:

- PDSN
- ATIF
- RLP

The PDSN and ATIF each have receive (RX) and transmit (TX) components. A Sun Netra T2000 system with four in-ports and four out-ports was configured for the four instances of the RLP application. [FIGURE 12-9](#) describes the architecture.

FIGURE 12-9 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 12-3](#).

TABLE 12-3 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 12-4](#).

TABLE 12-4 Configuration 2

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7
Strand 0	PDSN_RX_0	PDSN_RX_1	PDSN_RX_2	PDSN_RX_3	while(1)	while(1)	PDSN_TX_1	while(1)
Strand 1	RLP_0	RLP_1	RLP_2	RLP_3	while(1)	while(1)	PDSN_TX_2	while(1)
Strand 2	ATIF_RX_0	ATIF_RX_1	ATIF_RX_2	ATIF_RX_3	while(1)	while(1)	PDSN_TX_3	Profile thread
Strand 3	ATIF_TX_0	ATIF_TX_1	ATIF_TX_2	ATIF_TX_3	while(1)	PDSN_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 Sun Netra DPS Profiling API (refer to the *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*) and creates a simple function that collects hardware counter data for all the available counters per strand. The function is called from a relevant section of the application. The hardware counter data is related to application performance as the number of packets processed by the application-defined counter that is passed to the API. To reduce the performance impact of profiling, the profiling API is not called for each packet processed. For the RLP application and Sun Netra T2000 hardware combination, the API is called every five seconds, otherwise the counters overflow.

The pseudo-code in [EXAMPLE 12-1](#) shows the functions that were created to collect the hardware counter data.

EXAMPLE 12-1 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 [EXAMPLE 12-1](#). For a detailed description of these API functions refer to the *Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*.

A sample output based on the code in [EXAMPLE 12-1](#) is shown in [EXAMPLE 12-2](#).

EXAMPLE 12-2 Sample Profile Output

```
PROFILE_DUMP_START, ver, 2.1
CPUID, ID, Type, Cycles, PC, Grp, Evt_Hi, Evt_Lo, Overflow, User Data
4, 6043, 2, 30051d74c70, 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, 3094ddddeb10, 512598, 1, 3a303d12, 0, 0, 315a4700
4, 1fad3, 1, 3094ddd51c, 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 53](#). The output in [EXAMPLE 12-2](#) 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 12-5](#).

TABLE 12-5 Metrics

Metrics	Description
Instructions per packet	Average path length to process 1 packet
Instructions per cycle	Strands instruction processing rate
Packet rate (Kpps)	Packet processing rate
SB_full per 1000 instructions	The hardware counter rates per 1000 instructions enables comparison rates from different strands.
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_iss per 1000 instructions	
L2_dmiss_ld per 1000 instructions	

These metrics in [TABLE 12-5](#) provide insight into the performance of each strand and of each core.

Configuration 1 Results

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 12-10](#).

FIGURE 12-10 Results From Configuration 1

65% Line		342 Byte Packet			4in-4out							
Strands		Instruction /Packet	Instruction s/Cycle	Packet Rate(Kpps)	HW Counter / 1000 Instruction							
					SB_full	FP_inst r_cnt	IC_miss	DC_miss	ITLB_ miss	DTLB_ miss	L2_imis s	L2_dmis s_Id
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											

Configuration 2 Results

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 12-11](#).

FIGURE 12-11 Results From Configuration 2

Strands	90% Line Rate	342 Byte Packet			4in-4out							
		Instruction /Packet	Instruction s/Cycle	Packet Rate(Kpps)	HW Counter / 1000				Instruction			
					SB_full	FP_inst r_cnt	IC_miss	DC_miss	ITLB miss	DTLB miss	L2_imis s	L2_dmis s_Id
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											

Analysis

When comparing the processed hardware counter information it is necessary to correlate 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 enables you to optimize a particular section of the application.

Comparing the Instruction per Cycle columns from [FIGURE 12-10](#) and [FIGURE 12-11](#) 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. The user can estimate the effect of these misses by calculating the number of cycles these misses add to overall processing. Use information from TABLE 12-1 to calculate the worst case effect of the data cache and L2 cache misses. The results for these calculations are shown in TABLE 12-6 for configuration 1 and in TABLE 12-7 for configuration 2.

TABLE 12-6 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 12-7 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, the user can use the API to profile specific sections of the code. This type of profiling enables the user to make decisions regarding pipelining and reorganizing memory structures in the application.

Tutorial

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

- “Application Code” on page 363
- “Configuration Code” on page 366
- “Build Process” on page 368
- “Executing the Binary Image” on page 370

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.

[EXAMPLE A-1](#) lists the `ticktock.c` file and provides comment.

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 */
    }
}
```

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.

EXAMPLE A-1 ticktock.c File and Comments (Continued)

```
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;
}
```

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.

[EXAMPLE A-2](#) lists the `config.c` file and provides comment.

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

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.

EXAMPLE A-2 config.c File and Comments (Continued)

```
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 10](#). 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 ...
```

Frequently Asked Questions

This appendix provides frequently asked questions regarding Oracle's Sun Netra DPS.

- ["Summary" on page 372](#)
- ["General Questions" on page 374](#)
- ["Configuration Questions" on page 375](#)
- ["Building Questions" on page 377](#)
- ["Late-Binding Questions" on page 380](#)
- ["Eclipse Questions" on page 382](#)
- ["API and Application Questions" on page 383](#)
- ["Optimization Questions" on page 391](#)
- ["Legacy Code Integration Questions" on page 392](#)
- ["Example for the ipfwd Application" on page 384](#)
- ["Address Resolution Protocol Questions" on page 396](#)
- ["Oracle Solaris Domain and Sun Netra DPS Domain Question" on page 398](#)
- ["Traffic Generation" on page 398](#)
- ["Oracle Solaris TIPC Application" on page 399](#)

Summary

General Questions

- “What Is Teja 4.x and How Does It Differ From an Ordinary C Compiler?” on page 374
- “Where Are the Tutorials?” on page 375

Configuration Questions

- “What Purpose Are the Hardware Architecture, Software Architecture, and Mapping Dynamic Libraries?” on page 375
- “How Can I Debug the Dynamic Libraries?” on page 375
- “What Should I Do When the `tejacc` Compiler Crashes?” on page 376
- “What if the Hardware Architecture, Software Architecture, or Mapping Dynamic Libraries Crash?” on page 376
- “Can I Build Hardware Architecture, Software Architecture, and Mapping in the Same Dynamic Library?” on page 377
- “Can I Map Multiple Variables With One Function Call?” on page 377

Building Questions

- “Where Is the Generated Code?” on page 377
- “Where Is the Executable Image?” on page 378
- “How Can I Compile Multiple Modules on the Same Command Line?” on page 378
- “How Can I Pass Different CLI Options to Different Modules on the `tejacc` Command Line?” on page 378
- “How Can I Change the Behavior of the Generated `makefile` Without Modifying it?” on page 378
- “How Do I Compile the Reference Applications?” on page 379

Late-Binding Questions

- “What Is the Late-Binding API?” on page 380
- “What Is a Memory Pool?” on page 380
- “What Is a Channel?” on page 381
- “What Is the Difference Between OS Based and Non-OS Based Memory Pools and Channels?” on page 381
- “How Do I Access a Late-Binding Object From Application Code?” on page 381
- “Can I Define a Symbol in the Software Architecture and Use it in My Application Code?” on page 382

Eclipse Questions

- “How Can I Change the Build Command?” on page 382
- “How Can I Change the Compiler Invocation Command?” on page 382

API and Application Questions

- “How Do I Synchronize a Critical Region?” on page 383
- “How Do I Send Data From a Thread to Another Thread?” on page 383
- “How Do I Allocate Memory?” on page 384
- “When Should I Use Queues Instead of Channels?” on page 384
- “Why Is it Not Necessary to Block Interface or Queue Reads?” on page 384
- “Can Multiple Strands on the Same Queue Take Advantage of the Extra CPU Cycles if the Strands Are Not Being Used?” on page 385
- “Why Does the Application Choose the Role for the Strand From the Code Instead of the Software Architecture API?” on page 385
- “Is It Possible to Park a Strand Under Logical Domains as Done in a Non-Logical Domains Environment?” on page 386
- “What Is `bss_mem`?” on page 386
- “What Is the Significance of `bss_mem` Placement in the Code Listing?” on page 386
- “How Are `app.cmt2board.heap_mem0` and Similar Heaps Affected?” on page 387
- “Can You Clarify BSS, Code, Heap, and DRAM Memory Allocation?” on page 387
- “Does the `eth_*` API Support Virtual Ethernet (VNET) Devices?” on page 388
- “How Do I Calculate the Base PA Address for NIU or Logical Domains to Use with the `tnsmctl` Command?” on page 388

Optimization Questions

- “How Do I Enable Optimization?” on page 391
- “What Is Context-Sensitive Generation?” on page 392
- “What Is Global Inlining?” on page 392

Legacy Code Integration Questions

- “How Can I Reuse Legacy C Code in a Sun Netra DPS Application?” on page 392
- “How Can I Reuse Legacy C++ Code in a Sun Netra DPS Application?” on page 393

Sun CMT Specific Questions

- [“Is There a Maximum Allowed Size for Text and BSS in My Program?”](#) on page 395
- [“How Is Memory Organized in the Sun CMT Hardware Architecture?”](#) on page 395
- [“How Do I Increase the Size of the DRAM membank?”](#) on page 396

Address Resolution Protocol Questions

- [“How Do I Enable ARP in the RLP Application?”](#) on page 396
- [“How Do I Enable ARP Without Relying on a Control Domain?”](#) on page 397
- [“How Do I Enable ARP Using a Control Domain?”](#) on page 397

Oracle Solaris Domain and Sun Netra DPS Domain Question

- [“How Do I Access `kstat` Information From the Oracle Solaris Domain for Network Interfaces That Are in Use by the Sun Netra DPS domain?”](#) on page 398

Traffic Generation

- [“How Do I Stop Traffic Generation?”](#) on page 398

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 392.

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 363](#).

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 392](#).

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 375.

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 Oracle 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 378.

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 11, “Reference Applications”](#) on page 163.

Late-Binding Questions

Note – Refer to [“Late-Binding API Overview” on page 29](#) for more information on the Late-Binding API.

What Is the Late-Binding API?

The Late-Binding API is the Sun 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 392](#).

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 380](#)), so they benefit from context-sensitive generation.

What Is the Difference Between OS Based and Non-OS Based Memory Pools and Channels?

The operating system (OS) based memory pools and channels allocate buffer in the heap, which is limited by default. The non-OS based memory pools and channels allocate buffer with a memory map and have no limitation except the size of the RAM bank.

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 *Sun Netra Data Plane Software Suite 2.1 Update 1 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?

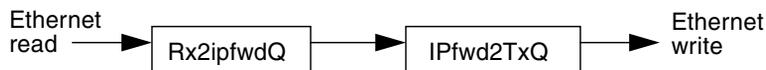
Generally, queues are more efficient than channels. Consider the following guidelines when deciding between queues or channels:

- 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 Sun 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 Logical Domains as Done in a Non-Logical Domains Environment?

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

Can You Assign Partial Cores to a Sun Netra DPS domain?

You must assign complete cores to the Sun 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.cmtlboard.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.

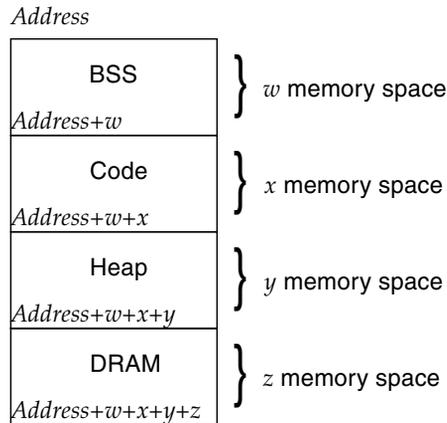
How Are `app.cmt2board.heap_mem0` and Similar Heaps Affected?

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-2 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.

Does the eth_* API Support Virtual Ethernet (VNET) Devices?

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

How Do I Calculate the Base PA Address for NIU or Logical Domains to Use with the tnsctl Command?

Command syntax:

```
tnsctl -P -v baseaddr
```

The *baseaddr* is needed when using NIU under logical domains; it is based on the logical domains configuration on the machine in question. The value is derived from the output of the `ldm` command for the domain in which the NIU will be operated under the Sun Netra DPS environment. This command is issued on the Oracle Solaris control domain.

```
# /opt/SUNWldm/bin/ldm list -l
NAME                STATE   FLAGS  CONS  VCPU  MEMORY  UTIL  UPTIME
ldg1                 bound  ----v  5000  16    4G
...
MEMORY
  RA                PA                SIZE
  0x8000000         0x48000000       4G
...
```

Assuming `ldg1` is the Sun Netra DPS domain in this example, then based on the above information, the *baseaddr* variable can be calculated as $PA - RA = baseaddr$. In the above example, the base PA address is `0x40000000` as calculated below:

$$0x48000000 - 0x8000000 = 0x40000000$$

How Do I Modify the IP Forwarding Application to Use a New Classifier Type Instead of the Default UDP Type?

The following is an example of modifying the IP forwarding application to the TCP classifier type:

1. Open the `ipfwd_classify.c` file.
2. Under the `classify_parse_entries()` function, add the following lines below the two UDP cases.

```
case FSPEC_TCPIP4:
    flow_spec_ip4_tcp_ioc(flow_entry_handle, port, chan,
flow_cfg);
    break;
case FSPEC_TCPIP6:
    flow_spec_ip6_tcp_ioc(flow_entry_handle, port, chan,
flow_cfg);
break;
```

3. Add the `flow_spec_ip4_tcp_ioc` and `flow_spec_ip6_tcp_ioc` functions to the file.

You can use `flow_spec_ip4_ioc()` and `flow_spec_ip6_ioc()` as a template. The only difference in the `ip4` case is the following three lines:

```
clsfy_ioc.flow_spec.fs_type = FSPEC_TCPIP4;
clsfy_ioc.flow_spec.ue.ip4.port.tcp.src = fe[i].src_port;
clsfy_ioc.flow_spec.ue.ip4.port.tcp.dst = fe[i].dst_port;
```

as opposed to:

```
clsfy_ioc.flow_spec.fs_type = FSPEC_UDPIP4;
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;
```

4. Do the same additions for `ip6`.
5. Open the `user_common.h` file.
6. Add the following function prototypes:

```
extern void flow_spec_ip4_tcp_ioc(void *flow_entry_handle, uint8_t port,
uint8_t chan, uint_t flow_cfg);
extern void flow_spec_ip6_tcp_ioc(void *flow_entry_handle, uint8_t port,
uint8_t chan, uint_t flow_cfg);
```

7. Open the `ipfwd.c` file:

8. Pass `FSPEC_TCPIP4` and `FSPEC_TCPIP4` options to `classify_parse_entries` instead of passing in `FSPEC_UDPIP4` and `FSPEC_UDPIP6`:

```
#ifdef IPV6
    classify_parse_entries(ipfwd_flow_config, port, chan,
                          FSPEC_TCPIP6, (void
*)&ip6_flow_tab[0]);
#else
    classify_parse_entries(ipfwd_flow_config, port, chan,
                          FSPEC_TCPIP4, (void
*)&ip4_flow_tab[0]);
#endif
```

9. Open the `ipfwd_flow.c` file:

10. Change all `IPPROTO_UDP` inputs to `ip4_flow_tab[]` and `ip6_flow_tab[]` to `IPPROTO_TCP` for all TCAM entries.

For example:

```
flow_spec_ip4_tab_t ip4_flow_tab[] = {
    {0, IPPROTO_TCP, 0, 0xFF, 0, 0xFFFF, 0, 0xFFFF,
     "192.30.50.0", "255.255.255.255",
     "192.31.50.1", "255.255.255.0",
     FLOW_ACCEPT, 0},
    .....
}
```

11. Recompile.

After that, you should be able to use the TCAM to parse for TCP packets.

How Do I Add a New Packet Type to `ntgen`?

Perform the following steps:

1. In the `apps/ntgen/src/common/protohdr.h` file, extend `struct gen_hdr_buf` with new header structures.
 - a. If new packet types introduce a new network layer protocol, change the `get_netproto_len()` function.
 - b. If new packet types introduce a new transport layer protocol, change the `get_tproto_type()` and `get_tproto_len()` functions.

2. In the `apps/ntgen/src/app/trace_buffer.c` file, modify the `fill_trace_buffer()` function so that the user supplied options are applied to the template packet to create traffic packets so that the trace buffer is filled.

3. Add modified headers in the logic flow that starts from the `modify_packet()` function.

If a template packet needs to be modified, `modify_packet()` is called. This function is the entry point for modifying different headers of a packet. A packet's headers is modified as Ethernet header first, network layer header next, and transport layer header last. New headers that need to be modified must be added in the logic flow that starts from this function.

4. If the new packet type introduces a new network protocol, add the handling for this protocol in the `process_net_layer()` function in the `apps/ntgen/src/app/parse_eth2.c` file.

5. Check if the new packet type uses IPv4 or IPv6.

If the new packet type introduces a new transport protocol, perform the following steps:

a. Open the `apps/ntgen/src/app/parse_ipv4.c` or `apps/ntgen/src/app/parse_ipv6.c` file.

b. In the `parse_ipv4()` or `parse_ipv6()` function, add the support for the new transport protocol.

If not the new packet type does not create a new transport protocol, the transport layer protocol header must be handled from the new network layer protocol handler that was added in Step 4.

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 <code>tejacc</code>	
<code>-O</code>	Enables all optimizations.
<code>-fcontext-sensitive-generation</code>	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 29](#) 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 Sun Netra DPS Application?

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

- [“Linking Legacy Code to Sun Netra DPS Code” on page 393](#)

- [“Changing Legacy Source Code” on page 393](#)

Linking Legacy Code to Sun Netra DPS Code

By linking legacy code to the Sun Netra DPS code as libraries, the legacy code is not compiled and changes are minimized. The legacy library is also linked to the Sun 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 Sun 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 Sun 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 Sun Netra DPS calls as only performance-critical parts of legacy code need to be ported to Sun Netra DPS. Error handling and exception code can remain unchanged.

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

Note – See [“How Can I Reuse Legacy C Code in a Sun Netra DPS Application?” on page 392](#).

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

- [“Mixing C and C++ Code” on page 394](#)
- [“Translating C++ Code to C Code” on page 394](#)

Mixing C and C++ Code

Sun Netra DPS generates C code, so the final program is in C. Mixing C++ and Sun 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 Sun 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 Sun Netra DPS code. The Sun 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://javashoplmsun.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 0x11100000 to 0x11200000, and so on. Heap memory is used by <code>teja_malloc()</code> .
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 395.

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 Sun 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) (0xffffffff00)
```

2. Compile the RLP application with ARP=on.

```
$ make clean  
$ make ARP=on
```

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

Sun 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 Sun 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 [Chapter 11, “Reference Applications” on page 163](#)).

Oracle Solaris Domain and Sun Netra DPS Domain Question

How Do I Access `kstat` Information From the Oracle Solaris Domain for Network Interfaces That Are in Use by the Sun Netra DPS domain?

This feature is available on the IP packet forwarding application (`ipfwd`). On the Oracle Solaris domain, use the following command line to access `kstat` information:

```
# kstat tnxge:0
module: tnxge                instance: 0
name:   Port Stats          class:   net
        crtime              2975750.16388507
        ipackets            6
        obytes              384
        opackets            6
        rbytes              384
        snaptime            3145512.6135888
```

To enable statistics in the `ipfwd` application, edit the `Makefile.nxge` file and uncomment the `-DKSTAT_ON` flag.

Traffic Generation

How Do I Stop Traffic Generation?

If Oracle's Sun Netra DPS application is in an unrecoverable state, then a single Ctrl-C might not exit the user interface application. In that case, pressing Ctrl-C four times will exit the user interface application and the Sun Netra DPS application can then be restarted from the primary domain by restarting the Sun Netra DPS domain.

Oracle Solaris TIPC Application

What Should I Do When the Oracle Solaris TIPC Application Is Not Able to Create a Socket and Does a Core Dump?

The TIPC socket library should be preloaded before running the Oracle Solaris TIPC application. Refer [“Installing TIPC” on page 155](#) to setup an environment to preload the library.

Glossary

A

- ADE** Sun Netra DPS Eclipse-based Teja Advance Development Environment (ADE) graphical user interface. Teja ADE views three Teja elements, hardware architecture, software architecture and mapping.
- AF** Assured forwarding.
- 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.
- CBS** Committed burst size.
- CG** Cipher group.
- CIR** Committed information rate.
- CLI** Command-line interface.
- 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. Sun Netra DPS native debugger is the default debugger and is useful for debugging during development.
- DMA** Direct memory access.
- DRR** Deficit round robin.
- DSCP** Differentiated services code point.

E

- EBS** Excess burst size.

- Eclipse** An open source community where projects are focused on building extensive development platforms, runtimes, and application frameworks.
- EF** Expedited forwarding.
- ESP** Encapsulating security payload.
-

F

- FIB** Forwarding information base.
-

G

- GCCfss** GNU C Compiler for Sparc Systems.
- GDB** GNU debugger that enables you to debug your program in C source code level.
- GRE** Generic Routing Encapsulation application.
- GUI** Graphical user interface.
-

I

- IP** Interprocess, as in IP addresses.
- IPC** Interprocess communication software mechanism that provides a means to communicate between processes that run in a domain under the Sun Netra DPS Runtime Environment 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.
- 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.
- Sun Netra DPS** Sun Netra Data Plane Software Suite. In this document, this suite is also referred to as Sun Netra DPS.
- Sun 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.
- NTGen (ntgen)** Sun Netra DPS traffic generator tool.
- 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.
- PHB** Per hop behavior.
- PIR** Peak information rate.
- producers** Threads sending messages to a channel.

Q

- QM** Queue Manager.
- QoS** Quality of Services.

R

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

S

- SA** Security Association.
- SAD** 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.

SP	Security Policy.
SPD	Security Policy Database.
SPDC	Security Policy Database Cache.
SPI	Security Parameter Index.
SPU	Stream Processing Unit.
SRTCM	Single Rate Three Color Marker.
strand	A hardware thread, multistrand partitioning firmware for Sun CMT platforms.
<code>SUNWndps</code>	Sun 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.
<code>SUNWndpsd</code>	Sun Netra DPS software package installed on the target deployment system. Contains the Sun Netra Data Plane CMT and IPC Share Memory Driver.
<code>SUNWndpsc</code>	Sun Netra DPS software package containing the Sun UltraSPARC T2 cryptography driver.

T

TCAM	Tertiary content addressable memory.
TCP	Transmission control protocol.
TC	Three color meter.
<code>tejacc</code>	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.
TIPC	Transparent Interprocess Communication protocol.
<code>tnsmctl</code>	Contained in <code>SUNWndpsd</code> package and contains the Sun Netra Data Plane CMT and IPC share memory driver.
TOS	Type of service.

TRTCM Two-rate three color marker.



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

A

access control list (ACL) reference application, 247
autoconfig tool, 143

B

basepaddress, calculating, 388
boot an application image, 12
building reference applications, 11

C

command-line options, tejacc, 33
common header file, 99
configuring IPC environment, 90
context-sensitive generation, 36

D

debugger configuration code, 70
debugger, GDB, 80
debugger, native, 70
diagnosing network applications, 333

E

Eclipse GUI, 109

F

FAQ, 371
file contents, software, 3
finite state machine API, 31
firmware
 checking version, 5
 installation, 4

frequently asked questions, 371

G

GDB debugger, 80
Generic Routing Encapsulation (GRE), 236

H

hardware architecture overview, 19

I

interprocess communication (IPC), 89, 102
IP packet forwarding
 ipfwd, 164
IPC
 configuring environment, 90
 overview, 89, 102
IPC channels, 97
ipfwd, 164
IPSec gateway reference application, 257
IPv4 and IPv6 packet forwarding
 ipfwd, 164

L

late-binding API, 29
late-binding elements, 24
logical domains environment, 91

M

map API, 31

N

Netra DPS Runtime API, 29
network interface unit (NIU), 123
NIU (network interface unit), 123
ntgen, 283

O

optimization options, 35
overview, 1

P

profiler, 39
profiler API examples, 44
profiler script, using, 53
programming methodology, 13

Q

questions, FAQ, 371

R

radio link protocol (RLP), 252
Receive Packet Classifier, 123
reference application instructions, 11
remote command-line-interface (CLI)
 accessing, 104
 coredump support, 106
 debugging remotely, 105
 introduction, 101
 IPC setup, 102
 system configuration, 106
RLP (radio link protocol), 252

S

software
 file contents, 3
 installation, 3
 package contents, 3, 4
software architecture and late-binding overview, 23
SUNWndps and SUNWndpsd package contents, 3, 4

T

tejacc basics, 33
tejacc compiler basic operation, 15
tipc-config tool, 153
traffic generator

ntgen, 283

transparent interprocess communication
(TIPC), 153

tuning network applications, 333

tutorial, 363

U

UltraSPARC T1 processor, 334
UltraSPARC T2 processor, 336
UltraSPARC T2, example environment, 98

V

virtual data plane channels, 96