Looking for the Lost Packets
Techniques for Debugging Packet Processing Systems based on Multi-core Intel Architecture Processors

December 2009
Looking for the Lost Packets
Executive Summary

This paper describes some of the typical problems engineers face while debugging packet processing systems, along with a collection of techniques applied successfully in the past to nail down the root cause of such problems.

Due to the complexity of these systems, it is imperative they be designed with debug capabilities from the start, as opposed to ignoring this aspect and later on face the problem of having no real means of investigation.

The techniques described in this paper represent a collection of built-in mechanisms to be provisioned from as early as the design phase to assist the system debugging at run-time, with the purpose of detecting any system errors from the first stages of development and testing.

After a quick presentation of the typical packet processing system under debug, we focus on describing the functional, stability and performance problems that are generally faced, along with some hints for the possible root cause of the problem and associated debugging techniques.
## Contents

Introduction .................................................................................................................. 5

The Typical System under Debugging ................................................................. 5

Functional Problems ................................................................................................. 6
  Problem Description ............................................................................................... 6
  Combat Techniques ............................................................................................... 7
    Statistics, Statistics, Statistics ............................................................................ 7
    Command Line Interface .................................................................................. 8
    Debug Messages ............................................................................................... 8
    Queue Monitor ................................................................................................. 9

Stability Problems .................................................................................................. 10
  Problem Description .............................................................................................. 10
  Hints for Possible Root Causes ........................................................................... 11
    Change in the typical sequence of events ......................................................... 11
    Flawed state machines .................................................................................... 11
    Deadlock ........................................................................................................... 11
    Resource exhaustion ......................................................................................... 11
    Incorrect assumptions ...................................................................................... 12
    Memory corruption ........................................................................................... 12
    Incomplete handling of the returned error codes ............................................. 12
  Combat Techniques .............................................................................................. 13
    Static Code Analysis ......................................................................................... 13
    Run-time Monitor/Logger ................................................................................ 13

Performance Problems ........................................................................................... 14
  Problem Description ............................................................................................. 14
  Hints for Possible Root Causes .......................................................................... 14
  Combat Techniques ............................................................................................. 14

Conclusion ................................................................................................................ 15
Introduction

This paper describes some of the typical problems engineers face while debugging packet processing systems, along with a collection of techniques applied successfully in the past to nail down the root cause of such problems.

Due to the complexity of these systems, it is imperative they be designed with debug capabilities from the start, as opposed to ignoring this aspect and later on face the problem of having no real means of investigation. The techniques described in this paper represent a collection of built-in mechanisms to be provisioned from as early as the design phase to assist the system debugging at run-time, with the purpose of detecting any system errors from the first stages of development and testing.

Debugging this type of system is not a closed topic, so this paper is not intended to be an exhaustive list of recipes that when applied will guarantee your success. Most solutions have been discovered the hard way and now shared with the hope they will save some of the time engineers spend debugging, an activity that no one seems to enjoy, but everyone seems do it sooner or later during the project. So if you have had similar problems in the past and you are unable to find your problem and solution described here, we would certainly like to hear your story from you!

After a quick presentation of the typical packet processing system under debug, we focus on describing the functional, stability and performance problems that are generally faced, along with some hints for the possible root cause of the problem and associated debugging techniques.

The Typical System under Debugging

The packet processing systems are generally complex systems, so debugging is not easy. If it was easy, nobody would have paid any attention to this paper, right?

Nowadays, these systems contain multi-core processors with several on-chip and off-chip configurable acceleration engines and hard-wired logic blocks, all of them performing under strict real time requirements to achieve the expected throughput rate for the traffic flows they are designed to support.

Typically, the role of these systems is to receive packets from the network or the backplane interfaces, map each packet to one of the known connections (packet classification), process the packet in accordance with the connection requirements and return the packet back to the network or the backplane.
One key aspect to remember is that even if the design is processor-centric and you are just writing software on the multi-core processor, you have to debug the whole system, not just a single device in isolation.

The diagram below illustrates a packet processing system with at least four Gigabit Ethernet ports based on the Intel® Xeon™ Processor 5500 Series. The system consists of two Quad-Core Intel Xeon Processor 5500 Series processors supporting the simultaneous multithreading technology, thus bringing the number of hardware threads in the system to 16. The speed of the Intel® QuickPath Interconnect bus is 6.40 GT/s. Each processor supports three channels of DDR3 memory. Each Intel® Gigabit ET Quad Port Server Adapter supports four Gigabit Ethernet ports and is connected to the system through the PCI Express Gen2 bus.

**Figure 1 Packet processing system based on two Quad-Core Intel® Xeon® Processor 5500 Series processors**

---

**Functional Problems**

**Problem Description**

Generally, the functional problems can be easily reproduced by hooking up a traffic generator to the system in the lab and sending just a few packets, ideally just a single packet. This is why people call them deterministic problems, as every time you run your simple experiment the system behaves the same way, as opposed to the
stability problems which often need a variable sequence of operations and input packets during a significant amount of time in order to be triggered.

As the problem can be reproduced with a few or even a single packet, you are able to take your time to analyze the state of the system in detail after each packet. You can even afford the luxury to use a step-by-step traditional debugger and follow your packet step by step through the processing pipeline as your breakpoints are getting hit.

Some good examples of such problems include receiving the output packets malformed (some fields are different than expected), or on the wrong output interface (routing problem), or not coming out at all (they are getting dropped somewhere along the processing pipeline).

Combat Techniques

Statistics, Statistics, Statistics

Implement statistics counters for each possible exit out of the processing pipeline. An even better solution would be to implement them for each block of the pipeline. The statistics counters should be designed so that no packets will ever get "lost", at the pipeline level or at each block level: the number of input packets should always equal the number of output packets plus the number of discarded packets. The statistics counters seem to be the Swiss army knife of debugging the packet processing systems.

For example, let’s assume that pipeline block A can send the input packet out to block B or block C after processing, or discard the packet as result of any of the conditions D, E or F being met. If each of these paths would have a counter associated with it that gets incremented every time a packet takes that path, then the packets can be traced down through the pipeline regardless of whether block A is working correctly or not. Remember, you can send just one packet at a time, so by looking at these counters you can easily find out what happens with each packet that is injected into the system by the traffic generator.

It is very inefficient to find out while debugging that you are unable to trace all the packets through the system because not all of your code paths have been consistently instrumented. Basically, some packets got lost somewhere within the system and you have no way of knowing where exactly this took place and why. One common mistake people make is to instrument just the main path and forget about the secondary paths through the pipeline. Usually the problems occur exactly on these paths which get frequently overlooked, as people focus on the functionality and performance of the main path.

The drawback of this approach is that too many statistics counters can affect the performance of the system due to memory accesses that are involved. Some people would say that this yet another manifestation of the famous Heidegger incertitude principle, according to which the state of the system cannot be observed without modifying the state of that system. The simple workaround is to conditionally compile
these counters into the code, so that they be enabled during the functional debugging and disabled for performance reasons after the functional testing has been completed.

Do not start optimizing for performance before all the functional problems have been solved.

Make sure that the system has passed all the functional testing before focusing on performance optimizations, otherwise all the optimization work is useless. The optimization would have to be completed all over again after the functional problems have been discovered, investigated and fixed. Remember, no networking box will ever be accepted for deployment if functionally flawed, no matter how amazing the performance numbers for some of the paths may be.

**Command Line Interface**

Building a Command Line Interface (CLI) into your application allows the user to interact with the system at run-time without having to stop the system under debugger control to inspect its state. Most of the modern networking equipments provide this feature for system configuration (e.g., for listing, adding or deleting entries from application tables such as the routing table or the ARP table), so your application should provide it as well.

As long as the CLI is in place for system configuration, why not add support for system debugging into the CLI as well?

When building the CLI, the commands should allow:

- Inspection of the statistics counters built into the application
- Inspection and modification of memory addresses and device registers, allowing access to the tables and data structures maintained by the application, packet descriptors and packet buffers and to any memory address in general
- Using other debugging tools described later on in this paper

The benefits of the CLI are that it allows inspecting the system state quickly and it provides a single entry point for accessing all the debugging tools at hand. Ideally, you send a packet and then just hit the keyboard once, and the full state of the system is conveniently displayed for you.

**Debug Messages**

There is probably no need to stress how useful for debugging the `printf` messages are, especially in the early stages of development. They do slow down the system and they will never make it to the release code, but general consensus among engineers is that they are very valuable for debugging.
Sometimes the printf mechanism is not readily available, usually on some special purpose processor cores that are part of the programmable accelerators, which do not have access to any output port to send the messages to. The software running on these cores, usually firmware, is very difficult to debug when lacking any mechanism that allows getting messages out of these cores.

The workaround is to emulate printf by assigning a memory buffer to these cores that they can use to log data in a predefined format, either statistics counters or strings. Another general purpose core can periodically poll this buffer, read the data, decode it and print it on the screen, or the user can trigger a new read of the log buffer through the CLI.

**Queue Monitor**

Generally, the blocks of the packet processing pipeline are interconnected through queues of messages, where a message can be a packet descriptor, a request message or a response message.

It would be very useful during debugging to inject one packet at a time and trace the path of that packet through the various blocks of the pipeline by determining which queues have been transited by the packet in question. The possible cases for the state of each queue after the packet transition through the system are:

- The write pointer has not been modified: the packet was not written to this queue
- The write pointer was incremented, but the read pointer is unchanged: the packet (or a message associated with it) was written to the queue, but not read from the queue, so there is a problem with the consumer of the queue, as it does not read the messages from its input queue
- Both the write and the read pointers have been incremented: the packet did transit through this queue, as it was written to the queue and also read from the queue later on

The steps to build the queue monitor are:

- Initialize the monitor: read the initial values for the write and read pointers for all the queues
- Send one packet and call the monitor: read the current values for the write and read pointers for all the queues and diff them against their previous values to identify which queues have been transited by the packet and which not

The monitor initialization and call are usually implemented as CLI commands. The monitor call command prints out the IDs of the queues that have been transited by the packet. Once the monitor has been initialized once, there is no need to initialize it again.
Looking for the Lost Packets

again for every new packet that is injected, as the current values of the queue
pointers are already known as result of reading them for the previous packet and
thus these values become their initial values for the new iteration. The monitor needs
to be reinitialized only after one or more packets have been injected into the system
without calling the monitor, as the previously read values for the queue pointers are
now obsolete.

For some scenarios, several messages are created written to the same output queue
for the same packet. For example, the IP fragmentation block might produce more
than one output packet for the same input packet. If not all the messages are read
from its output queue by the next consumer block in line, then there might be a
problem with the consumer block.

For some other scenarios, it is meaningful to inject more than one packet between
two consecutive monitor calls. One example is the IP reassembly scenario, where
several IP fragments have to be received before the IP reassembly block produces
one output packet to its output queue.

Stability Problems

Problem Description

Stability problems are the most difficult to debug, as they represent problems that
cannot be triggered with basic low-rate functional tests. They need stress tests close
to the real traffic conditions the device has to handle during real-life scenarios.

These problems usually require high traffic input rate and a long time before they
activate (minutes, hours or even days), so by that time many millions of packets
have already transited through the system with no errors, thus making it impossible
to analyze the system operation on a packet-by-packet basis. Sometimes it is not
even possible to reproduce them with the equipment in the lab, having to attempt
debugging the system while it is live in the field. These problems usually hide a flaw
that is not fatal, meaning that the system is able to run correctly for some time
before their impact cripples down the system.

The traditional debugging techniques do not work well for these problems. Running
one of the cores under debugger control may be defeated by the fact that, while one
of the cores is stopped as result of hitting a breakpoint, the rest of the system
continues to process packets that keep coming in and modify the state of the system
as the system continues to process the packets. This renders the state of the system
no longer relevant for inspection because the system cannot keep up the pace with
the input stream, which leads the packet queues to overflow and ultimately to a
corrupted system state.

As these problems are difficult to reproduce on low data rates, debugging under
heavy traffic loads becomes a necessity.
Hints for Possible Root Causes

Change in the typical sequence of events

Corner cases that are not triggered for low rates may now be triggered. The sequence of events and actions that take place while processing the packets is different than the typical sequence. Race conditions you were not aware of are now awakened.

Flawed state machines

The implementation of the state machines that are so commonly used for packet processing may be flawed. There are cases that are not robustly handled by the state machines, which now reach different states than expected.

In the worst case scenario, the state machines may have to be massively cleaned up or even completely redesigned to remove the complexity that is not required.

Optimize your state machines for simplicity and readability, not for (often questionable) performance gain.

Deadlock

Synchronization between the producers and the consumers of the same queue may be flawed, making the producers think the queue is full or the consumers think that the queue is empty. Other possible root causes may be the incorrect usage of semaphores or other synchronization primitives, or waiting for an event that never takes place.

Resource exhaustion

A critical resource is exhausted and never replenished as result of incorrect usage, leading to incorrect operation. One of the most common scenarios is the permanent exhaustion of the buffer pools as result of buffer leakage. Buffers are allocated from the pool, but not all of them are correctly released back to their pool. These buffers are practically lost, as the software simply "forgot" about them, so it does not longer make use of them.

As result, the buffer pool shrinks over time, as fewer and fewer buffers are available in the pool at any given time which leads to performance degradation over time, as more and more input packets cannot be accommodated by the system and have to be dropped. The buffer pool eventually becomes empty and, as the pool is never replenished with buffers, no more packets will ever get out of the system.

If the code branches that leak buffers are frequently hit, then the problem is triggered relatively quickly. What usually happens is that the leaky code branches are
infrequently hit, as they typically handle some infrequent error cases, so it takes a significant amount of time to reach the buffer pool exhaustion.

For example, consider a system which has a leaky ARP table aging process that runs once every 120 seconds leaking one buffer on every run. If the buffer pool initially contains 1K buffers, then the system has to run non-stop for about 34 hours until the buffer pool exhaustion is experienced and the output cut-off takes place.

Because people tend to wait for the later stages of the project before applying stress tests like running the system under heavy traffic non-stop for a few days in a row, this explains why this problem is usually discovered so late into the project.

A related problem is represented by an under-dimensioned buffer pool, which causes periodic packet drops as the pool reaches exhaustion. As opposed to the previous problem, the pool does get replenished correctly with previously allocated buffers and the output traffic does resume eventually, so this is a performance tweaking problem rather than a stability one.

**Incorrect assumptions**

The software is not considering the case when a message sent to a queue is dropped due to the queue being full, assuming that the delivery of the message is guaranteed. Consequently, if the software is relying on the consumer actions associated with the handling of that message to be performed in order to continue working correctly, then it operates based on false assumptions. Same if the software is relying on receiving back a response to this message, as the response never comes since the request message never made it to the other side.

**Memory corruption**

Some local variables might be left uninitialized before their value is read and used by the application. The software might not consider the fact that reading from a message queue can fail due to the queue being empty (queue underflow) or writing to a message queue might fail due to the queue being full (queue overflow). The software might incorrectly attempt to read/write more data from/to a buffer than allowed by the buffer size. As result, inconsistent data is read or incorrect memory addresses are written, leading to the data structure corruption problem that is so hard to debug.

**Incomplete handling of the returned error codes**

The software might not be handling all the possible error codes returned by the called functions, either by assuming success or by ignoring some of the error codes.

Make sure that all the possible values for the return codes are handled.
Combat Techniques

Static Code Analysis

Make sure that all the local variables are initialized and all the return codes are handled.

Run-time Monitor/Logger

Do not let the system errors go unnoticed!

The first step is to instrument the code with statistics counters tracking the resource usage and the various error conditions that can take place and make them available through the CLI. Examples of relevant counters are:

- Number of free buffers for each buffer pool
- Queue occupancy for each queue
- Current position for all the semaphores in the system
- Number of free entries within the various tables maintained by the application (e.g., routing table, ARP table)
- Number of DMA errors and retries
- Bus occupancy (can be calculated as the number of transactions multiplied with the length of each transaction since the last monitor invocation)

The second step is to implement a run-time monitor. This represents a callback function that is periodically invoked on timer events and checks for some of the most common error conditions that can take place in the system and, if any such condition is met, trigger the corresponding alarm. Examples of relevant error conditions that may seriously impact the system operation:

- One or more buffer pools are consistently empty or almost empty
- Some queues are consistently almost full (occupancy more than 90%)
- Specific queues have been written but not read in the last period of time (consumer deadlock)
- Specific semaphores busy during the last number of monitor invocations
- Some tables maintained by the application are full or almost full
- The buckets of some hash tables became too long (possible indication that the hashing function is not efficient)
- The number of DMA errors or retries are above their acceptable threshold
- Bus occupancy dangerously close to the upper limit
The run-time monitor can also log the full state of the system for further static analysis, as a graphical representation over time can uncover some less obvious problems. The monitor should be disabled or reduced to a bare minimum during normal regime as it eats computing cycles.

**Performance Problems**

**Problem Description**

The statement for the performance problems is that the output packets are correct and the system has been proven stable, but the performance metrics like the throughput rate or the latency do not meet the expected targets for some of the supported traffic scenario.

As the system is stable, it is arguably preferable to tackle performance tweaking problems than debugging stability problems. At least the performance problems are usually straightforward to reproduce. Some people would state that, once you have reached this stage, you can relax a little bit.

**Hints for Possible Root Causes**

If the performance problem is massive, then it might hide a functional or stability related problem. It may also be a design problem.

Design with the targeted performance numbers in mind from design day 1!

Otherwise, the performance tweaking problem might be caused by the incorrect dimensioning of resources when matched against the performance requirements.

**Combat Techniques**

Revisit the design, if need be.

Instrument the code with time stamps and profile the application to identify the problem regions that are candidates for optimization. Do not optimize those blocks that do not have a major contribution to the overall performance problem of the system. In theory, the blocks to pick for optimization should be those 20% of the blocks that use up to 80% of the packet budget.

Analyze the data collected by the real-time monitor looking for performance bottlenecks.
Overlooking some of the platform related architectural considerations might also have a negative impact on the system performance:

- **Cache line**: On Intel Architecture processors, the size of the cache line is 64 bytes. Try to maximize the number of cache hits by minimizing the number of cache lines that have to be used for storing your data structures. For example, if one data structure is less or equal to 64 bytes in size, by allocating it in memory on 64 byte aligned addresses, it will fit into a single cache line rather than spanning across two cache line. If two cache lines are used to store a structure which normally fits into a single cache, then the cache hit probability for accessing an instance of this data structure is cut in half.

- **Zero copy**: Make sure that the packet does not have to be copied from one buffer to another in your design, as the memory copy is expensive.

§

**Conclusion**

This paper presented several techniques for debugging functional, stability and performance problems related to packet processing systems.
Looking for the Lost Packets

Authors

Cristian F. Dumitrescu is a Software Engineer with the Intel Architecture Group.

Acronyms

ARP  Address Resolution Protocol
CH   Channel
CLI  Command Line Interface
DMA  Direct Memory Access
DDR  Double Data Rate
DDR3 Third Generation DDR
Gen2 Second generation
GT/s Giga Transactions per Second
IP   Internet Protocol
MAC  Media Access Control
NIC  Network Interface Card
PCI  Peripheral Component Interconnect
PCIe PCI Express
PHY Physical Layer Device
QPI  Intel QuickPath Interconnect
SMT  Intel Simultaneous Multithreading Technology

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification.