UACEE International Journal of Computer Science and its Applications - Volume 2: Issue 1 [ISSN 2250 - 3765]

Debug and Optimization Techniques for Performance Enhancement of Termination Application in Real Time System

Sandeep Malik
NMG
Freescale Semiconductor Ltd
Noida, India

Yashpal Dutta
NMG
Freescale Semiconductor Ltd
Noida, India

Abstract— **Due to the rapid increase in network services per network node, the need for high throughput infrastructure is growing. Throughput and latency are two important criteria that need to be considered while working on Real Time Embedded Systems. With the advent of Real Time Operating systems, low latency can be achieved, but sometimes at the cost of performance. The situation becomes complex if the system has a networking termination application in the user-space operating with the kernel space driver.**

The paper presents how changes in Real Time Linux impose complexity in a network application both in kernel and user space. This paper also discusses ideas to identify problems and fixes available using Linux debugging tools. This paper concludes by sharing performance improvement data achieved in a user space networking application.

Keywords— **Performance, Real Time, Linux, Femto Cell, Termination, Networking, Debugging, User Space**

I. Introduction

With the increased Internet usage and the availability of high network bandwidth, more and more IP traffic flows per network node. Along with high traffic throughput requirements, systems also demand low latency for certain services like the Femto Cell market. The data traffic processing in a Femto Cell use case can be thought of as two parts. One part interacts with the wireless side of the network and the second part interacts with the wire-line side. The wireless part of network has critical time line requirements that need to be met, whereas the wire-line side has throughput requirements. The system needs to address both these timing and throughput requirements. The low latency requirements can be met using the Real Time System (Linux with RT patch in our case). The RT patch has a big impact on throughput and the system needs to be tweaked to obtain the best of both latency and throughput.

The usage of Linux with RT patch has been available for quite some time. The motivation to write this paper arose when one user-space networking application running on native Linux was being ported to Linux with Preempt RT patch. After porting the user space application, the overall performance dropped by more than 50%.

This paper discusses the debug techniques used to determine the reason for the system performance drop and attempts at solutions.

To validate the design approaches, *Freescale's P2020[4]* platform was used. This platform was configured for Power Architecture Technology core running at 1 GHz with System Bus running at 500MHz and DDR operating at 400MHz. The system used Linux 2.6.33 with the RT patch applied. This paper also explains the incremental performance enhancements that were observed when these design approaches were tested on the Freescale P2020 platform.

Let's start with an overview of the system from a software perspective, which indicates the components involved in the packet processing path for a user space termination use case. Here the preferred system is receiving encrypted packets from the IPSec tunnel in the network, which after decryption needs to be provided to the user space for further processing. The user space application, after processing, provides these packets to the kernel space for one more level of security operation to be performed before transmitting the packet to network.

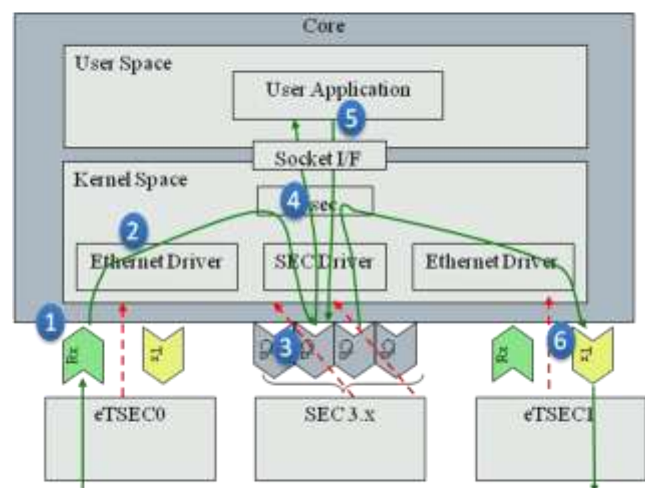

Figure 1. System overview from software prespective

The system has following the processing units:

1) Ethernet Rx interrupt handler (Red Dotted Line).
2) Ethernet bottom half running in netif_rx SOFT_IRQ.
3) SEC(Security Block) Interrupt Handler.
4) SEC Bottom half running in netif_rx SOFT IRQ.
5) User space application responsible for receiving the packets from kernel space post-decryption processing and giving it back to kernel space.
6) Ethernet Tx completion interrupt handler for cleaning up of Tx queue.

In this system the user space process is supposed to run at the highest priority so that whenever there is some job to be done by the user space, it will preempt other processing units and start processing the packets.

II. Debug Techniques

Debugging is broadly categorized in two classes:

1) Functional Debugging (Static): Mainly used to debug and fix issues, if any, in the system functionality.
2) Performance Debugging (Dynamic): Mainly used to identify and fix issues in system performance. This includes debugging latency and throughput.

This paper discusses Performance Debugging. Profiling and tracing support for Linux kernel can be used to debug latency and performance issues. The following section covers the available tools in Linux that can be used for performance debugging.

A. RT-Test Suite

This suite consists of multiple utilities. Cyclic-test, which is a part of the RT-Test suite, can be used to determine RT capability of a given system. The presence of IO activities or a specific stress condition with cyclic-test can be used to check worst-case latency.

Cyclic-test clubbed with "-b" option can be used to fight high latencies. If latency exceeds the threshold provided with "-b" option, function tracing is written to trace ring-buffer and cyclic-test aborts. This function traces data can further be used to identify the cause of high latency.

B. Ftrace

FTrace is a tracing utility directly available in Linux under "Kernel hacking" configuration option. Ftraces provides features to assist tracking down Linux kernel problems. Kernel tracing capability using ftraces helps in understanding internal kernel activities. If enabled, tracing directory is available in debug file-system. In addition to functional debugging, ftrace helps in latency analysis. Kernel provides configuration options to enable ftraces[2]. Various tracing options can be enabled in Linux configuration and can be seen from "available_tracers" file in tracing directory.

C. OProfile

Analyzing the performance impact of system loading on application code can be done with profiling. Oprofile is one such profiling tool-set for kernel and application profiling. Oprofile on the Powerpc architecture helps identify issues like poor-cache utilization, branch mis-predictions, low IPC, and L1/L2 TLB misses using the core's Performance Monitor support. Profiling support [CONFIG_PROFILING and CONFIG_OPROFILE] must be enabled in Kernel configuration for Oprofile[6] to work.

III. Analysis of Preempt RT Patch Impact

Using the above-mentioned debugging techniques, the RT System was profiled. The analysis of the profiling data for the RT and Non RT system revealed the following findings.

A. Interrupt handlers being converted to Kernel Threads

The RT-Preempt patch in Linux converts the interrupt handlers to kernel threads to ensure that the real time timelines for a process have been met. Due to this change, scheduler invocation is required even to schedule the interrupt handlers Kthreads i.e. Top Half, which in non-RT case are invoked from the same context in which the system is executing. In a heavily loaded system, most of the time the interrupt processing keeps on getting preempted by other interrupts. This additional preemption eventually causes context switch and thus adds to latencies.

B. Increased overhead of context switching

With the introduction of Preempt RT patch in Linux, most of the kernel code has become preemptible. The profiling data shows that context switching overhead from user-space to kernel space takes around 3x more cycles then a similar Non RT based implementation. This overhead is mainly due to the checks required in the scheduler to maintain the real time behavior of the system.

When referring to the context switch from kernel space to user space, the reference solution is a termination application using the socket interface for communication with kernel. In this socket interface, the type of socket used is RAW sockets and PACKET_MMAP option is used for zero copy implementation.

IV. Solutions for These Problems

This section shares design techniques that can be used to address the above issues.

SEEK DIGITAL LIBRARY

A. *Making the interrupt handlers as non kernal thread*

While doing the performance analysis using *oprofile[6] and ftraces[2]*, it was found that lots of CPU cycles were consumed due to the interrupt handlers getting preempted frequently. Due to the preemption, the ISR (Top Half) often didn't even get a chance to schedule bottom-half so the bottom half processing was getting delayed and the system had an unnecessary overhead of context switching without doing any fruitful work. Apart from the context switch, there is another overhead caused by the need of invocation of scheduler to schedule the ISR kernel thread. This additional overhead of scheduler invocation can be avoided for selective interrupts which have short ISR and are called very frequently.

The overhead of context switching and scheduler invocation due to the interrupt handlers being executed as kernel threads was overcome by replacing these kernel threads with Hard ISR's. The hard ISR's can't be preempted by other kernel threads in the system and can be invoked from the same context in which the system is executing when the interrupt occur. This exercise helped improve throughput by about 15%.

B. *Avoiding the context switch overhead*

On further analysis of the context switch overheads, it was observed that the changes introduced at the time of context switching by Preempt RT patch were necessary to guarantee real-time responsiveness. Hence, to resolve the performance issue, we focused was on reducing the number of context switches in the system.

The kernel network driver was designed to use NAPI to minimize the interrupts overhead but the interrupts in the system were still affecting the system performance. To overcome this apart from NAPI; interrupt coalescing was enabled in the hardware. Benchmarking results show there is around *30%* performance enhancement in the system throughput when this approach of combining the interrupts together clubbed with NAPI.

After analyzing the profiling data, it was seen that the process executing in kernel space used to wake up the process running in user-space every time there is a processed packet available. This causes the user space process to wake up and to pick up one packet and give the same to the kernel via socket interface. This waking up of user space process for every packet caused the context switching from kernel space to user space for every packet entering the system. This was a big overhead and needed to be resolved to address the performance issue

To resolve this, a NAPI like mechanism was implemented in the kernel processing part where the processed packets are accumulated and once either a certain threshold amount of packets are processed or a certain amount of time has elapsed, the trigger was not given to the user space thread. Similar implementation was done at the user level processing unit it receives packets from kernel space but does not give the packets back to the kernel space task until a certain number of packets are collected by user space task.

Benchmarking results show that there is around *30%* performance enhancement in the system throughput when this approach NAPI like implementation was done at user space as well as kernel space.

Thus after implementing all the above techniques the system throughput increased more than 100% and the overall system throughput got better both in terms of bandwidth as well as overall response time of the system.

V. Conclusion

Various techniques to debug and improve the system performance of a network packet processing system that involves communication between user space process and kernel space drivers, and evaluates them on a dual core platform P2020 from the Freescale QorIQ P2 platform series have been discussed.

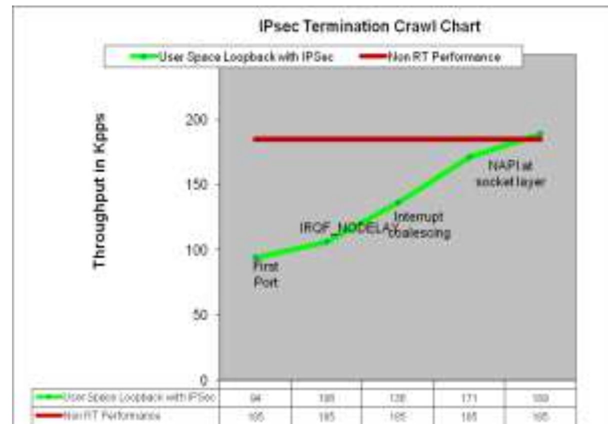

Figure 2. Performance scaling chart

Figure-2 shows the performance scaling chart for the reference system after implementing these ideas on preemptive RT kernel.

Using the design approaches explained in this paper, the system packet termination throughput can be improved by around *70-100%.*

References

[1] https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO+rt.wiki.kernel.org+RT_PREEMPT_HOWTO

[2] Ftrace - http://lwn.net/Articles/290277/

[3] A realtime preemption overview - http://lwn.net/Articles/146861/

[4] Freescale P2020 QorIQ processor: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P2020.

[5] NAPI (New API) - http://www.linuxfoundation.org/collaborate/workgroups/networking/napi.

[6] Oprofile – http://oprofile.sourceforge.net