Cases Study of the Performance of Real-Time Linux on the x86 Architecture

Qingguo Zhou
Network Center, Lanzhou University
Tianshui South Road 222, Lanzhou, Gansu, 730000 P.R.China
zhouqg@lzu.edu.cn

Baojun Wang
Ginwave Company
Shenzhen, P.R.China
wangbaojun@ginwave.com

Nicholas McGuire
Opentech GmbH
Lichtenstein Str.31 A-2130 Mistelbach Austria
der.herr@hofr.at

Abstract

This paper presents the experimental study of the performance of Real-Time Linux based on the x86 architecture. Holding the hardware constant and using different measurement methodologies, we measured a list of performance (ten kinds of test), including the overheads incurred during operating systems context switching test, conditional variable test, rt-fifo read/write test, mbuff read/write test, etc. Our experiments and analysis show that Real-Time linux provides good raw performance and it is suitable as an operating system platform for developing and running soft and hard real-time applications.

1 Introduction

Unlike Linux, RTLinux provides hard real-time capability. It has a hybrid kernel architecture with a small real-time kernel coexists with the Linux kernel running as the lowest priority task. This combination allows RTLinux to provide highly optimized, time-shared services in parallel with the real-time, predictable, and low-latency execution. Besides this unique feature, RTLinux is freely available to the public. As more development tools are geared towards RTLinux, it is becoming more popular in the real-time application domain.

What is real-time? How about the performance of RTLinux? There are several definitions and uses of the phrases 'real-time operating system' and 'real-time performance'. In this article we just take into account hard real-time. Hard real time is that the entire application fails if it misses even one deadline. Deadlines must always be met and worst-case delay is of the utmost importance. In this article, We do the case study in these key metrics: scheduling jitter, conditional variable, context switch, rt-FIFO, mbuff, semaphore, mutex, contested mutex, etc. we hope that this paper could be intended to serve as a guide when evaluating real-time operating systems and the claims made by their vendors. The test suite described in this paper could run on any x86 based system and can be extended to more complicated tests as required.

The rest of the paper is organized as follows. In Section 2, we describe the test environment for evaluating the various components of the OS latency, and we formally define performance metrics for measurement and present the experimental results.Finally in Section 3 we state our conclusions.

2 Performance Metrics

2.1 Experimental Setup

The goal of this paper is to evaluate RTLinux performance. Our tests are conducted under RTLinux-3.2-pre3. We executed all of our tests on a common x86 PC with CPU Celeron 300A MHz and 128M SDRAM.

2.2 Performance Metrics and Experimental Result

We study the performance analysis of RTLinux operating systems by measuring the following key metrics.

Scheduling Jitter Scheduling jitter show the difference between when processes or threads wish to begin execution and when they actually are allowed to execute by RTLinux. This measures the scheduling deadline for a thread. This is often a limiting factor for real-time applications since it determines how much precision can be expected. It also provides guarantee on the largest delay that the applications by experience do to system performance. This test creates a thread that is available. The thread runs at 1 kHz (period of 1ms) and computes the difference between when it was scheduled to wakeup and the time it is actually woken up for many times while the thread runs. We have the test on three different environments: normal condition, the condition with I/O visit (using ls -lR simulates), the condition with the memory disturb (using cat /dev/mem), and we show the experimental data in figure1. Usually scheduling jitter will be affected notably for the I/O load and memory disturb, but we can conclude that RTLinux shows strong robust with strong real-time performance.

RT-FIFO RT-FIFO (First-In-First-Out) is one kind of RTLinux standard communication interfaces. It is used to interact between real-time tasks and user-level applications. Data in rt-FIFO is dynamically updated. RTLinux is analogous to UNIX pipes.

We perform a number of measurements on these systems to determine any latency that using them may cause as well as any latency they may experience due to other operations on the system. The kernel maintains exactly one pipe object for each rt-FIFO special file that is opened by at least one process. The rt-FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, one ends of the rt-FIFO couldn't open until the other end close.

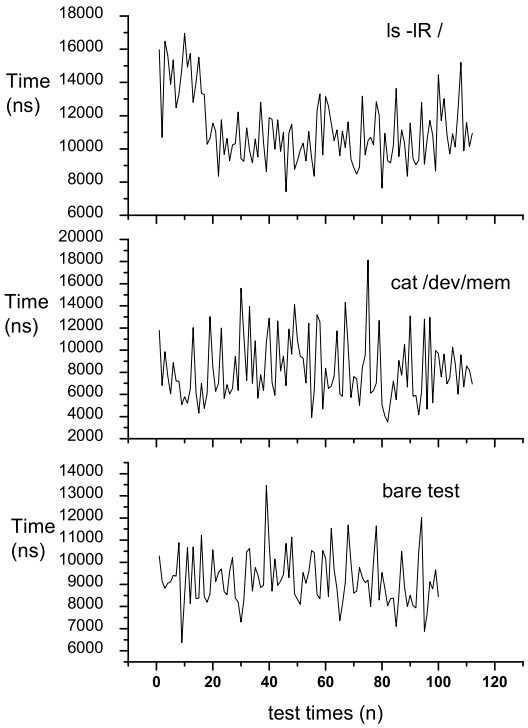

FIGURE 1: *Scheduling jitter test on three different conditions*

As the incertitude of the written data size, we only give a simple measurement with transmission velocity. After a constant size block of data was written and the measurement time has obtained, we calculate the data getting from rt-FIFO and get the transmit speed. We measure the two implementations with the different size of the transmit data. Table 2 shows the corresponding relationship between the rt-FIFO size and transmission speed when the size of the transmission data Sizetrsmit is 1MB.And the Table 1 is on the transmission data Sizetrsmit =10 MB.

Size of FIFO	Speed(MB/s)
10KB	604.148
100KB	479.607
1MB	224.082
10MB	*
100MB	*

TABLE 1: *Sizetrsmit = 10 MB * represents kernel panic.*

Size of FIFO	Speed(MB/s)
10B	*
100B	47
1KB	329
2KB	541
3KB	684
4KB	786
5KB	867
6KB	916
8KB	963
9KB	690

TABLE 2: *Sizetrsmit=1 MB '*' represents kernel panic.*

From these tables, we know the size of rt-FIFO needn't be so large, when size of rt-FIFO is from 1kB to 10 kB and it is the best size once written, the transmission speed is the fast. That is consistent with the BUFSIZ that the stdio recommends. In this experiment, when the size of rt-FIFO is 8KB, it get the best performance.Because when the size of rt-FIFO is for less content,it need more times for transmitting data. We recommended that size of rt-FIFO is in the range from 1 to 10 KB will be the better choice.Usually the system set the default value 4kB for rt-FIFO,so it's a good recommended value.

We study the performance with its' szie from 4KB to 1MB, and every time 1MB data is wrote in the rt-fifo and the buffer size is 8kB.(Figure 2)

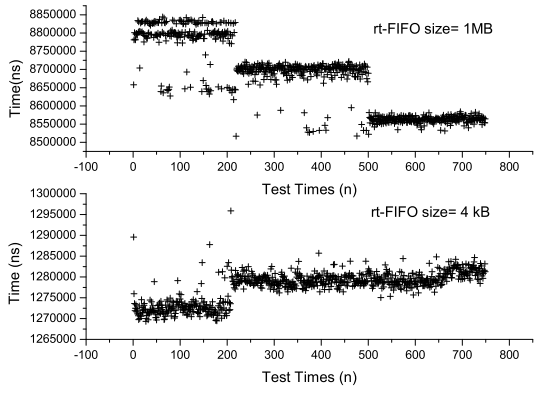

FIGURE 2: *Result of rt-FIFO under different size*

MBUFF Mbuff is also one kind of RTLinux provided standard communication interfaces. It is idle memory between kernel and user task for communication application. It is for the share data and is provided with share memory mechanism, it means data can be transmitted freely. We can think mbuff as full duplex data communication .Real-time task and user task can exchanges data by some special rules in mbuff, but it can not go beyond the mbuff size. This can cause kernel panic for illegally accessing memory. When we adjust mbuff size, we can observe the effect of real-time tasks. The efficiency experiment result is as followed.

When the size of mbuff is 4KB, every time we write the data with the size of 4kB to mbuff .While the other conditions were constant, we add the mbuff size to 1MB, the result shows in Figure 3.

Mbuff Size	Speed(Mb/s)
10B	8
100B	42
1KB	107
10KB	150
100KB	237
200KB	157
300KB	129
400KB	120
600KB	112
800KB	107
1MB	103
2MB	102
3MB	102
4MB	101
5MB	101
6MB	101
7MB	101
8MB	100
9MB	100
10MB	*

TABLE 3: *The transmission speed under the different mubff sizes. * represents kernel panic.*

We also measure the performance of scheduling jitter with the disturbed by the mbuff writing/reading. Scheduling jitter is the user task, not the real-time task, but the mbuff writing/reading is the real-time task. So the mbuff write/read process should affect scheduling jitter. This is because of the resource competition. We show the experimental result in figure 4. Before break point (Figure 4), we do jitter test with the mbuff in the W/R status. At the break point, the mbuff is in the sleep status.

Conditional variables A condition (short for Condition variable) is synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied.

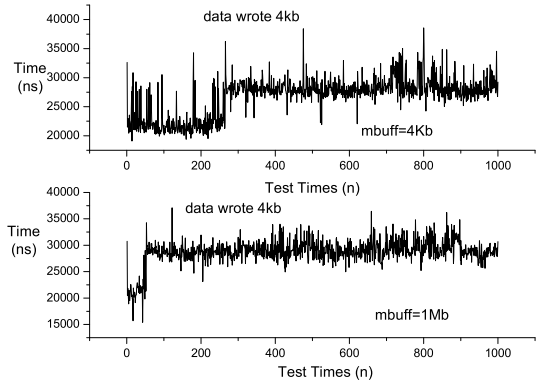

FIGURE 3: *The result of the normal Mubff test*

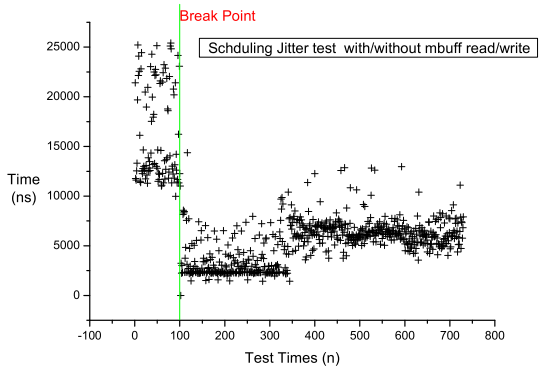

FIGURE 4: *Jitter test with/ without mbuff read/write*

The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition. A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it. On the conditional variables test, we measure the latency between a pair of operates for many times, `pthread_cond_wait()` and `pthread_cond_signal()`. We specify a cycle and let RTLinux kernel process some tasks. We expect it can finish this task in this cycle. In fact, the results must have some difference with the theory data. We set the theory data as a, the result in practice as b, so the bias defines as $p = b - a$. In the specified experiment times, we can get a largest bias series of pmax and a minim one of pmin. We do the test for N times, and then we can get N couples series of pmin and pmax. The data is sent to user space from the kernel space by rt-FIFO. We show the result in figure5 below.

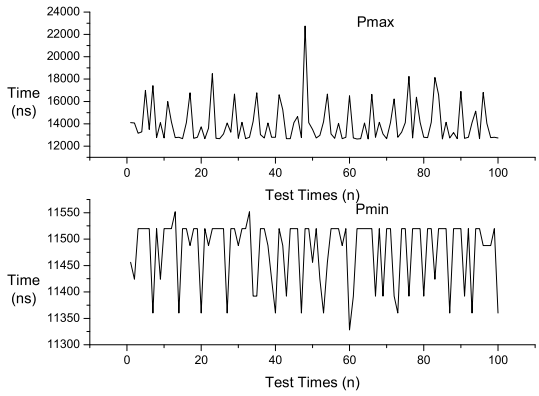

FIGURE 5: *Conditional variables test result*

The other test The other measurements have the resemble principle, we only show the result in the below table 4:

Name	min	max	avg
Context switch	7744	10304	8194
Mutex	2048	8256	2997
Spinlock *	-	-	-
Contested mutex	10976	15680	12110
Semaphore	9056	13952	10097
Priority Invension Recovery	1632	20096	1902

TABLE 4: *The result of the rest tests*

*: The spin lock test is invalid because our test computer is a uniprocessor machine - spin lock should only affect the SMP system. And the spinlock should be an empty operation in UP (uniprocessor) system.

3 Conclusions

In this project, we measured several real-time operating system key metrics to measure the RTLinux performance. We have evaluated the real-time behavior of RTLinux by measuring the latency of various kernel variants. It's important because most of them are very common used in the rt-task. Measuring them can also help us evaluate the real-time performance of the entire system. Our experiments and analysis

show that Real-Time linux provides good raw performance and it is suitable as an operating system platform for developing and running soft and hard real-time applications.

Acknowledgements

We would like to acknowledge to Jing Tao and Tang Jun from Lanzhou University for their help and fruitful discussions that have contributed to improve the quality of our work. We also especially thank Opentech EDV Research GmbH support this work.

References

[1] Phil Wilshire, 2000, Real Time Linux: Testing and Evaluation, Proceeding of Second Real-Time Linux Workshop, Orlando,FL, 2000.

[2] Opentech EDV Research GmbH, http://www.opentech.at/

[3] Frederick M.Protor, 2001, Measuring Performance in Real-Time Linux, the Third Real-Time Linux Workshop in Milan Italy.

[4] Cort Dougan, 2003 March,VME under RTLinuxPro,VME-Bus Systems Journal

[5] Cort Dougan, 2004 March,Lies, misdirectoin and realtime measurements,C/C++ Users Journal.