SystemTap Tapsets for the Real-Time Linux Kernel

Nitin Kesarwani
Department of Computer Engineering, NITK Surathkal
Mangalore, 575025, India
nitin966@gmail.com

Lalatendu Patra
Department of Computer Engineering,
NITK Surathkal
Mangalore, 575025, India
patra.mailbox@gmail.com

Annappa Basava
Department of Computer Engineering,
NITK Surathkal
Mangalore, 575025, India
annappa@nitk.ac.in

***Abstract*—Latency is an essential factor for measuring effectiveness of realtime applications. An effective realtime system aims at guaranteeing a practical deadline for a task, rather than improving throughput of the system. A subset of these applications includes the ones, which deploy realtime operating systems (RTOS). Linux RTOS has varied applications, few of them being at business trading centre, submarines, missile launching systems, satellite navigation system, etc. Considering the criticality of these systems, its top most priority that these RTOS should be almost near to perfection as they form the core. Hence, these systems need to be tested thoroughly, before they are applied anywhere. SystemTap is one such scripting tool which extracts information from a running kernel, which is unlike the traditional method of using printks. We aim at testing the performance of given RTOS by writing SystemTap scripts for various scenarios(provided by RTOS development teams) that arose as a result of problems faced in the past.**

Keywords-SystemTap, RTMutex, Futex, CF Scheduler, Static markers

I. INTRODUCTION

Realtime operating systems are characterized by the property that they ensure a bounded latency for a process, not providing high throughput necessarily. They generally use specialized scheduling algorithms whose aim is to ensure quick response to a change in process's priority rather than focusing on the amount of work the processes perform. One such example of the realtime operating system is the Linux realtime operating system which gains soft realtime characteristics after the *CONFIG_PREEMPT_RT* patch is applied to it. The philosophy of this patch is to minimize the amount of kernel code that is non-preemptible, while also minimizing the amount of code that must be changed in order to provide this additional preemptibility. In particular, critical sections, interrupt handlers, and interrupt-disable code sequences are normally preemptible. The *CONFIG_PREEMPT_RT* patch leverages the SMP capabilities of the Linux kernel to add this extra preemptibility without requiring a complete kernel rewrite. The features which *CONFIG_PREEMPT_RT* patch provides are preemptible critical sections, preemptible interrupt handlers, preemptible "interrupt disable" code sequences, priority inheritance for in-kernel spinlocks and semaphores, deferred operations and latency-reduction measures.

We brief about some of the important subsystems of the kernel for which we'll be writing scripts.

A. The O(1) Scheduler

Prior to the 2.6.23 kernel [5], O(1) scheduler was implemented, which was O(1) in time.

1) How it works?? : Each CPU has its own runqueue, and it is a priority list, 140 priorities (100 real time tasks, 40 user tasks). Each priority has a list, in which processes of the same priority gets added in FIFO manner. To find which job is to be scheduled next, CPU finds out "which bit is set next" in the priority array. Each CPU has two arrays, *active* and *expired.* The active array of lists has all processes that have been selected from swap space to run. When a running process exceeds its allotted time slice, it is pushed into expired runqueue, and, its priority and future time slice is recalculated.

2) Other features: Load-balancing and dynamic task prioritization, to prevent a task from hogging the CPU.

B. The Completely Fair Scheduler (CFS)

CFS uses *time-ordered Red-Black trees* for implementation [6]. As these trees are balanced, we get an *O(logn)* guaranteed time, which is impressive even in worst cases of implementation.

Major features and policies:

- Modular scheduler introduced the scheduling classes in the kernel
- Identifying process with "gravest need" for CPU
- Group scheduling, helps in imparting fairness to group of jobs, rather than jobs themselves.

It uses an appeasement policy that guarantees fairness. Whenever a task enters a runqueue, its *wait_runtime* starts incrementing to count the payoff that needs to be done when it gets scheduled, depending upon the number of processes

978-1-4244-8666-3/10/$26.00 © 2010 IEEE

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on July 06,2021 at 08:58:38 UTC from IEEE Xplore. Restrictions apply.

in the runqueue as well as the priorities of the tasks. As soon as it is scheduled, its wait_runtime value starts decrementing till other process becomes its left-most child. At this time, it's preempted. Thus, CFS tries to achieve the ideal situation by making wait_runtime to become zero.

C. Static Probe Points

ftrace starts given command and according to tracing script given via command-line arguments; it traces its system calls, symbol entry points in general, and possibly other events as well. It uses the *Frysk* framework to implement tracing. With kernel markers, the placing of probe points is easy:

```
#include <linux/marker.h>
trace_mark(name, format_string, ...);
```

The name is a unique identifier, which is used to access the probe.

Code which wants to hook into a trace point must call:

```
int marker_probe_register(const char *name, const char *format, marker_probe_func *probe, void *pdata);
```

D. Futexex

A *futex* (short for "fast userspace mutex") is a Linux construct that can be used to implement basic locking, or as a building block for higher-level locking abstractions such as *semaphores* and *POSIX* mutexes.

Futex operations are carried out almost entirely in userspace; the kernel is only involved when a contended case requires arbitration. This allows locking primitives that use futexes to be very efficient: since most operations do not require arbitration between processes, most operations can be performed without needing to perform (relatively expensive) system calls.

E. RTMutex

RT-mutexes extend the semantics of simple mutexes by the *priority inheritance* protocol. A low priority owner of a rtmutex inherits the priority of a higher priority waiter until the rtmutex is released. If the temporarily boosted owner blocks on a rtmutex itself, it propagates the priority boosting to the owner of the other rtmutex it gets blocked on. The priority boosting is immediately removed once the rtmutex has been unlocked.

This approach allows shortening the block of high-priority tasks on mutexes which protects shared resources. RT-mutexes are optimized for fastpath operations and have no internal locking overhead when locking an uncontended mutex or unlocking a mutex without waiters.

II. Literature Review

To find out whether problems involving any of the above subsystems have occurred, we need introspection of a running kernel. Thus, there is a need of a tool which should provide proper diagnostic information to the user/tester, so that finding the solution to a problem becomes easier. SystemTap is one such tool. It provides an infrastructure to gather a lot of diagnostic information about a running Linux system as explained in [2].

Until SystemTap, in order to gather information from the running Linux system, there used to be a need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence as described in [1]. The process is time consuming and requires in-depth knowledge of multiple subsystems. SystemTap uses dynamic instrumentation to make this same level of data available without the need to modify kernel source or rebuild it (except for the case where static probe points are used). It delivers this data via a powerful scripting facility which works in following stages: parsing script and reporting syntactical errors, elaborating output using standard tapset libraries, translation into a C file, compilation into a kernel module, loading generated module into running kernel and displaying output/error/skipped probes etc. Upon exit, the loaded module is removed from the running kernel.

This flow asserts that script gets converted to a kernel module, and loaded into the kernel. Thus we need not reboot the system, also avoiding tracing the log files (unlike what [3] does using printk).

Existing tools like **iostat, vmstat, top** and **oprofile** [1],[3] are valuable for understanding certain types of performance problems, but there are many kinds of problems, as discussed by [4], that they don't readily expose, some of them being:

1) Interactions between applications and operating system
2) Interactions between processes
3) Interactions between kernel subsystems
4) Problems that are obscured by ordinary behavior and require examination of an activity trace

Often these problems are difficult to reproduce in a test environment, making it desirable to have a tool that is sufficiently flexible, robust and efficient to be used in production environments. These scenarios further motivate our work on SystemTap.

III. Problem Statement And Its Scope

The work involves verifying that SystemTap functions properly on real-time kernel, fixing problems in SystemTap or kernel, if any, that prevents SystemTap from functioning, understanding the working of real-time kernel, understanding typical functionality, latency and performance problems faced on real-time kernel, designing tapsets for various such scenarios, testing the written patches to confirm the information collected is useful and finally publishing these tapsets for use by Linux community.

We worked on following scenarios:

1) Print details (arguments) of all the futex system calls a task makes. In case any of the futex syscalls returns an error, catch the error and print a warning message.
2) Capture information on tasks being migrated from one CPU to another (with the task pid or comm being passed as parameter). Migration is one of the sources of latencies.

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on July 06,2021 at 08:58:38 UTC from IEEE Xplore. Restrictions apply.

3) Gather information on all preemptions suffered by a particular task.
4) Information on how much time a task spends waiting on the runqueue (in runnable state, waiting for CPU).
5) Collect all relevant information on all the priority boosting and de-boosting that a particular task went through (due to PI mutexes).
6) Locking statistics (time spent spinning, sleeping, etc). Since mutex code is in Systemtap blacklist, so a static trace-point based script.
7) Capture information on signals that are missed by any task (missed wakeups, for example). There had been quite a few issues in the past which were due to tasks missing wakeups.
8) rtTop command that gives information on the running RT tasks (cpu, preemptions, total latencies, etc.).
9) Something like a latencyTop that gives information on the various latency sources.

Importance of problems holds from the very fact that the problem scenarios have been an issue of concern for the realtime development team. Example, in case of capturing information for missed signals: missed signals are those which are missed by the recipient process, such as missed wake up call. So, if a high priority process is sleeping and is waiting on a signal, and if that signal is missed, then unwanted latency comes into frame as the process continues to sleep.

So, with the help of our script, we shall help in tracing all these issues and aim to provide useful and relevant information to the user so that the area of concern is specified.

IV. Implementation Details

The methodology used to solve the problem involved two steps:

A) Setting up of the RT Kernel and SystemTap:

1) Installation of Real-time kernel: Versions:
- linux-2.6.26.5
- Real time patch patch-2.6.26.5-rt9

2) Getting SystemTap to run:

i) Install SystemTap package

ii) Kernel Support: Install linux-image-debug-generic

iii) Also, create a symlink for vmlinux in /lib/modules

```
# sudo ln -s /boot/vmlinux-debug-$(uname -r)
  lib/modules/$(uname- r)/vmlinux
```

iv) Devel environment: Kernel headers and gcc are needed for module compilation. libcap-dev is needed for SystemTap packages >=0.61

3) Compiling the kernel for static markers: enable the *ACTIVATE_MARKERS* option in the config file for the kernel.

B) Scripting Work: Work corresponds to problem scenarios described in chapter 3.

1) Script 1: Probe point is syscalls.futex. We begin by initializing various futex operations into an array (e.g. operation =0 means *FUTEX_WAIT*, taken from futex.h in the kernel source). For each system call, a flag is indexed by the address of a futex. So, if the futex at a particular address has been grabbed, the flag shows a truth value, causing contention for others who want to grab the same futex lock. In return path, we check whether there's an error in return path or not and print the error message along with the reason of error.

2) Script 2: Probe point is the function "__migrate_task" in the sched.c file. This requirement is to print migration status. 4 different cases may happen, which are:

i) Target cpu may be offline, resulting in no migration.

ii) Process's affinity of cpu doesn't have flag set for target cpu.

iii) Task has already migrated due to pull_task called by the target cpu.

iv) Successful migration, as all of above didn't happen.

We use 4 different markers for these 4 conditions which, correspondingly, are:

1) __migrate_task_cpu_offline
2) __migrate_task_affinity_changed
3) __migrate_task_already_moved
4) migrate_task_done

We also print the kernel backtrace as extra information. The script has been tested by manually migrating a task by changing its affinity using the *taskset* command.

3) Script 3: Probe point is *context_switch* function, before the call to *switch_to* function. Whenever the *context_switch* function gets called, the control passes over the static marker placed within the function and the handler of our script gets executed. The script initializes various states of transition (*TASK_RUNNING*, *TASK_INTERRUPTIBLE*, etc) which has been taken from include/sched.h. Since context switches are very frequent, we display the statistics of context switch that occurred in the past 1 millisecond using a timer probe.

4) Script 4:.There are 4 functions of interest for this script in the scheduler, which are:

i) activate_task – Puts a task into the runqueue.

ii) deactivate_task – Moves a task out of runqueue.

iii) context_switch – Gives information on which pair of tasks are being scheduled in/out.

iv) finish_task_switch – It performs cleanup functions after context switch. If a task is in runnable state, after being context switched, it places it in runqueue again.

We probe above 4 functions and make use of *gettimeofday_us()* function to return entry and exit time of runqueue, using which we can find the difference, which is our required result.

5) Script 5: Probe point is *__rt_mutex_adjust_prio()* which gets called while any adjustment to priority of a process is to be done. It calls *rtmutex_getprio()* function to get new priority of a task. This function alters the priority, hence, if the return value is greater than the previous normal priority of the task, then a boosting of the priority has taken place, else if it is lesser then deboosting.

6) Script 6: We used 3 static markers here:

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on July 06,2021 at 08:58:38 UTC from IEEE Xplore. Restrictions apply.

i) rtmutex_slowlock probes *rt_mutex_slowlock* call to *schedule_rt_mutex*.
ii) rtspinlock_slowlock probes *rt_spin_lock_slowlock* call to *schedule_rt_mutex*
iii) __futex_wait probes schedule done due to wait on futex.

In above 3 markers, we note the time of call and set flags. When the *try_to_wake_up()* function is called, we find the reason of call from the flag for a particular process and find the delta time spent. When *do_exit()* function is called for a thread, we print its sleep statistics.

7) Script 7: Probe point is *signal.send* which runs whenever a signal is sent to a process. It probes function *specific_send_sig_info()* in the signal.c file. From here we are able to extract information related to the sender process, receiver process and the signal sent. We put the data obtained into the respective associative arrays. Other associative arrays stores information about the process and the signal. Other probe point is a static marker *getsignal_to_deliver* which is placed just before the end of the function *get_signal_to_deliver()* in the signal.c file. This function is called by the *do_signal()* function in the signal.c file. The function get_signal_to_deliver() actually extracts a signal from the signal queue which is then handled by the *handle_signal()* function. The static marker gives us two parameters. The first one is the signal number and the second is the return value of get_signal_to_deliver(). The return value is greater than zero if signal is successfully extracted from the queue. Hence from this probe point we are able to determine the signal which is received. At the end of the script we subtract the total received signals of a particular type from the total signals sent of the particular type and determine the missed signals. We then display the data accordingly.

8) Script 8: In this script, we use same three 3 static markers as mentioned in script 6. In *try_to_wake_up* function, the realtime process wakes up and the difference between its sleep and wakeup is the time delay or the latency. We gather the number of context switches which the realtime function faced, from the */proc/<pid>/status* file, by taking the difference of two values, one at start of the script and other at the end. Also, we print out the percentage CPU utilization by each realtime process.

9) Script 9: Probe points are 116 functions which are probable cause of latency, gathered after reading kernel code. In *begin* probe, we initialize the reason for each type of latency. Now, in each function which is being probed for latency, we gather the time entry and find out the difference upon exit. If the function is real cause of latency, then this difference is bound to be large (greater that customizable threshold). Each of these functions should have a predictable behavior in realtime scenario, which is decided by threshold time which it should take in the worst case for its completion. If it takes more time, this means that it is causing unnecessary latency in the system. Repeated reporting of such a function may lead to further development as it increases a chance of finding subtle bug in that function's implementation logic, because of which it's been showing an unpredictable behavior. The threshold has to be set by the developer/expert of the system because deadline requirements are not same for all the systems.

V. RESULTS AND ANALYSIS

We discuss in this section, the problems faced, followed by Results obtained, in form of screenshots for all scripts at the end. (For more clear view refer [7])

A) Problems Faced

1) STAP_MARK support had been withdrawn. Problem was solved by the usage of trace_mark.

2) Static markers problem:

Use of static markers, following error used to be thrown during kernel compilation:

VDSO arch/x86/vdso/vdso32-sysenter.so.dbg
CC kernel/sched.o
In file included from kernel/sched.c:31:
include/linux/marker.h:33: error: expected declaration specifiers or ... before 'va_list' [...]

A patch provided by *Lai Jiangshan*t fixes the bug of missing modpost entry in Module.markers.

3) After running any script, upon dmesg, a big bug was shown generally. We have reported this bug to the Systemtap and realtime kernel developers and this is one of our genuine findings in this project.

4) The handle_signal() is an inline function and architecture dependent. Hence, we could not gather complete statistics from it.

VI. CONCLUSION

SystemTap is a tool for Linux Operating System that allows developers and system administrators to deeply investigate behavior of the kernel and even userspace applications in order to discover error conditions, performance issues or just to understand how the system works. In course of the project we found that it has immense potential and careful and intelligent use can help to monitor minute intricacies of kernel which were not possible before this. It is because of this reason many software giants like Red Hat, IBM, and Oracle are promoting and contributing in its development.

The scripts developed can be used individually and with slight modification can be used as a part of larger scripts. We hope the scripts developed in this project will help the intended users to closely monitor the certain aspects of the kernel and serve its purpose.

VII. FUTURE WORK

Linux Kernel is an ever growing code. Changes get incorporated into the kernel every now and then. The scripts developed mainly probes the kernel code. New major changes to kernel code may lead to newer issues. Those issues need to be fixed by introducing corresponding changes in the scripts. In other words the scripts need to evolve along with the kernel.

The scripts are developed mostly for generic cases. Further customization can be done to the scripts match specific cases depending upon case scenario.

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on July 06,2021 at 08:58:38 UTC from IEEE Xplore. Restrictions apply.

The scripts developed can be evoked from shell scripts and the output can be filtered further. It provides opportunity for larger gui based application to use these scripts and extract relevant information.

REFERENCES

[1] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, Brad Chen, "Locating System Problems Using Dynamic Instrumentation", 2005 Linux Symposium

[2] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[3] Prasanna S. Panchamukhi. (Aug 2004) "Kernel debugging with kprobes: Insert printk's into linux kernel" http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnx%w07Kprobe. (Dec.27, 2008)

[4] Unknown,(Jan 2002) Linux project publications:RAS,http://www.ibm.com/developerworks/opensource/library/os-ltc-ras/

[5] Robert Love (2007) "Linux Kernel Development" , Pearson Education, Delhi India, "ProcessManagement" pg. 23-38

[6] Robert Love (2007) "Linux Kernel Development" , Pearson Education, Delhi India, "Process Scheduling" pg. 39-62

[7] http://picasaweb.google.com/nitin966/Screenshots#

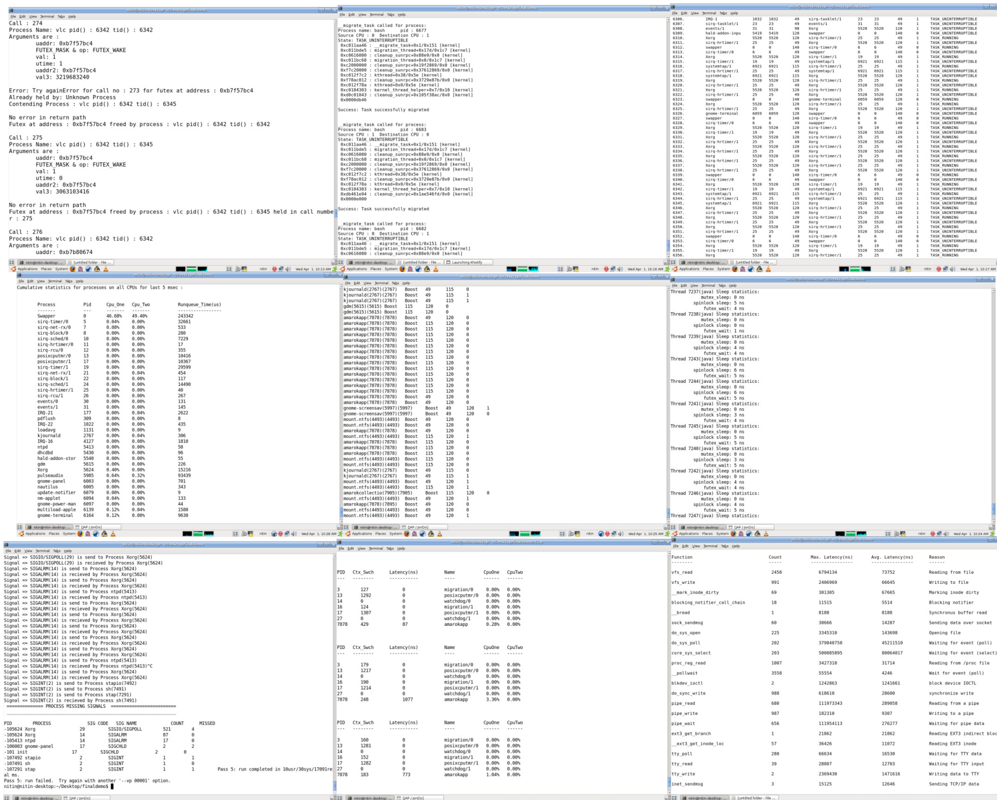

Fig. 1: Result of scripts Displays results of all scripts in order

In *Proceedings of the Third Symposium on Operating*

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on July 06,2021 at 08:58:38 UTC from IEEE Xplore. Restrictions apply.