Assessment of the Realtime Preemption Patches (RT-Preempt) and their impact on the general purpose performance of the system

Siro Arthur
Research Fellow, Distributed And Embedded Systems Lab (DSLab)
Lanzhou University, China
Distributed & Embedded Systems Lab, School Of Information Science and Engineering
Tianshui South Rd 222Ŀanzhou 730000 ṖṘĊhina
siro@lzu.edu.cn

Carsten Emde
Open Source Automation Development Lab, OSADL
Carsten.Emde@osadl.org

Nicholas Mc Guire
Distributed And Embedded Systems Lab (DSLab)
Lanzhou University, China
mcguire@lzu.edu.cn

Abstract

With the maturing of the Realtime Preemption Patches (RT-Preempt) and their stepwise integration into the Mainline Linux kernel since version 2.6.18, we set out to answer the questions:

* How good is RT-Preempt with respect to the worst-case latency?

* How expensive is RT-Preempt with respect to a possible performance degradation of the system?

Taking that a lot of the preemption techniques deployed have their origin in scalability demands and not so much in realtime requirements, the most interesting case to look into is related to uni-processors - on these we would expect the worst-case impact of RT-Preempt. To answer the question, we ran an extensive benchmark series on 2.8-GHz P4 and 1-GHz/600MHz VIA CIII boards, measuring general OS performance parameters as well as the realtime capabilities. For the latter, a trivial parport toggle program was used.

The results show that high-end CPUs are well supported by RT-preempt in general. Low-end systems typically of interest for automation and control, however, still need some work.

In this paper we will outline the method used for evaluation and present the details of the results.

This work was partly supported by the Open Source Automation Development Lab (OSADL).

Keywords: RT-Preempt, Jitter, Performance

1 Introduction

Linux is a General Pupose Operating System (GPOS) and as such is designed to provide good overall throughput rather than guarantee deterministic response for applications requiring real time. Though stock Linux has support for POSIX real time scheduling policies (SCHED_FIFO, SCHED_RR), these 'services' only increase the priority of an application but did not necessarily make the rest of the system more preemptive. Thus the configurability of task policies was included but the ability of the system to honor these in all situation was still limited - though it should be noted that creating the necessary infrastructure to allow managing real-time capabilities of tasks/threads was the first (and quite large) step towards real-time in mainstream Linux.

Over the last decade or so, there have been several attempts to add real time capabilities to Linux. At present, several ways of approaching real time Linux have been proposed and/or implemented [1] [2][3]. However, as far as hard real time is concerned, interrupt abstraction techniques - RTLinux[4], RTAI[5], L4/Fiasco[6], ADEOS[7], Xenomai[8], XtratuM[9] - have been (and still are) the most widely used - at least in the free/open source community.

The key issue with regard to preemptibility is management of interrupt disable sections - the responsiveness of a system is limited by the amount of time (proportional to the code length) which is executed with disabled interrupts in addition to hardware related artefacts that can't really be attributed to the OS in question. Thus Interrupt abstraction makes the entire GPOS kernel preemptive by adding a Hardware Abstraction Layer (HAL) to intercept and manage the interrupts depriving Linux of ever really disabling hardware interrupts - though the RTOS layer still has interrupt-disable sections of course, these are substantially shorter due to simplicity and sheer size of the respective RTOS HAL, and the RTOS HAL implementation have been designed from the very beginning with this problem in mind. Basically, a relatively small patch is applied to the kernel that diverts interrupt/timer control from Linux to an RT nanokernel/microkernel as soon as it gets loaded, i.e. the nanokernel itself is implemented as (a) Linux kernel module(s). It includes a real-time scheduler that runs Linux as the lowest priority thread (with respect to its real-time threads/domains), thereafter. This technique is capable of guaranteeing true hard real-time for GNU/Linux. Since it involves minimal modifications to the Linux kernel, it is probably the leanest and cleanest approach to adding real-time to Linux from a theoretical stand-point.

The main drawback with this method is that the available API and resources are limited - unrestricted access to the full plethora of GNU/Linux libraries is imposible without losing determinism. Another little snag is that code has to be run for kernel space, which is slightly more difficult to implement and/or debug and errors have more severe consequences, though both L4/Fiasco and XtratuM have addressed this issue by providing seperated addresss spaces for real-time domains.

However, there exists the LXRT userspace realtime module of RTAI that allows user space programming with a slight penalty of performance to access user-space resources when not in RT-mode and switch to RT-mode by a dedicated system call - then of course again with the aformentioned constraints.

Clearly the disadvantage of all of these approaches is the maintenance issue, the need for programmers to learn to opperate in multiple paradigms and interact between these efficiently and without side-effects. Further these approaches don't really benefit the existing applications and unfortunately many have missed appropriately addressing standards conformance - further complicating the ability to utilize them efficiently.

Lastly, there has been conciderable license related issues, which have been clarified by a number of technological developments that advanced the RT-capabilities of the HAL concept beyond the infamous FSMLabs patent.

Now, despite the advantages and wide-scale usage of the interrupt abstraction technique, the only realtime Linux approach that has garnered mainstream (Linux) acceptance is the preemption improvement patch. Started back in the 2.2.X kernel days by ingo molnar and Kurt Dougan continuously advancing with TimeSys' and MontaVista's efforts to improve preemptiveness in the context of mainstream linux and finally leading to a number of key design descisions regarding locking and scheduling during the transition to the 2.6.X series of kernels that set the environment necessary to target real-time in mainstream Linux.

This is a patch set that is maintained by a small group of (core) mainstream developers, led by Ingo Molnar. It allows nearly all of the kernel to be preempted, with the exception of a few very small regions of code ("raw_spinlock critical regions") - and the long standing uglyness of some historic code sequences (i.e. ttys). This is done by replacing most kernel spinlocks with mutexes that support priority inheritance and are preemptive, as well as moving all interrupt and software interrupts to kernel threads. (dubbed interrupt threading), which by giving them there own context allows them to sleep among other things.

It further incorporates high resolution timers - a patch set, which is independently maintained by Thomas Gliexner [10]. The high resolution timer patch allows precise timed scheduling and removes the dependency of timers on the periodic scheduler tick (jiffies) [11]. According to documentation found at rt.wiki.org, the high resolution time patch enables POSIX timers and nanosleep() to be as accurate as the hardware allows (around 1usec on typical hardware).

And as noted - along these big building blocks for kernel preemptivness, a lot of smaller changes related to scalability of Linux on multiprocessor systems have increased the real-time performance dramatically. Notably the introduction of RCU, lock-free synchronisation concepts (i.e. sequence-locks) and the years of removing the Big Kernel Lock (BKL) by fine-grain resource local locking, have had a profound impact on the ability to achieve improved preemptiveness. It should be noted that these efforts are by no means at there end and that building awareness of the scalability issues and the real-time constraints has simply taken a very long time - the transition from "thinking UP" to "thinking SMP" has made it to the kernel developer community - the next step of "thinking RT" is still in progress.

However, from a hard real-time perspective, it has been argued that with this approach, it is impossible to guarantee the worst case latency for it is not feasible to test all control paths the kernel may take - and therefore impossible to guarantee hard real time. Figures provided are therefore statistical in nature rather than conclusive. This has been subject to a lot of discussion and debate; several articles and threads in various mailing lists.

But with the maturing of the realtime preemption patch set and its stepwise integration into the Mainline Linux kernel since version 2.6.18, we set out to find out its 'viability' with respect to worst case latency and how expensive it was with respect to a possible performance degradation of the GNU/Linux system.

Now, taking that a lot of the preemption techniques deployed have their origin in scalability demands and not so much in realtime requirements, the most interesting case to look into is related to uniprocessors - on these we would expect the worst-case impact of RT-Preempt. We ran an extensive benchmark series on 2.8-GHz P4 and 1-GHz and 600MHz VIA CIII boards, measuring general OS performance parameters as well as the realtime capabilities. For the latter, a trivial parport toggle program was used.

This - we think - could be of particular interest to the Automation and Control Community as it directly maps to interacting with peripheral devices which is generally the critical issue for Automation.

2 Methodology

We selected three kernels: 2.6.21.5, 2.6.22.1 and 2.6.23-rc1. We chose to test multiple kernels reduce the chances of results being too specific to a kernel version. For the software benchmark, the LMbench test suite was used. LMbench is one of the most commonly used system level benchmarks that has generally shown reliable results. Essentially, we used lmbench to compare the un-patched kernels against patched kernels. For each patched kernel version, separate benchmarks were run against kernels of different RT related configuration settings. Now, while individual results might not be absolutely reliable with respect to absolute numbers (as these are quite system specific e.g. motherboard type etc), the comparison holds valid and is a clear indication of the impact that the RT related modifications at kernel level can have on system performance.

However, in order to not rely on software benchmarking alone, we also ran a simple hardware test to observe jitter. This consisted of a simple program that toggled the output of the parallel port producing a sqaure wave. Basically, this simple realtime thread requested to be scheduled periodically at 50us to produce a pulse with an ON interval of 50us at every 1ms. We chose 'at every 1ms' for we wanted to produce clear pictures with 'discernable' jitter for the different kernel configurations. While not a strict scientific method, it does ensure that the entire hardware chain is in the loop and thus the results are at least qualitatively reliable. Regarding any quantitative interpretation, again hardware specifics are the limitation though comparisons of course hold valid.

Once again, for each kernel version, jitter tests were run for the RT_PREEMPT configuration settings. These tests consisted in running the parallel port toggling thread under 'moderate' system load: we run 'make -j4' on a kernel, untar'ed another gziped kernel archive on another console while executing 'find /' on a fourth console. These jitter diagrams are then contrasted with those acquired by running the same program under RTLinux/GPL, RTAI and LXRT. However, we decided to be 'unkind' to the interrupt abstraction techniques and run them under relatively heavy system load (dubbed, stress test)[12]. Jitter measurements were acquired by connecting an oscilloscope to an output pin of the parellel port (figure 1 below). A Tektronix TDS 2014B Digital Storage Oscilloscope with a USB Flash Drive storage facility was used for the acquisition of the jitter patterns. The Oscilloscope's display was set to mode 'Persist' at 'Infite' value. The acquired images were then downloaded to a USB Flash drive and transferred to the a computer for further editing.

The PREEMPT(_RT) jitter patterns were recorded for several minutes only - this is of course a systematic problem of our approach - but nevertheless the results are usable as an indication of the overall performance of rt-preempt. However, the jitter patterns for RTLinux, RTAI and LXRT were acquired over several hours.

As a note on the kernel configuration settings: CONFIG_HIG_RES_TIMER was always set to true (=y) for all settings.
For all CONFIG_NO_HZ, CONFIG_HZ was set to 1000 as (currently) CONFIG_NO_HZ actually im-

plements the one shot timer.

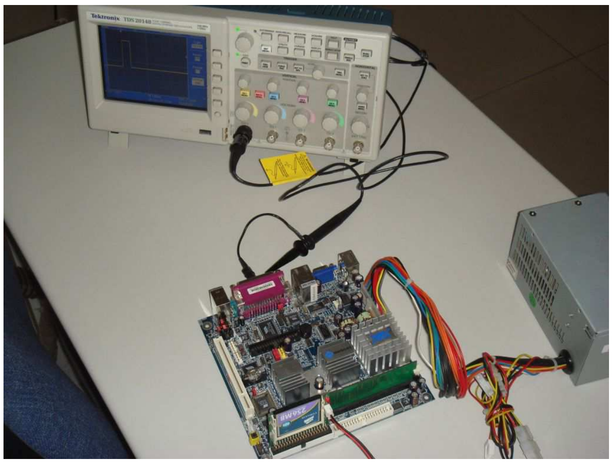


FIGURE 1: *A Tektronix TDS 2014B Digital storage scope attached to a VIA C3 600MHz fanless board*

3 Results

3.1 LMbench Results

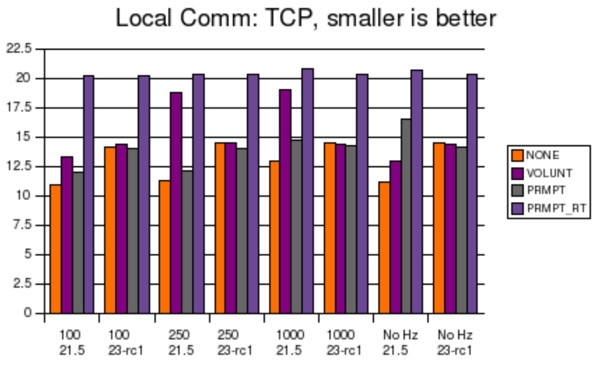


FIGURE 2: *Local Communication: TCP - latencies in us*

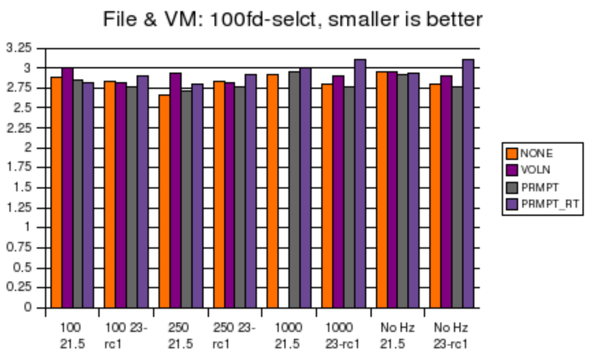


FIGURE 3: *File & VM system: 100fd_selct - latencies in us*

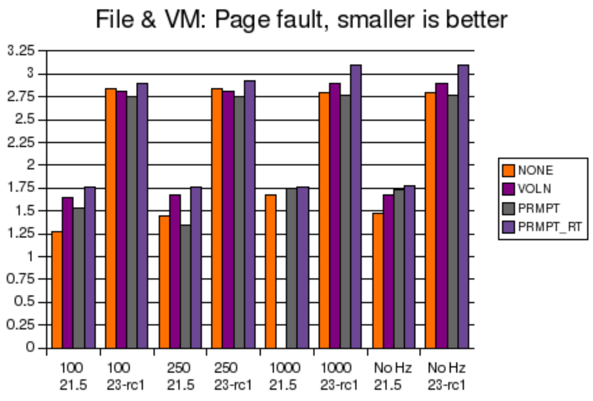


FIGURE 4: *File & VM system: Page fault - latencies in us*

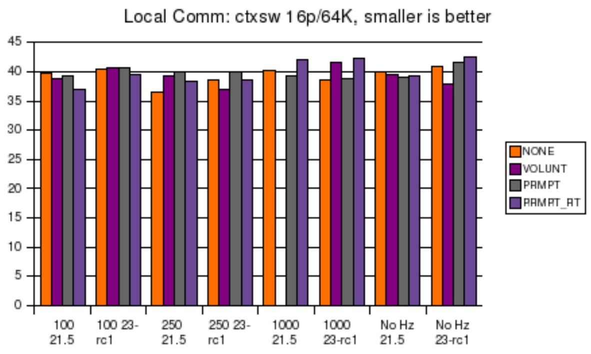


FIGURE 5: *Context switching times - latencies in us*

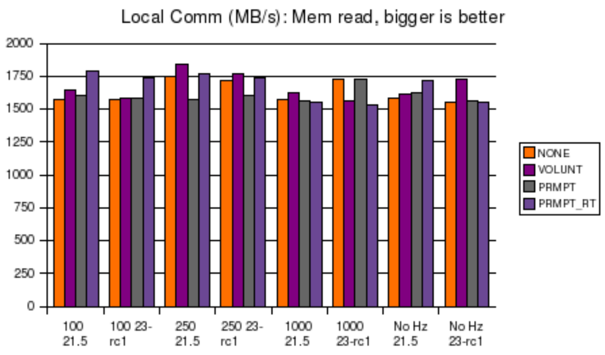


FIGURE 6: *Local Communication: Mem read - latencies in MB/s*

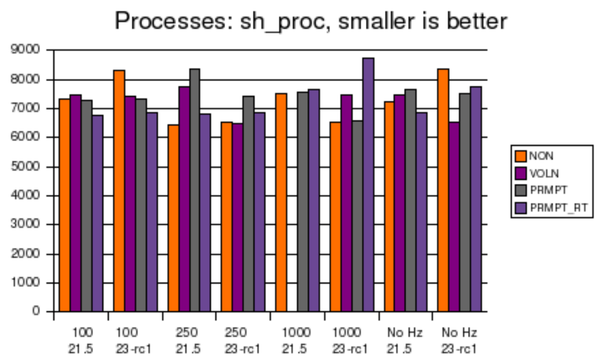


FIGURE 7: *Processes: sh_proc - latencies in us*

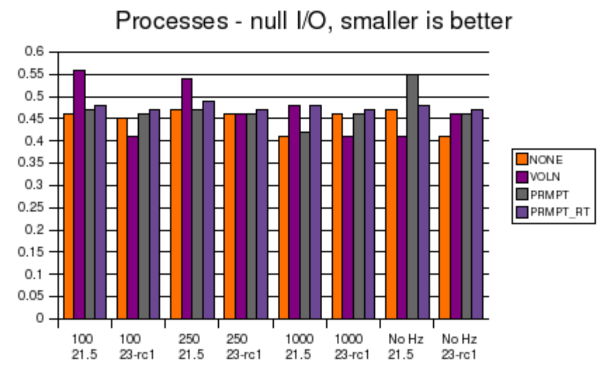


FIGURE 8: *Processes: null I/O - latencies in us*

3.2 Jitter Patterns

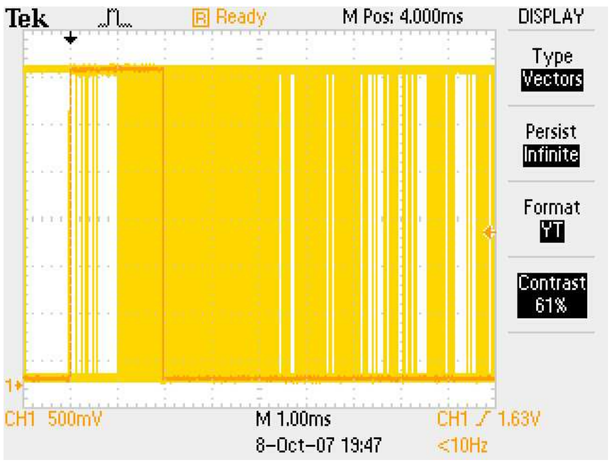


FIGURE 9: *Jitter on 2.6.23-rc1 CONFIG_PREEMPT_NONE, VIA C3 600MHz*

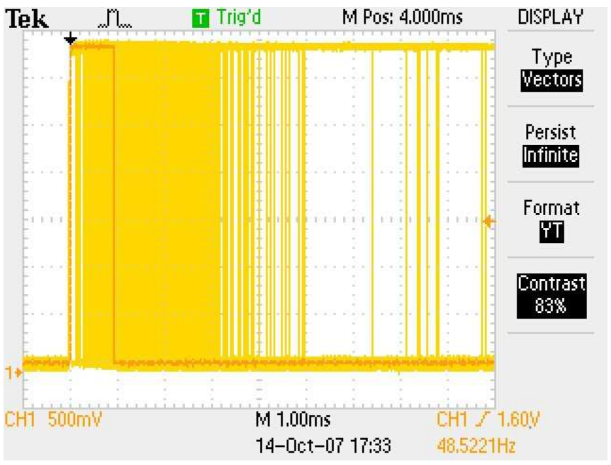


FIGURE 10: *Jitter on 2.6.23-rc1 CONFIG_PREEMPT_NONE, P4*

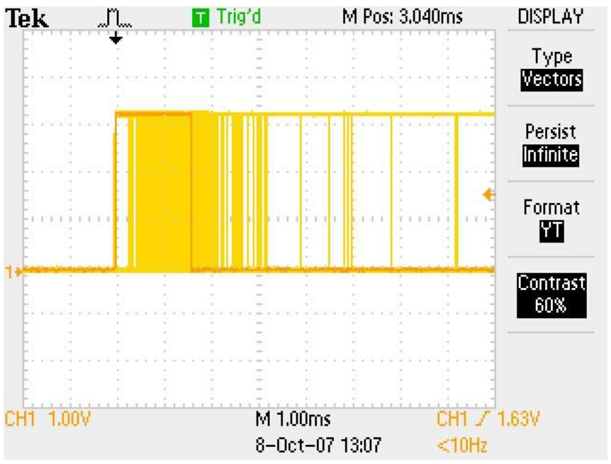


FIGURE 11: *Jitter on 2.6.23-rc1 CONFIG_PREEMPT, VIA C3 600MHz*

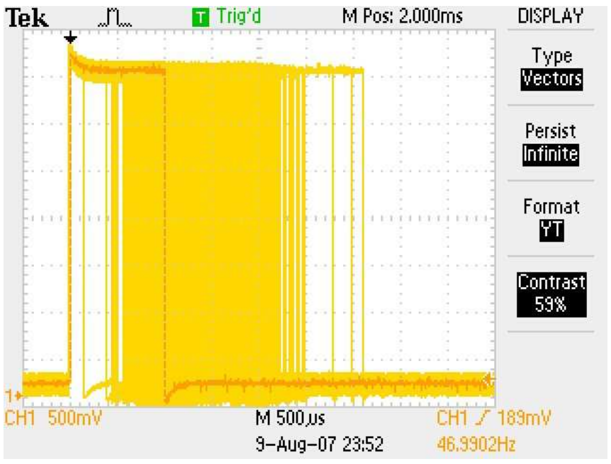


FIGURE 12: *Jitter on 2.6.23-rc1 CONFIG_PREEMPT, P4*

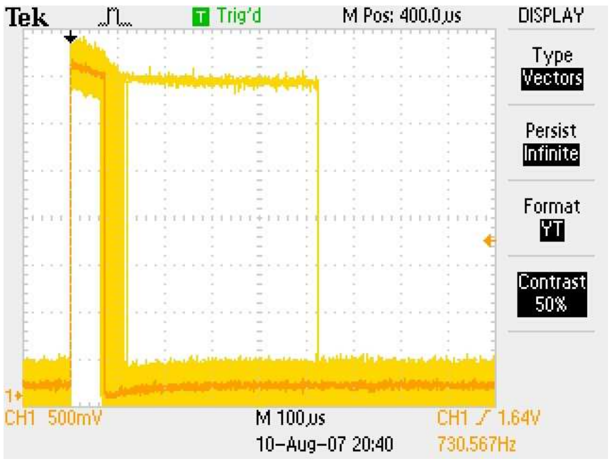


FIGURE 13: *Jitter on 2.6.23-rc1-rt1 CONFIG_PREEMPT_RT, VIA C3 600MHz*

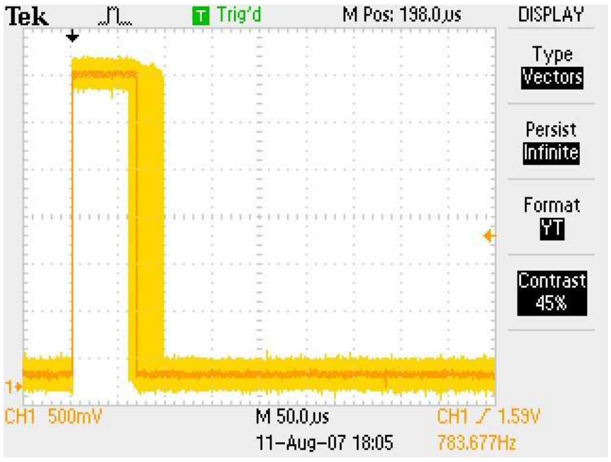


FIGURE 14: *Jitter on 2.6.23-rc1-rt1 CONFIG_PREEMPT_RT, VIA C3 1GHz*

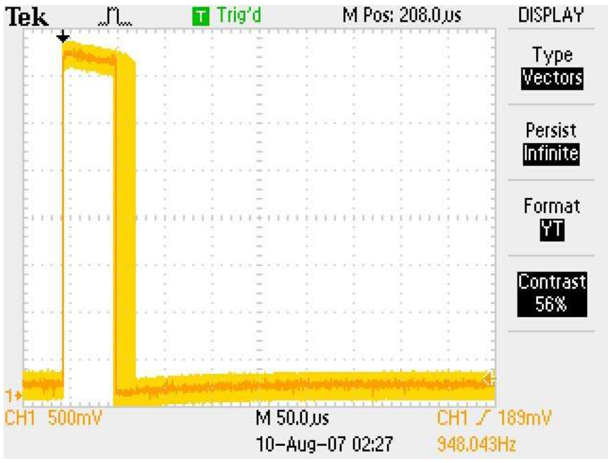


FIGURE 15: *Jitter on 2.6.23-rc1-rt1 CONFIG_PREEMPT_RT, P4*

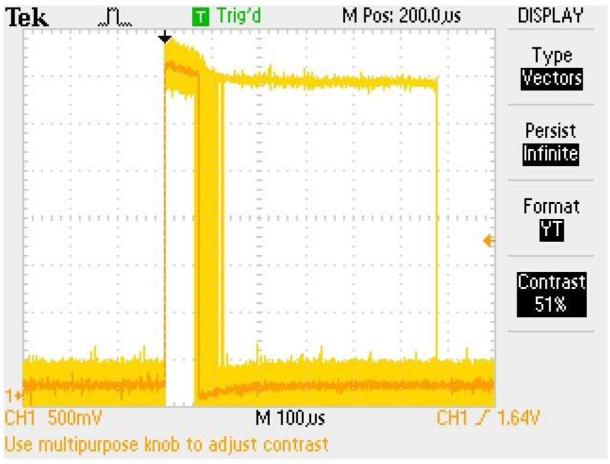


FIGURE 16: *Jitter on 2.6.22.1-rt6 CONFIG_PREEMPT_RT, VIA C3 600MHz*

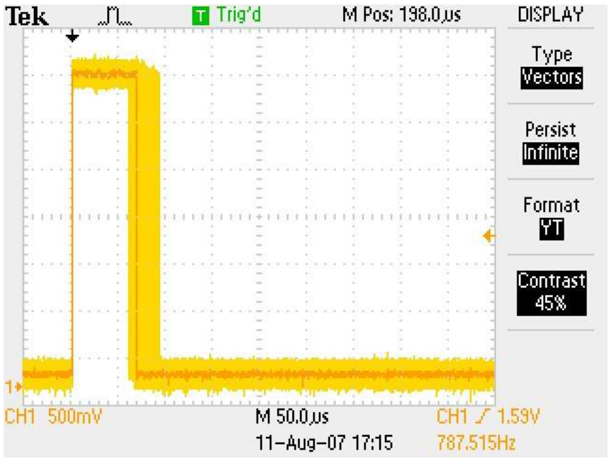


FIGURE 17: *Jitter on 2.6.22.1-rt6 CONFIG_PREEMPT_RT, VIA C3 1GHz*

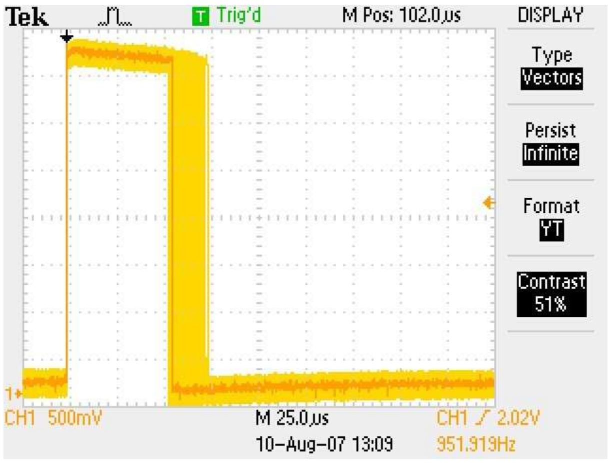


FIGURE 18: *Jitter on 2.6.22.1-rt6 CONFIG_PREEMPT_RT, P4*

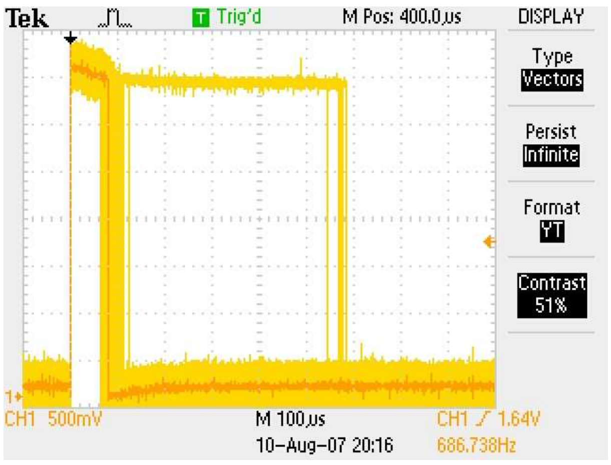


FIGURE 19: *Jitter on 2.6.21.5-rt20 CONFIG_PREEMPT_RT, VIA C3 600MHz*

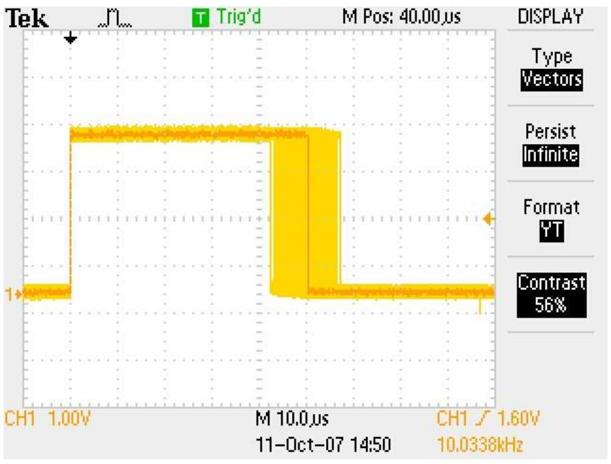


FIGURE 22: *Jitter on RTLinux v3.2-rc1*

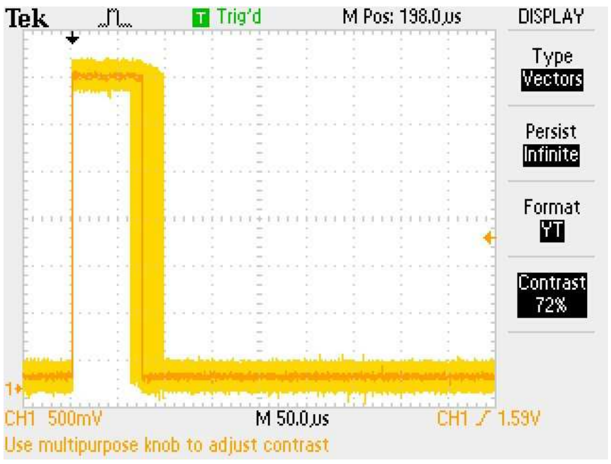


FIGURE 20: *Jitter on 2.6.21.5-rt20 CONFIG_PREEMPT_RT, VIA C3 1GHz*

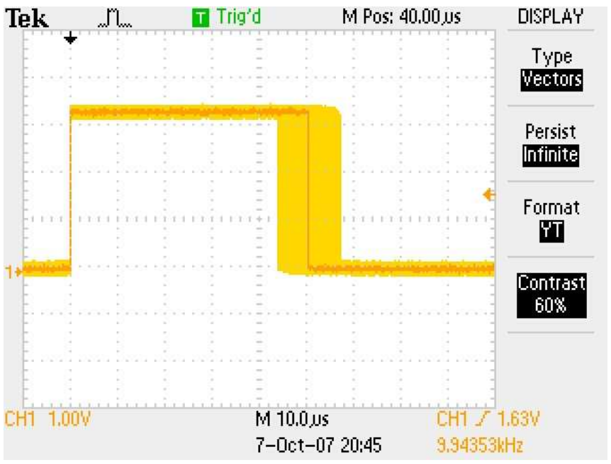


FIGURE 23: *Jitter on RTAI v3.5*

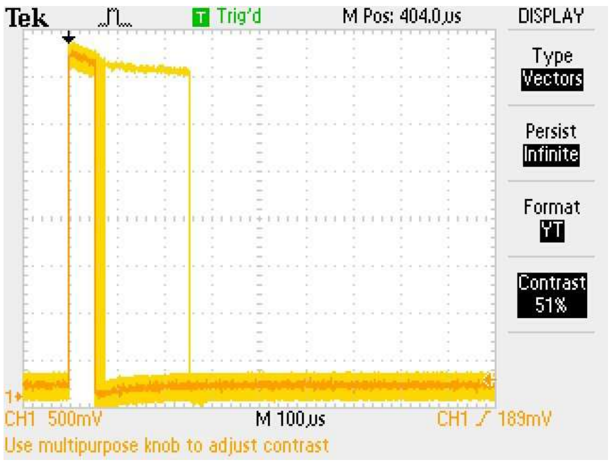


FIGURE 21: *Jitter on 2.6.21.5-rt20 CONFIG_PREEMPT_RT, P4*

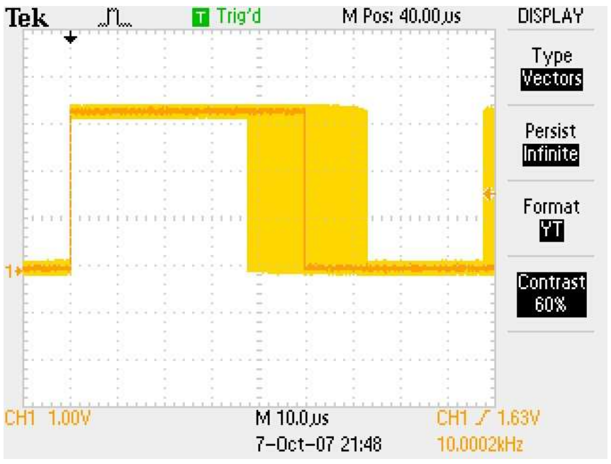


FIGURE 24: *Jitter on LXRT (RTAI v3.5)*

4 Discussion

4.1 LMbench Results

On average, it can be said that PREEMPT_RT has the most negative impact on the system albeit not by significant margin. Its impact on general performance of the system can clearly be seen from figures 2 and 4. While figures 3 and 8 also - less clearly - reflect this, it is very difficult to judge its impact from figures 6 and 7. Unlike results obtained (earlier) from running tests against previous patched kernel version e.g. 2.6.14-rt20, where the performance of these kernels were significantly below that of the unpatched versions, we see no significant differences here. Apperently PREEMPT_RT has no significant degrading impact on the general performance of the system in its current version.

4.2 Jitter Patterns

The main objective here was to access determinism of scheduling of PREEMPT(_RT) and contrast results with those obtained by running RTLinux, RTAI/LXRT. We wanted to scope how well an rt thread could react to an event; how well it could "time stamp" the event. The jitter here would serve as an indicator of how strong other tasks in the overall system affect the determinism of the real time thread - how deterministically the system could control an external signal.

As can be seen from figures 9 and 10 i.e. 2.6.23-rc1 with CONFIG-_PREEMPT_NONE on a VIA C3 600MHz and a P4 2.8GHz, we can't even begin thinking of any hard realtime - take note of the time scales involved and how things even get worse on the low end VIA C3 600 MHz board; the best our rt thread (that demanded scheduling at 50us) could get scheduled on the P4 at 1ms while on the VIA 600MHz, 2ms. Fig 11 and 12 show jitter on a VIA C3 600MHz and a P4 2.8GHz, respectively, with CONFIG_PREEMPT on 2.6.23-rc1. We see some improvement in preemptiveness but still things dont even come close to any hard real time (P4 still giving 1ms and VIA C3 600MHz still at 2ms). Now, with PREEMPT_RT enabled, we see a dramatic improvement in schedubility of the real time threads - figures 13 - 21. We now approach the 50us - even with the low end VIA boards. Note the general trend: The faster the processor, the better the real time performace: About 55 - 60us on the P4 2.8GHz, 70us on the VIA C3 1GHz and 80us on the VIA C3 600MHz board as illustrated below.

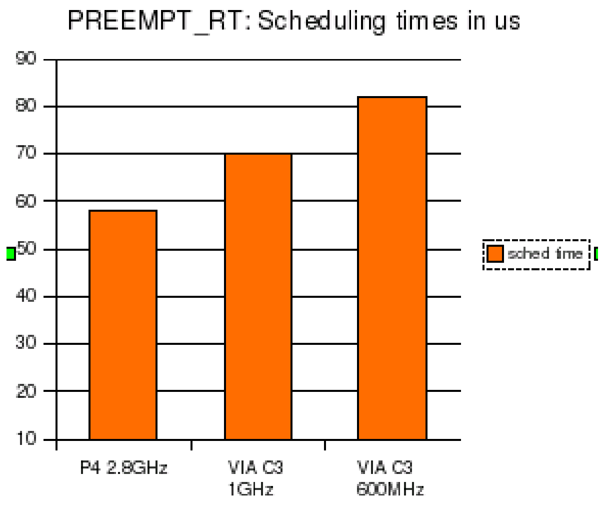
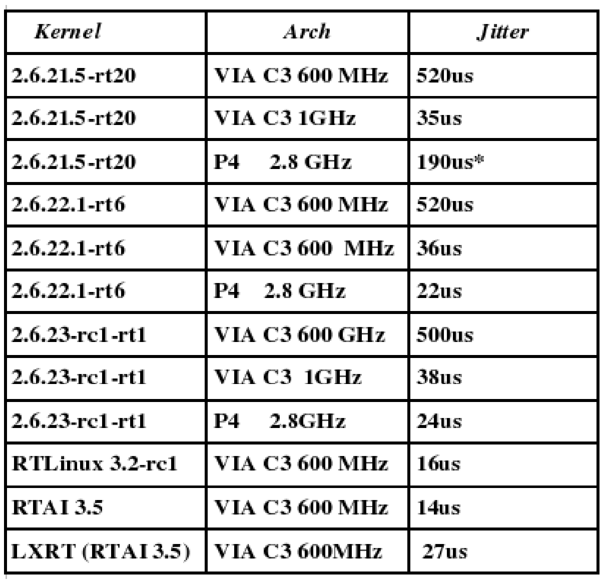


FIGURE 25: *Scheduling times on P4 2.8GHz and VIA 1GHz, 600MHz for an rt thread demanding 50us*

Now, even though we don't hit the 50us mark with precision i.e. we have some visible scheduler latency - and though we can't talk of hard real-time here in the strict sense of the word - we definitely see a tremendous reduction in jitter along with significant improvement in schedubility. Moreover, better real time performance can be achieved with relaxed timming constraints. However, we can't guarantee worst cases here. "Guarantee" in the sense that it is simply not posible to test all control paths in a real time application might take and also due to the overall complexity of the the implementation.

The rt.wiki recommends usage of ACPI for kernels above 2.6.18. However, we didn't see significant change in performance with this option enabled on these architectures.

With the interrupt abstraction techniques, , figures 22 - 24, we actually strike 50us with some serious precision. The maximun jitter about this mark is somewhat symmetrical for all and can actually be determined unlike with RT_PREEMPT. This is true hard real time in the sense of guaranteed worst case execution delays. The table below summarises worst case jitter for PREEMPT_RT versus its interrupt abstraction homies.



Kernel	*Arch*	*Jitter*
2.6.21.5-rt20	VIA C3 600 MHz	520us
2.6.21.5-rt20	VIA C3 1GHz	35us
2.6.21.5-rt20	P4 2.8 GHz	190us*
2.6.22.1-rt6	VIA C3 600 MHz	520us
2.6.22.1-rt6	VIA C3 600 MHz	36us
2.6.22.1-rt6	P4 2.8 GHz	22us
2.6.23-rc1-rt1	VIA C3 600 GHz	500us
2.6.23-rc1-rt1	VIA C3 1GHz	38us
2.6.23-rc1-rt1	P4 2.8GHz	24us
RTLinux 3.2-rc1	VIA C3 600 MHz	16us
RTAI 3.5	VIA C3 600 MHz	14us
LXRT (RTAI 3.5)	VIA C3 600MHz	27us

FIGURE 26: *Table showing worst case jitter for PREEMPT_RT, RTLinux, RTAI and LXRT*

*Now, due to PREEMPT_RT's implmentation, mad glitches in jitter can occur at any time e.g. fig 21. The (PREEMPT_RT) results displayed above are not meant to be conclusive but rather serve as an indicator of the effect of PREEMP_RT.

5 Conclusion and Future Work

As per the results obtained from running LMbench against the above mentioned kernels, each with different configuration settings, we see no significant impact of PREEMPT_RT on the general performance of the system unlike the preempt patches of earlier kernel versions.

PREEMPT_RT significantly improves preemptiveness in Linux. Though we may not -yet - speak of deterministic behaviour or guarantee worst case as with its interrupt abstraction (RTLinux, RTAI, L4-Fiasco, Xenomai, XtratuM etc) counterparts it does bring some significant real-time to Linux. From the jitter images above, we could say that PREEMPT_RT is not "definitely unsuitable", but we are not claiming suitability. However, CONFIG_PREEMPT and CONFIG_PREEMPT_VOLUNTARY are definitely not suitable for hard real time.

As for future work, we need fix systematic problems with the above approach before extending to further architectures. Though it should be noted that the restrictions of test run times is also due to the large number of configurations that were to be tested, reducting these will most likely be necessary in future runs. A comparison of other results with other benchmarks (AIM 7, bonny++) etc, and posibly application sector specific benchmark suits would be a necessary extension.

References

[1] *http://www.linuxdevices.com/articles/AT7005360270.html*

[2] *http://www.linuxdevices.com/articles/AT8211887833.html*

[3] *http://www.linuxdevices.com/articles/AT7554348551.html*

[4] *Victor Yodaiken, RTLinux Manifesto, 1999*

[5] *https://www.rtai.org*

[6] *http://os.inf.tu-dresden.de/fiasco*

[7] *http://home.gna.org/adeos/*

[8] *http://www.xenomai.org*

[9] *http://www.linuxdevices.com/articles/AT7554348551.html*

[10] *http://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch*

[11] *http://rt.wiki.kernel.org/index.php/High_resolution_timers*

[12] *http://www.captain.at/howto-linux-real-time-patch.php*