How fast is fast enough? Choosing between Xenomai and Linux for real-time applications

Dr. Jeremy H. Brown
Rep Invariant Systems, Inc.
38 Cameron Ave, Suite 100, Cambridge, MA, 02140, USA
jhbrown@repinvariant.com

Brad Martin
Rep Invariant Systems, Inc.
38 Cameron Ave, Suite 100, Cambridge, MA, 02140, USA
bmartin@repinvariant.com

Abstract

We needed data to help ourselves and our clients to decide when to expend the extra effort to use a real-time extension such as Xenomai; when it is sufficient to use mainline Linux with the PREEMPT_RT patches applied; and when unpatched mainline Linux is sufficient.

To gather this data, we set out to compare the performance of three kernels: a baseline Linux kernel; the same kernel with the PREEMPT_RT patches; and the same kernel with the Xenomai patches. Xenomai is a set of patches to Linux that integrates real-time capabilities from the hardware interrupt level on up. The PREEMPT_RT patches make sections of the Linux kernel preemptible that are ordinarily blocking.

We measure the timing for performing two tasks. The first task is to toggle a General Purpose IO (GPIO) output at a fixed period. The second task is to respond to a changing input GPIO pin by causing an output GPIO pin's value to follow it. For this task, rather than polling, we rely on an interrupt to notify us when the GPIO input changes.

For each task, we have four distinct experiments: a Linux user-space process with real-time priority; a Linux kernel module; a Xenomai user-space process; and a Xenomai kernel module. The Linux experiments are run on both a stock Linux kernel and a PREEMPT_RT-patched Linux kernel. The Xenomai experiments are run on a Xenomai-patched Linux kernel.

To provide an objective metric, all timing measurements are taken with an external piece of hardware, running a small C program on bare metal.

This paper documents our results. In particular, we begin with a detailed description of the set of tools we developed to test the kernel configurations.

We then present details of a a specific hardware test platform, the BeagleBoard C4, an OMAP3 (Arm architecture) system, and the specific kernel configurations we built to test on that platform. We provide extensive numerical results from testing the BeagleBoard.

For instance, the approximate highest external-stimulus frequency for which at least 95% of the time the latency does not exceed 1/2 the period is 31kHz. This frequency is achieved with a kernel module on stock Linux; the best that can be achieved with a userspace module is 8.4kHz, using a Xenomai userspace process. If the latency must not exceed 1/2 the frequency 100% of the time, then Xenomai is the best option for both kernelspace and userspace; a Xenomai kernel module can run at 13.5kHz, while a userspace process can hit 5.9kHz.

In addition to the numerical results, we discuss the qualitative difficulties we experienced in trying to test these configurations on the BeagleBoard.

Finally, we offer our recommendations for deciding when to use stock Linux vs. PREEMPT_RT-patched Linux vs. Xenomai for real-time applications.

1 Introduction

We work with robotics, an inherently "real-time" discipline. Many of our customers need us to determine when it is sufficient to use a stock Linux distribution, and when we need to take the extra effort to seek additional real-time support from the PREEMPT_RT Linux patches, or from Xenomai, a real-time system that integrates with Linux to provide hard-real-time capabilities.

In this paper, we present a test suite we developed to characterize the performance and limitations of Linux and Xenomai for two benchmark real-time tasks. The first task is to toggle a General Purpose IO (GPIO) output at a fixed period. The second task is to respond to a changing input GPIO pin by causing an output GPIO pin's value to follow it. For this task, rather than polling, we rely on an interrupt to notify us when the GPIO input changes.

To provide an objective metric, we run processes on the *test system*, but measure their performance using an external *measurement system* which runs a C program on bare metal.

We present and discuss the specific numerical results for our first test platform, a popular embedded system called the BeagleBoard. We also discuss some of the difficulties we experienced in configuring and testing Linux and Xenomai on the BeagleBoard.

Finally, we present our thoughts on how to decide when to expend the effort to use Xenomai, and when to simply use stock Linux.

1.1 Categories of real-time

"Real-time" is an ambiguous term. Every time-sensitive application has its own requirements which are not easily captured by a simple description. For this paper, we have adopted the following specific definitions:

Soft : The real-time requirements should be met most of the time according to a subjective user interpretation. A traditional example is a process playing music on a desktop system. Soft real time performance is subjective, but generally adequate on typical Linux desktop systems.

Life-safety hard : The real-time requirements requirement must be met 100% of the time by the system. If violated, someone may be injured or killed, and/or substantial property damage may occur. We do not recommend any of the software evaluated in this paper for life-safety hard applications.

100% hard : The real-time requirements requirement should be met 100% of the time by the system. An example is a process control program, where timing failures result in product manufacturing defects.[1]

95% hard : The real-time requirements should be met at least 95% of the time. An example is a data collection system where data samples are invalid when the requirement is missed, but it is acceptable to lose some of the data.[2]

In the rest of this paper we limit our analyses to the 95% and 100% hard real-time categories.

1.2 Technology background

Linux is a general-purpose interactive operating system. It was designed to support multiple processes, running on a single processor. The default configuration is designed to optimize for total system throughput, rather than for interactivity or the ability to perform real-time work. A number of approaches have been taken to enhance Linux's utility in real-time contexts. [14] is a recent survey of approaches and actively-supported platforms. In this section, we limit ourselves to two basic approaches.

Making Linux more real-time: Various PREEMPT patches try to make the Linux kernel itself more real-time by reducing the durations for which high-priority operations can be blocked, at the cost of reducing overall throughput.

In kernel 2.6, the CONFIG_PREEMPT build flag makes most of the kernel preemptible, except for interrupt handlers and regions guarded by spinlocks. This allows interactive and/or high priority real-time tasks to run even when some other task is in the middle of a kernel operation. According to

[1] There is a cost tradeoff analysis here that is well outside the scope of this paper: how much will it cost to produce 100% hard real-time software, and how much will it save you in manufacturing defects over some time period? Does that pay off compared to, say, building 99% hard real-time software more quickly and at lower cost, and accepting a slightly higher defect rate?

[2] Note that this definition is still very general. It covers a system that misses one operation every 20 cycles, and a system that misses blocks of 20 operations every 400 cycles. For some applications, these are not equivalent systems! Consider a 20Hz control system for a robot helicopter — a 50ms outage once a second is going to be much more survivable than a one second outage every 20 seconds.

Label	Implementation	Real-time strategy
linux-chrt-user	Linux userspace	`chrt` used to invoke
xeno-user	Xenomai userspace	`rt_task_set_periodic` called; uses RTDM driver
linux-kernel	Linux kernelspace	implemented using hrtimers
xeno-kernel	Xenomai kernelspace	`rt_task_set_periodic` (kernel version) called

Table 1: Periodic task types

Label	Implementation	Real-time strategy
linux-chrt-user	Linux userspace	`chrt` used to invoke
xeno-user	Xenomai userspace	`rt_task_create` called; uses RTDM driver
linux-kernel	Linux kernelspace	implemented as top-half IRQ handler
xeno-kernel	Xenomai kernelspace	implemented as RTDM IRQ handler

Table 2: Response task types

[11], with the CONFIG_PREEMPT option "worst case latency drops to (around) single digit milliseconds, although some device drivers can have interrupt handlers that will introduce latency much worse than that. If a real-time Linux application requires latencies smaller than single-digit milliseconds, use of the CONFIG_PREEMPT_RT patch is highly recommended."

The CONFIG_PREEMPT_RT[12] patch, maintained separately from the primary Linux sources, adds harder real-time capabilities to Linux. It makes many spinlock-guarded regions preemptible, moves IRQ handlers into threads, and adds various other real-time features. When people refer to Real Time (RT) Linux, they typically mean a Linux kernel with the CONFIG_PREEMPT_RT patches applied.

Adding real-time under Linux: Rather than relying on improving Linux's ability to preempt, Xenomai[7, 8] adds a real-time subsystem underneath Linux, and exposes its capabilities through Linux.

At the bottom, Xenomai relies on the Adeos[1, 16] I-pipe software to receive hardware interrupts. Adeos passes these events to its software clients in priority order; the Xenomai system has higher priority than Linux. Thus, the Linux kernel receives only virtual interrupt events, and those only after higher-priority software (e.g. the Xenomai layer) has had an opportunity to respond first. Similarly, when the Linux kernel blocks interrupt handlers, it does so only for itself; high-priority Xenomai threads will receive their events from the I-pipe on schedule.

Xenomai has a host of usability features that are well outside the scope of this paper, including implementations of multiple real-time APIs; the ability to migrate threads between the non-real-time Linux domain into the real-time Xenomai domain; etc.

2 Measurement system design

It is common in the literature to report real-time test measurements made by the real-time system being tested. For objectivity, we prefer not to rely on self-measurement/self-reporting, so we developed a simple external, real-time measurement system. This system also serves as the source of input events for measuring response latency.

2.1 Architecture

We selected the Atmel AVR microcontroller as our measurement platform. AVRs are used on the popular Arduino series of hobbyist microcontroller boards.

2.2 Software

The measurement software is written in C, and compiled under the AVR Studio IDE.

Response mode: In RESPONSE mode, the measurement system waits a random interval from 3 μs up to a configurable maximum period, then lowers its output pin (i.e. the input to the test system) and measures how long until the test system lowers its output pin in response. It then immediately raises its output pin and waits for the test system to do the same before beginning another cycle. Measurements are taken on falling edges only. When instructed to stop, the measurement system reports a histogram of response latencies.

Periodic mode: In PERIODIC mode, the measurement system expects the test system to raise and lower a GPIO pin at a specified periodic rate. When instructed via the serial interface, the measurement system begins measuring the actual period between successive falling edges on its input pin.

When instructed to stop, the measurement system emits a histogram showing how many samples were measured at each actual offset, centered about the expected period.

Because the measurement system measures inter-falling-edge times, a single delay in test system ping generation produces two off-period measurements: one long measurement, followed by one short one. E.g., if the test system is supposed to generate falling edges every 1000μs, and it actually generates them at time T=0μs, T=1000μs, T=2050μs, T=3000μs, T=4000μs, the test system will report measurements of 1000μs, 1050μs, 950μs, and 1000μs.

Histograms: Histogram granularity depends on the expected or maximum period, and the memory available on the specific AVR selected. Along with each histogram, the system reports the number of outliers falling above and below the range covered by the histogram, along with the maximum and minimum values seen.

3 Test suite details

3.1 Hardware

Any platform capable of running Linux and Xenomai, and exposing at least two GPIO pins, is a viable candidate for our test suite.

3.2 Task implementations

We wrote code to implement both response and periodic tasks using four different methods: Linux userspace, Linux kernel, Xenomai userspace, and Xenomai kernel. Thus, we have 8 distinct sets of code.

We run the Linux userspace processes with real-time priority 99 (using `chrt 99`) to give them real-time scheduling priority.

For hardware platform independence, our code relies exclusively on standard Linux and Xenomai APIs.

For the response task, the code relies on an interrupt to detect GPIO input changes.

All userspace processes call `mlockall` to prevent paging.

Most of our Xenomai-related code is derived from example code distributed with the Xenomai source tree.

3.3 RTDM driver support

To support the Xenomai userspace tests, we wrote a small kernel module which provides the needed GPIO operations as an RTDM (Real Time Device Model) device driver; Xenomai userspace processes access GPIOs by reading and writing using `rt_dev_read` and `rt_dev_write`. The device driver relies on Xenomai's RTDM IRQ handling and the Linux GPIOlib interfaces. The core read and write routines are presented below; error handling has been omitted for brevity.

```
static ssize_t simple_rtdm_read_rt(
  struct rtdm_dev_context *context,
  rtdm_user_info_t * user_info, void *buf,
  size_t nbyte)
{
  int ret;
  rtdm_event_wait(&gpio_in_event);
  if (nbyte <= 0) return 0;
  ret = rtdm_safe_copy_to_user(
    user_info, buf,
    gpio_in_value ? "1" : "0", 1);
  if (ret) return ret;
  return 1;
}

static ssize_t simple_rtdm_write_rt(
  struct rtdm_dev_context *context,
  rtdm_user_info_t * user_info,
  const void *buf, size_t nbyte)
{
  int ret;
  char value;
  ret = rtdm_safe_copy_from_user(user_info, &value,
                                 buf+nbyte-1, 1);
  if (ret) return ret;
  gpio_set_value(gpio_out, (value == '0' ? 0 : 1));
  return nbyte;
}
```

3.4 Response task implementations

In the interrupt-response task, the test system is required to wait for an interrupt from an input GPIO pin. When that happens, the system must set an output GPIO pin to match the input signal. The input transitions happen at random intervals.

Our implementation approaches are summarized and labeled in Table 2; the labels are used in following graphs and tables.

We briefly describe each approach below, and

present a snippet of code modeling our approach. The snippets presented here are condensed for brevity and reorganized for clarity.

Linux userspace: This uses the GPIO pin interface devices in `/sys/class/gpio` to operate the pins. `in_value_fd` and `out_value_fd` refer to `/sys/class/gpio/gpio<N>/value` files.

```
for (;;) {
  int cnt = read(in_value_fd, buf, 16);
  write(out_value_fd, buf, cnt);
  int result = poll(&poll_struct, 1, -1);
  lseek(in_value_fd, 0, SEEK_SET);
}
```

Linux kernelspace: This uses the GPIOlib kernel interface to the GPIO pins in a typical Linux IRQ "top-half" handler.

```
irqreturn_t irq_func(int irq, void *dev_id) {
  int output_value = gpio_get_value(gpio_in);
  gpio_set_value(gpio_out, output_value);
  return IRQ_HANDLED;
}
```

Xenomai userspace: This uses blocking-read on our GPIO RTDM device driver (see Section 3.3.).

```
for (;;) {
  rt_dev_read(device, buf, 4);
  rt_dev_write (device, buf, 1);
}
```

Xenomai kernelspace: This uses the GPIOlib kernel interface to the GPIO pins, and Xenomai's RTDM (real-time) IRQ handling.

```
int irq_func(rtdm_irq_t *irq_handle) {
  int output_value = gpio_get_value(gpio_in);
  gpio_set_value(gpio_out, output_value);
  return RTDM_IRQ_HANDLED;
}
```

3.5 Periodic task implementations

In the periodic task, the test system is required to toggle the value of an external GPIO periodically, based on an internal timer. Our implementation approaches are summarized and labeled in Table 1; the labels are used in following graphs and tables.

We briefly describe each approach below, and present a snippet of code modeling our approach. As above, the snippets presented here are condensed for brevity and reorganized for clarity.

Linux userspace: The native Linux GPIO interface for userspace applications is based on reading from and writing to special files in `/sys/class/GPIO` using standard C read and write calls. Our code uses the Linux high-resolution timer interface for the period, and the GPIO pin interface devices in `/sys/class/gpio` to operate the output pin.

```
for (;;) {
  write(gpio_fd, output_value ? "1\n" : "0\n", 2));
  output_value = !output_value;
  read(timer_fd,
       &periods_elapsed,
       sizeof(periods_elapsed));
}
```

Linux kernelspace: This uses the kernelspace hrtimer interface for the period, and the Linux kernel's GPIOlib interface.

```
enum hrtimer_restart timer_func(struct hrtimer* timer) {
  gpio_set_value(gpio_out, output_value);
  output_value = !output_value;
  hrtimer_forward_now(timer, half_period);
  return HRTIMER_RESTART;
}
```

Xenomai userspace: This uses Xenomai's userspace RT_TASK with a periodic timing for the period, and our RTDM driver (see Section 3.3) which provides access to the GPIO pins via read and write operations.

```
while (1) {
  int size = rt_dev_write (
    device,
    output_value ? "1" : "0", 1);
  output_value = !output_value;
  rt_task_wait_period(NULL);
}
```

Xenomai kernelspace: This uses Xenomai's kernelspace RT_TASK with a periodic timing for the period, and the GPIOlib kernel interface.

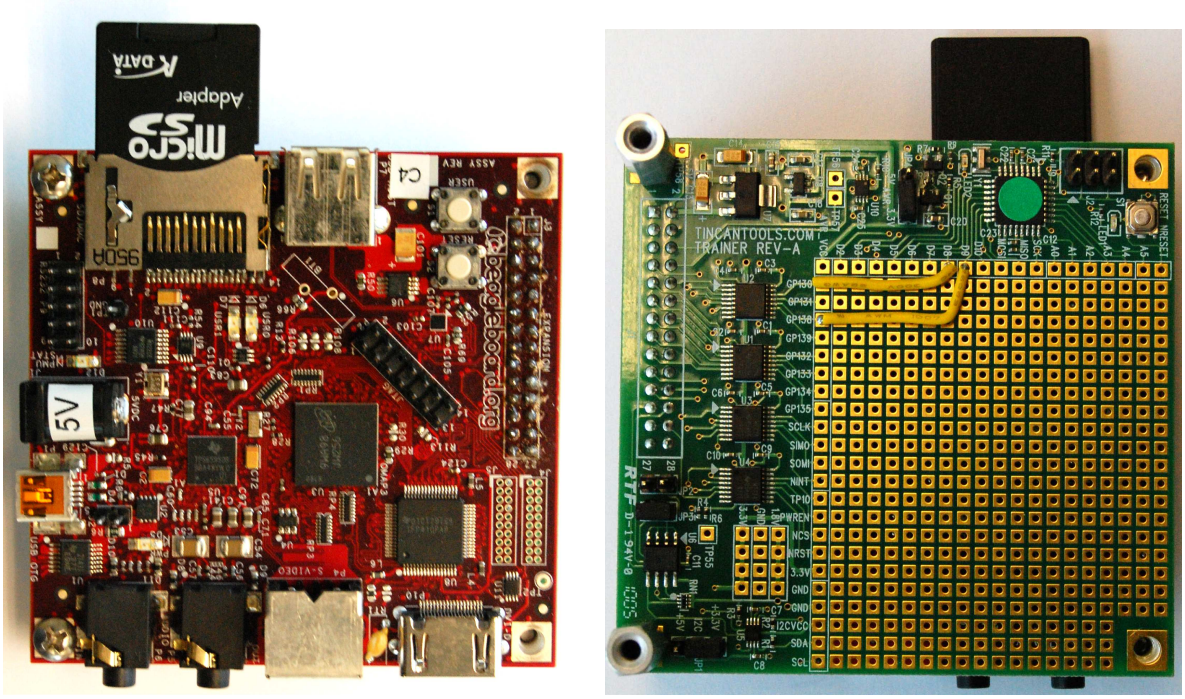
```
while (1) {
  gpio_set_value(gpio_out, output_value);
  output_value = !output_value;
  rt_task_wait_period(NULL);
}
```

3.6 Test control

A python script manages running each test variation and collecting the resulting logfiles. The script is responsible for GPIO mux and pin configuration, kernel module insertion, process initiation, priority management, and subsequent cleanup. Except for the selection of appropriate GPIO pin numbers, it is hardware platform independent and should work on any Linux-based system.

We run the script with real-time priority 10 to ensure timely operations even on a heavily-loaded system.

The control script reprioritizes pre-existing real-time processes to a maximum real-time priority of



(a) BeagleBoard Rev C4 OMAP3 Single Board Computer (SBC)

(b) BeagleBoard Trainer daughtercard with AVR-OMAP I/O wiring.

Figure 1: Test system hardware components

Label	Kernel	Patches	Version	Source
stock	"Stock" Linux	None	2.6.33.7	[4]
RT	Real-time Linux	CONFIG_PREEMPT_RT	patch-2.6.33.7-rt29	[10]
xeno	Xenomai-enabled	Adeos (Xenomai)	2.6.33-arm-1.16-01 (post 2.5.3 HEAD)	[6]

Table 3: Operating system configurations

80. On the RT kernel, for the response task, an IRQ thread is spawned for the input GPIO pin; and for the periodic task, a soft IRQ thread is spawned for the hrtimers. In both cases, the control script bumps the IRQ thread priority to 99.

3.7 Data analysis

A Python script built on the matplotlib python libraries parses the logfiles from test runs. It generates the charts and tables found throughout this report.

3.8 Test load

A bash script starts up heavy loads for the system. It runs a combination operations that, empirically, load the test system quite heavily:

- It launches a subprocess that tight-loops invoking `scp` to copy a large file from and to a USB-attached hard drive via the loopback network interface.
- It launches a subprocess that tight-loops invoking `dd` to copy a large file from an SD card to `/dev/null`
- It launches 500 instances of dd copying from /dev/zero to /dev/null. These are run at nice level 20 to preserve a modicum of system interactivity.

4 Profiling the BeagleBoard

4.1 The hardware

For our initial test platform, we used a BeagleBoard Rev C4 OMAP3 Single Board Computer (SBC), shown in Figure 1(a)). The OMAP3 microprocessor is an ARM architecture. The board runs at 720MHz.

To run our measurement system, we selected the BeagleBoard Trainer daughtercard, shown in Figure 1(b). The Trainer features an Atmel AVR, logic level shifters enabling direct connections between the AVR and the OMAP3's signals, and a protoboard space for making those connections. We used the protoboard space to wire the two OMAP GPIO signals to IO ports on the AVR.[3]

We used the OMAP's GPIO 130 for output in all tests, and GPIO 138 for input and corresponding interrupt generation for response tests. These GPIO signals run straight to the OMAP, with no (external) busses to bottleneck.

[3]Caution: many of the pads on the Trainer protoboard area are mis-labeled!

4.2 Kernel configurations

We installed the Ubuntu Lucid Linux distribution on the BeagleBoard, using the demo root fs image from [3]. We ran experiments against three distinct kernel configurations, summarized in Table 3. All three configurations were built using the CodeSourcery cross-compilation toolchain. The stock configuration is built with CONFIG_PREEMPT enabled. The Adeos patches used in the xeno configuration are included as part of the Xenomai sources.

4.3 Response experiments

Test procedure: Each experiment was run for two hours. Each was run on the same hardware.

We configured the measurement system to issue stimuli at random intervals from $3\mu s$ to $7071\mu s$. At this period, the output histograms have about 600 buckets with granularity of 1 μs.

Results: Table 4 shows the basic statistics for the experiments.

Figure 2 presents detailed performance data for each experiment, graphed on a linear scale. Dashed lines indicate the envelopes within which 95% of measurement samples occurred.

The 95% and 100% values are in some cases separated by orders of magnitude. Figure 3 plots these values on a log scale.

4.4 Periodic experiments

Test procedure: Each experiment was run for two hours. Each was run on the same hardware.

We ran the periodic task experiments with a period of $7071\mu s$. This is a semi-arbitrary period chosen to be unlikely to align itself with periodic system activities. With this period, the measurement system granularity is 1 μs per bucket, with about 600 buckets.

Results: Table 6 shows the basic statistics for the experiments.

Figure 4 presents detailed performance data for each experiment, graphed on a linear scale. Dashed

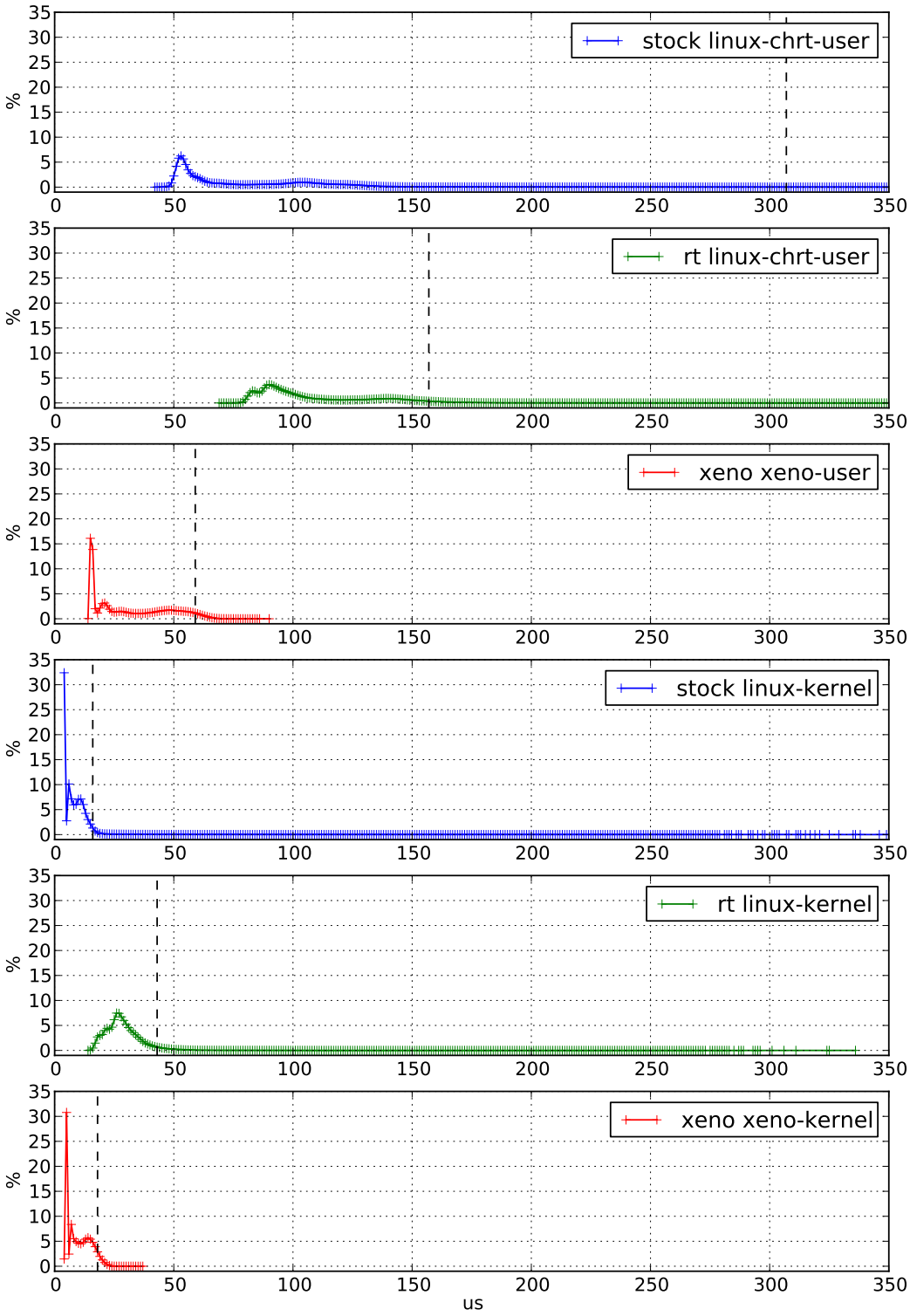


Figure 2: cross-configuration response experiments: time from GPIO input change to GPIO output change. A dashed vertical line denotes the region containing 95% of the samples for that experiment.

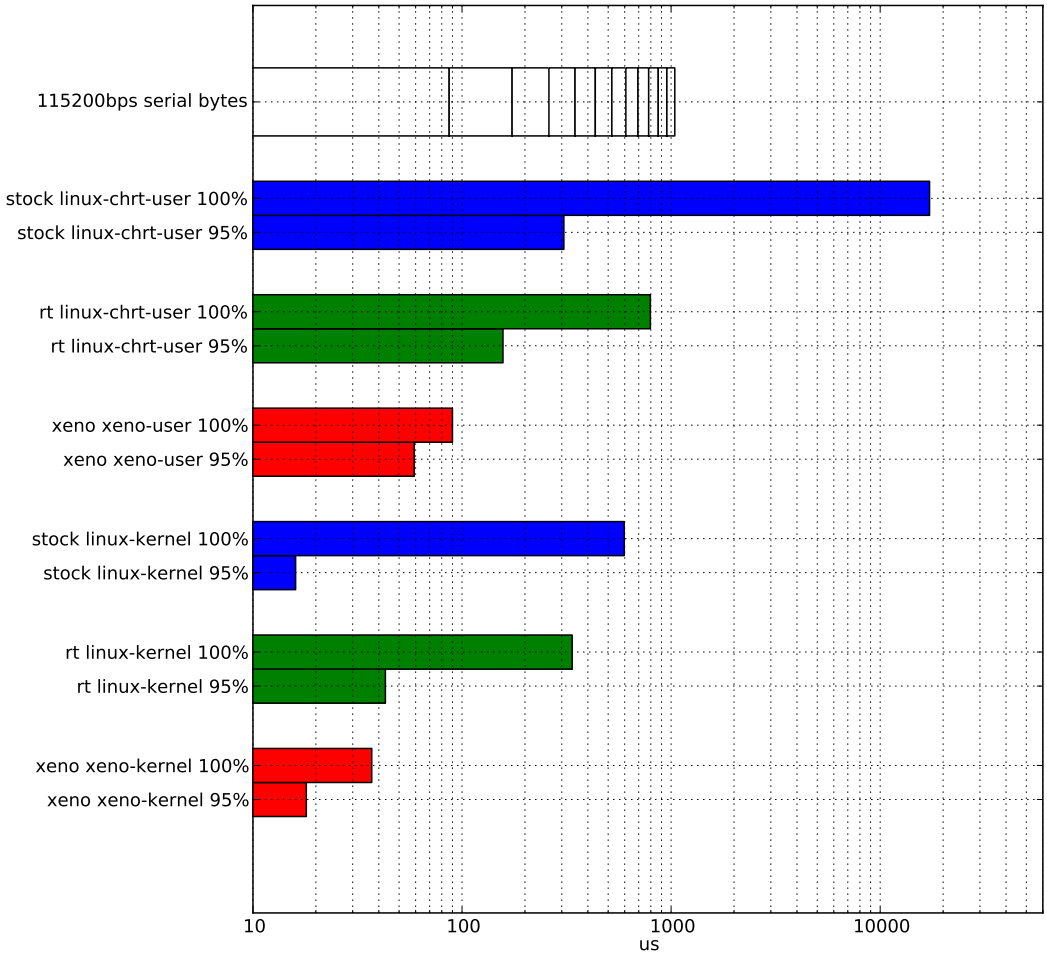


Figure 3: cross-configuration response experiments: maximum and 95% envelope response times plotted on a log scale. Transmission time for 12 serial bytes at 115200bps ($87\mu s$each) is plotted for comparison.

Config	Experiment	# samples	Median	95%	100%
stock	linux-chrt-user	1840823	$67\mu s$	$307\mu s$	$17227\mu s$
rt	linux-chrt-user	1849438	$99\mu s$	$157\mu s$	$796\mu s$
xeno	xeno-user	1926157	$26\mu s$	$59\mu s$	$90\mu s$
stock	linux-kernel	1259410	$7\mu s$	$16\mu s$	$597\mu s$
rt	linux-kernel	1924955	$28\mu s$	$43\mu s$	$336\mu s$
xeno	xeno-kernel	1943258	$9\mu s$	$18\mu s$	$37\mu s$

Table 4: cross-configuration response experiments: latency from input GPIO change to corresponding output GPIO change.

Config	Experiment	95% period	100% period
stock	linux-chrt-user	1.63 kHz	0.03 kHz
rt	linux-chrt-user	3.18 kHz	0.63 kHz
xeno	xeno-user	8.47 kHz	5.56 kHz
stock	linux-kernel	31.25 kHz	0.84 kHz
rt	linux-kernel	11.63 kHz	1.49 kHz
xeno	xeno-kernel	27.78 kHz	13.51 kHz

Table 5: cross-configuration response experiments: approximate highest frequency possible for which latency does not exceed 1/2 period, for 95% and 100% cases.

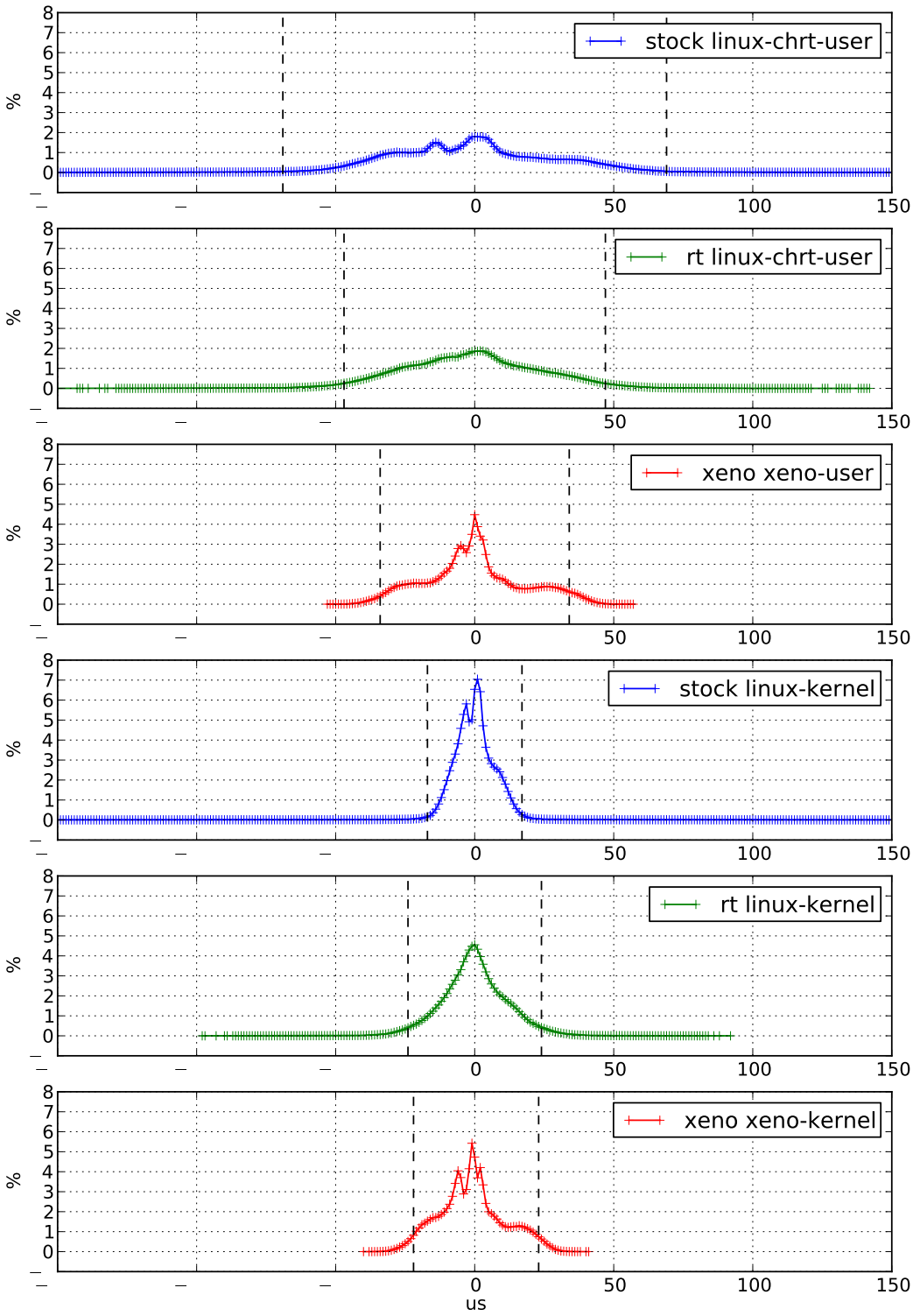


Figure 4: cross-configuration periodic experiments: timing jitter. Dashed vertical lines denote the region containing 95% of the samples.

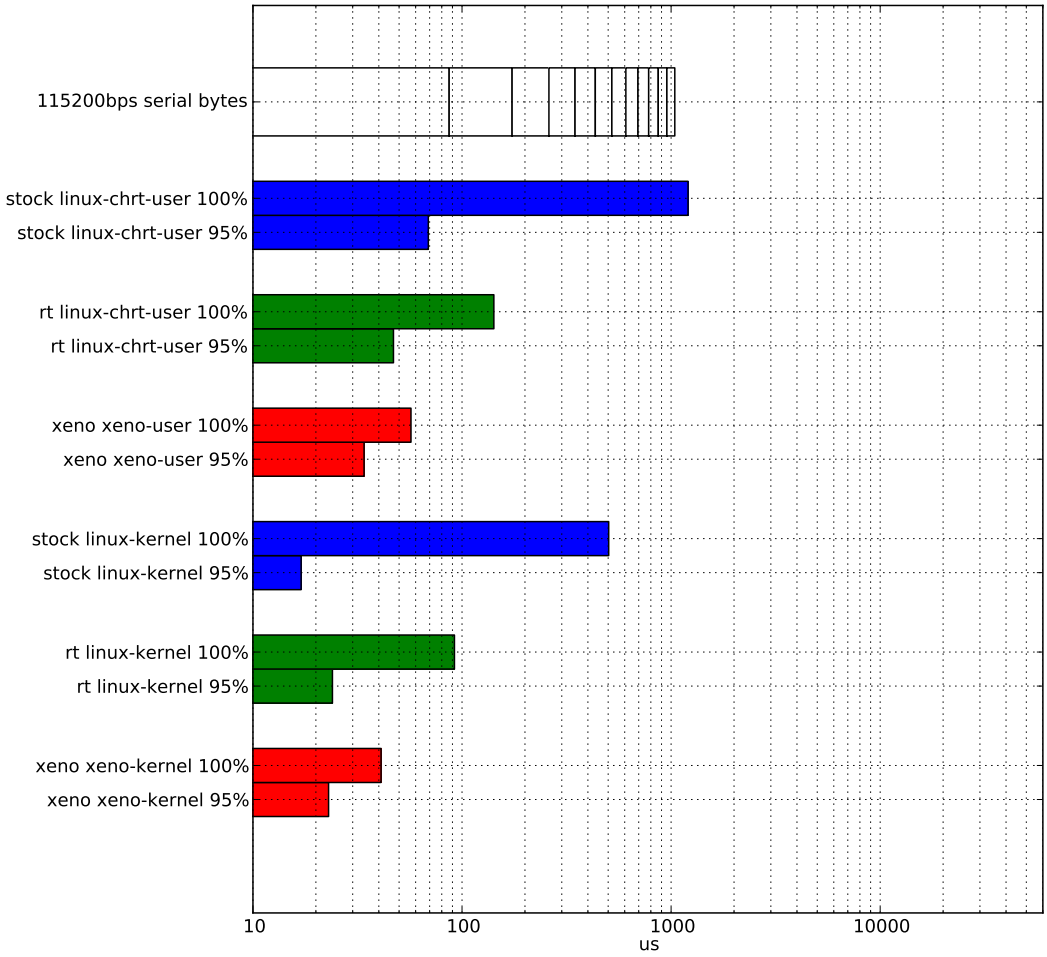


Figure 5: cross-configuration periodic experiments: maximum and 95% envelope jitter magnitudes plotted on a log scale. Transmission time for 12 serial bytes at 115200bps (87μseach) is plotted for comparison.

Config	Experiment	# samples	Median	95%	100%
stock	linux-chrt-user	1018391	-2μs	69μs	1205μs
rt	linux-chrt-user	1018387	-1μs	47μs	158μs
xeno	xeno-user	1018318	-1μs	34μs	57μs
stock	linux-kernel	1018255	0μs	17μs	504μs
rt	linux-kernel	1018249	0μs	24μs	98μs
xeno	xeno-kernel	1018449	-1μs	23μs	41μs

Table 6: cross-configuration periodic experiments: jitter relative to expected time period between successive falling edges. 95% and 100% are absolute values.

Config	Experiment	95% period	100% period
stock	linux-chrt-user	7.25 kHz	0.41 kHz
rt	linux-chrt-user	10.64 kHz	3.16 kHz
xeno	xeno-user	14.71 kHz	8.77 kHz
stock	linux-kernel	29.41 kHz	0.99 kHz
rt	linux-kernel	20.83 kHz	5.10 kHz
xeno	xeno-kernel	21.74 kHz	12.20 kHz

Table 7: cross-configuration periodic experiments: approximate highest frequency possible for which jitter does not exceed 1/2 period, for 95% and 100% cases.

lines indicate the envelopes within which 95% of measurement samples occurred.

The 95% and 100% values are in some cases separated by orders of magnitude. Figure 5 plots these values on a log scale.

4.5 Implementations across configurations

To evaluate the impact of the PREEMPT_RT patches and Xenomai patches on native Linux code, we ran the Linux user-space and kernel-module implementations on the Xenomai-patched kernel as well; the results for all three configurations are shown side-by-side in Tables 9 and 8.

Across all experiments, the RT kernel has significantly better (lower) maximums (100% marks) than do the stock kernel or the Xenomai kernel. It also significantly outperforms the other two kernels on the 95% mark for the Linux userspace implementations. The PREEMPT_RT authors are clearly succeeding in limiting maximum scheduling latency.

The response tests median column suggests that the scheduling overhead due to either patch set is significant. The Xenomai patches add about 50% to median kernelspace response time, and about 25% to median userspace response time. RT has more impact: it adds 300% to median kernelspace response time, and 48% to median userspace response time.

The periodic tests median column conveys less information, since all three kernels have excellent median periodic timing. The 95% column indicates a wide range of variation at this more stringent mark. The Xenomai kernelspace jitter is 65% worse than the stock kernel jitter, while the RT kernel is 41% worse. The RT userspace is actually the best of the three at the 95% mark, with 17% less jitter than the stock kernel; Xenomai has 85% worse jitter than the stock kernel.

4.6 Qualitative testing difficulties

Along the path to generating the numerical test results presented in this section, we ran into a number of qualitative difficulties. These impact our overall assessment of when to choose which system.

Stock Linux: As described in Section 4.7, the stock Linux kernel would occasionally (roughly once every 16 hours) complain of spurious IRQs; this event would correspond with several unusually large response latencies.

Real-time Linux: The RT configuration was our most problematic.

The primary RT target platform is x86; ARM support is not as mature. To get our system to boot without warnings, we had to disable the real-time clock (RTC) module, since the code path for setting it up on the BeagleBoard tries to sleep in an invalid context.

To get our tests to work without generating kernel warnings, we had to convert a small number of GPIO-related spin-locks into raw spin locks.

Our original test loads (see Section 3.8) included 1000 NICE'ed dd processes, but that routinely caused the RT kernel to lock up; we reduced the count to 500, and that seemed to solve the problem.

Even with the reduced number of processes, with the test loads running, the RT kernel becomes much less responsive than the other two kernels were even with the higher load.

Xenomai: We initially patched our kernel with Xenomai release 2.5.3.

Unfortunately, that version suffered from an inability to tolerate IRQ pressure; this led to kernel panics when the test system changed the GPIO input too rapidly. Once we had figured this out, and read through the Xenomai-help archives to find similar problems, we checked out the Xenomai source repository head which included a fix for the IRQ pressure bug.

As we began running our tests with a heavily-loaded system, however, we began to see kernel warnings and oopses. Gilles Chanteperdrix, via the xenomai-help mailing list, provided a one-line patch that resolved this problem by making it safe to call gpio_set_value from a Xenomai context.

4.7 Discussion

Consistency: The Xenomai implementations stand out for having by far and away the smallest difference between their 95% and 100% hard performance measurements. Non-Xenomai implementations show factors of 6 or more (often much more) between 95% and 100% performance; Xenomai implementations are factors of 2 or less.

Config	Experiment	# samples	Median	95%	100%
stock	linux-chrt-user	1840823	67μs	307μs	17227μs
rt	linux-chrt-user	1849438	99μs	157μs	796μs
xeno	linux-chrt-user	1842548	84μs	259μs	23645μs
stock	linux-kernel	1259410	7μs	16μs	597μs
rt	linux-kernel	1924955	28μs	43μs	336μs
xeno	linux-kernel	1939653	11μs	26μs	537μs

Table 8: Comparison of response linux-based experiments across configurations.

Config	Experiment	# samples	Median	95%	100%
stock	linux-chrt-user	1018391	-2μs	69μs	1205μs
rt	linux-chrt-user	1018387	-1μs	47μs	158μs
xeno	linux-chrt-user	1018450	-2μs	87μs	923μs
stock	linux-kernel	1018255	0μs	17μs	504μs
rt	linux-kernel	1018249	0μs	24μs	98μs
xeno	linux-kernel	1018307	0μs	28μs	743μs

Table 9: Comparison of periodic linux-based experiments across configurations.

95% hard performance: Stock Linux kernelspace has the best 95% hard performance, and stock Linux userspace the worst. Xenomai userspace is substantially better than Linux userspace run on either stock or RT, and performs within a factor of 2-4 of Linux kernelspace. Linux userspace implementation performs better on RT than on stock.

100% hard performance: Xenomai kernelspace is the best performer. Correspondingly, the naive thing to do is implement every real-time application in Xenomai kernelspace. However, this is also the most labor- and maintenance- intensive approach, so we would not recommend it as a default strategy! Xenomai userspace performs around a factor of 2 slower than kernelspace, but still handily outperforms all non-Xenomai implementations, in most cases by factors of 5 or more.

Bugs, priorities, and the meaning of "100%": A couple of times during testing "dry runs", we saw surprisingly high 100% measurements (over 3000μs) for the Linux userspace response test running on the stock kernel. In one such run, not only did the 100% mark go surprisingly high, but it was also impossible to compute the 95% mark because so many outliers fell beyond our measurement system's histogram range.

Upon investigation, we found that during the experiments in question, the system reported a glitch: `Spurious irq 95: 0xffffffdf, please flush posted write for irq 37`.

In more recent tests, including those reported on here, we again saw that glitch. However, the test results did not show the high latency we had previously seen.

The major configuration difference between the older tests and the newer ones is a change in our procedure: previously, we had run our tests with a real-time priority of 50, not 99, and we did not adjust the priority of other real-time processes (e.g. `watchdog`) downward.

Even before adjusting the realtime priorities, the glitch did not happen in every run. This goes to illustrate the point that a low 100% value in one run is no guarantee of hitting the same low mark in another — and a low mark on a 2 hour test is no guarantee of hitting the same low mark over a 365-day operational time once deployed!

Serial data: Serial devices are common in robotics and other real-time applications.

Consider a serial device running at 115,200bps (the maximum "standard" baud rate). It can send data, say a sensor measurement, at a maximum rate of 11.5kBps. If a single measurement's data packet contains 12 bytes (provided for comparison in Figures 5 and 3), the device can send packets at just under 0.1kHz (i.e. period of just over 10,000μs).

In this case, by the time the entire packet has been transmitted, the data itself is a full period old. The system designer would have to consider how much additional latency or jitter will affect system

performance after such an up-front latency.

On the BeagleBoard, a userspace process running with real-time priority on stock Linux real-time can respond to streaming data at the 0.1kHz period for 95%-hard performance. However, for a 100% hard requirement, a userspace process running on stock Linux is inadequate – but RT looks like a viable solution even for the 100% case.

Frequency view: The discussion about serial data introduced frequency as a way of considering the performance implications of each implementation and configuration.

Table 5 shows the stimulus-response frequencies that can be achieved subject to the arbitrary requirement that response latency may not exceed 1/2 period. Similarly, Table 7 shows the periodic event frequencies that can be achieved on the BeagleBoard subject to the arbitrary requirement that jitter may not exceed 1/2 period.

As noted, our 1/2 period assumptions are quite arbitrary. Each real-time application is likely to have its own requirements, leading to correspondingly different practical frequency limits.

5 Recommendations

Based on our experiments with the BeagleBoard, we offer the following recommendations on how to determine which implementation approach to use for your real-time application.

5.1 BeagleBoard

Let's assume that you are implementing a real-time application to run on a BeagleBoard, operate its GPIO pins, and share the board with a heavy non-real-time processing load. In that case, Figure 6 captures our recommended decision procedure.

One note: for 100% hard applications, we immediately steer you into a Xenomai world. Xenomai separates the real-time interrupt-handling paths from the complexities of the Linux kernel. We believe this will reduce the likelihood of rare and irreproducible events that cause timing requirements to be violated. The benefits of this separation are reflected in the consistency of Xenomai performance — that is, in the relatively small range between median and 100% performance numbers on Xenomai's numbers across periodic and response tasks, and userspace and kernelspace.

5.2 Everything else

While we believe that the merits of the three platforms will probably stay relatively constant across variations, they may vary somewhat with hardware, OS version, system load, application, etc. Thus, if you are considering another hardware platform, application function, or you have a substantially different metric, you may want to generate your own decision flow-chart.[4]

The most important thing is to profile your candidate hardware and kernel; you can re-use the structure of Figure 6; just preface each implementation with a profiling step. E.g., for a 95% hard task, start with stock Linux, and run a synthetic benchmark user-space process which in some fashion resembles your ultimate application. Provide a synthetic system load which resembles your anticipated actual load. Collect data for metrics which you can directly relate to your application requirements. After that, if you're marginal, profile an RT kernel, and so on.

The goal of this exercise isn't to generate high-precision numbers at each step. Instead, the goal is to determine whether you're clearly in the black, marginal, in trouble, or off the map. If you're completely off the map, you can probably move straight to dedicated hardware.

5.3 Other considerations

The cost of custom: If your application isn't throw-away, consider long-term maintenance overhead cost. In all likelihood, nobody is distributing pre-built, patched kernels for your platform, so using RT or Xenomai configurations commits you to maintaining a custom kernel in-house.

Building custom-patched kernels is time consuming, and you will almost certainly encounter more bugs than you will in stock Linux. Identifying, classifying, and resolving those bugs is time consuming.

Furthermore, since patches are developed and distributed separately, patch sets may lag the stock kernel head.

Device driver: Do you have to write a new device driver to use your hardware with Xenomai?

Do you have to modify or rewrite an existing Linux device driver to get it to work reliably in an RT configuration?

[4] Or hire us to generate it for you!

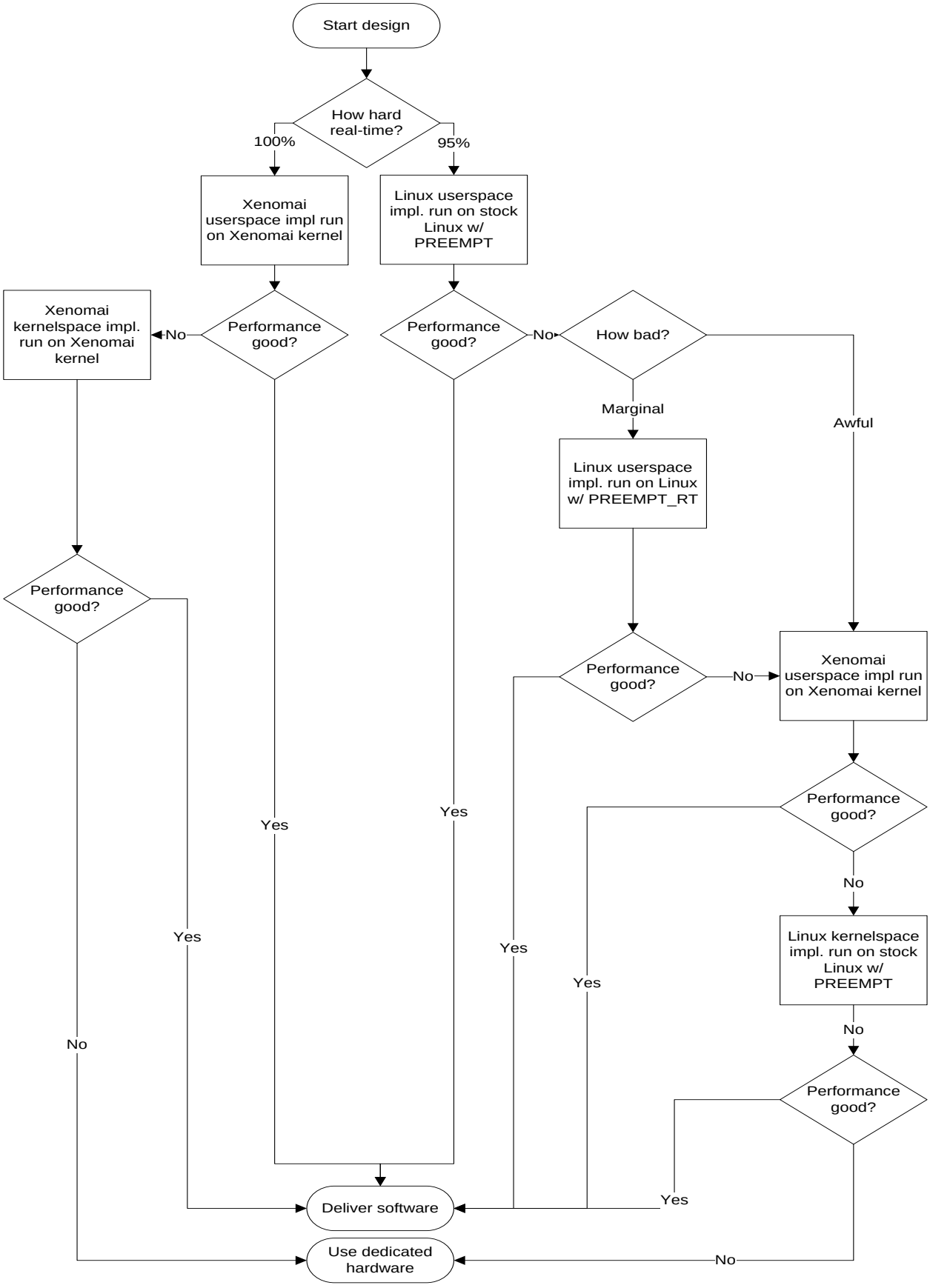


Figure 6: Flowchart for selecting an real-time application implementation strategy.

Do you have to write a completely new device driver no matter what, e.g. for custom hardware?

These may have a heavy influence on your choice of configuration.

6 Related work

A complete review of benchmarking literature is well beyond the scope of this paper. Here we discuss some directly relevant prior work evaluating Linux and/or Xenomai systems.

[2] compares RTAI, VxWorks, Xenomai, and stock Linux performance for use in a nuclear fusion application. They perform three different benchmarks. The first is directly comparable to our responsivity benchmark. The test system is required to change an output DAC line when an input ADC line changes. The metrics are latency from input to output, and latency jitter. The measurements are taken by oscilloscope applied to the lines. All implementations are kernel mode. All code runs on a Motorola MVME5500 (PowerPC architecture).

The reported latencies range from $69.2\mu s$ (VxWorks) to $73.2\mu s$ (Xenomai); jitter is sub-$1\mu s$ in all cases. Xenomai is outperformed by the stock Linux kernel (which is, in turn, outperformed by RTAI and VxWorks.)

The paper does not report how many measurements were taken for each configuration. The system is unloaded for all reported numerical measurements, although the authors comment that Linux performance measures "hold only for a system which is not loaded, and soon decrease when the workload increases."

While the paper does not report how latency and jitter are calculated, the sub-$1\mu s$ jitter values seem qualitatively different from the variations we observed in testing the BeagleBoard. As shown in Table 4, in our testing, the Xenomai kernel response implementation showed nearly a factor of 4 difference between median response ($9\mu s$) and slowest response ($37\mu s$.) Furthermore, note that while the Linux kernel does outperform the Xenomai kernel on a 95% basis in our results, the converse is true on a 100%-basis. Based on these distinctions, we suspect that the measurement methodology and sampling duration used in [2] have limited validity in deciding whether any of the measured systems can be used for 100%-hard nuclear fusion control.

The second benchmark described in [2] is a similar latency test; however, the input thread notifies a second thread to perform the output write. The additional latency, compared with the first experiment, is determined to be the scheduling overhead, with a maximum of under $6\mu s$ on stock Linux. The third experiment involves separating the input and output functions onto separate computers; the input system sends a UDP packet to the output system over gigabit Ethernet. The latency reported ranges from $101\mu s$ on RTAI+RTnet to to $157\mu s$ on VxWorks. [9] extends and deepens the real-time networking comparisons.

[13] reports on a robot control application that has an $80\mu s$ hard real-time latency requirement for small IEEE 1394 (FireWire) data transactions. This paper compares a Xenomai userspace and an RTLinux Pro kernelspace query/response implementations. It reports that for a 4-byte request/response, Xenomai has a $61\mu s$ latency while RTLinux Pro has a $58\mu s$ response. Jitter is not reported.

[15] describes common sources of latency in Linux x86 systems. It also includes a number of (self-measured) latency results based on outputs to one parallel port pin which is wired to another parallel port pin used for input. Latency is the time from when the system stimulates the output pin to when it handles an incoming interrupt from the input pin. All measurements are strictly reported against stock Linux; however, the computational load and hardware are varied, resulting in dramatically different responsivity histograms.

[5], performed at the same institution two years later, is in some ways similar to our present work. It reports (self-measured) responsivity experiments run using a parallel-port loopback, as well as periodic activity tests with internal measurement. It reports results across a kernel and userspace implementations for Linux with various preemption patches, for RTAI, and for Xenomai. In general the trends are as expected. It is noteworthy, however, that CONFIG_PREEMPT increases the average latency not just of stock Linux results, but also of Xenomai results; the authors discuss some possible causes. The measurements are taken over periods of 1 minute each, which the authors note is a brief enough period to put measured maximum values into question.

7 Conclusion

We have presented a test system for evaluating the performance of two real-time tasks on Linux and Xenomai systems. The most important feature of this suite is that it uses an external system, running code directly on "bare metal", to perform all

measurements. This avoids the inherent untrustworthiness of self-reported performance measurements.

We ran the suite on a specific hardware platform, the BeagleBoard, using three different kernel configurations: stock Linux, Real-time (PREEMPT_RT) Linux, and Xenomai. We presented and analyzed data from these specific tests. We also presented our general conclusions about when each kernel configuration might be most appropriate.

7.1 Acknowledgments

We want to thank several people. Gilles Chanteperdrix fielded our questions on the xenomai-help mailing list to help us get Xenomai working on our hardware. Thomas Gleixner and Gowrishankar similarly helped us with PREEMPT_RT via the linux-rt-users mailing list. Finally, Carsten Emde offered several valuable comments on an earlier draft of this paper. Many thanks to you all.

References

[1] Adeos home page. `http://home.gna.org/adeos/`.

[2] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. *Nuclear Science, IEEE Transactions on*, 55(1):435–439, 2008.

[3] BeagleBoard Ubuntu Lucid Linux Demo Image. `http://rcn-ee.net/deb/rootfs/ubuntu-10.04-minimal-armel.tar.7z`.

[4] Linux 2.6 kernel with BeagleBoard patches. `https://code.launchpad.net/~beagleboard-kernel/+junk/2.6-stable`.

[5] Markus Franke. A quantitative comparison of realtime linux solutions. Technical report, Chemnitz University of Technology, March 5 2007.

[6] Xenomai git source code repository. `git://xenomai.org/xenomai-2.5.git`.

[7] Xenomai: Real-time framework for linux. `http://www.xenomai.org`.

[8] Philippe Gerum. Xenomai - Implementing a RTOS emulation framework on GNU/Linux, April 2004. `http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf`.

[9] A. Luchetta, A. Barbalace, G. Manduchi, A. Soppelsa, and C. Taliercio. Real-time communication for distributed plasma control systems. *Fusion Engineering and Design*, 83(2-3):520 – 524, 2008. Proceedings of the 6th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research.

[10] CONFIG_RT_PREEMPT patch. `http://www.kernel.org/pub/linux/kernel/projects/rt/patch-2.6.33.7-rt29.gz`.

[11] Real-time linux frequently asked questions. `https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions`.

[12] The real-time linux wiki. `https://rt.wiki.kernel.org/index.php/Main_Page`.

[13] M. Sarker, Chang Hwan Kim, Jeong-San Cho, and Bum-Jae You. *Development of a Network-based Real-Time Robot Control System over IEEE 1394: Using Open Source Software Platform*, pages 563–568. IEEE, 2006.

[14] N. Vun, H. F. Hor, and J. W. Chao. Real-time enhancements for embedded linux. In *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 737–740, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Thomas Wiedemann. How fast can computers react? Technical report, Chemnitz University of Technology, December 21 2005.

[16] Karim Yaghmour. Adaptive Domain Environment for Operating Systems. `http://www.opersys.com/ftp/pub/Adeos/adeos.pdf`.