‘Real Time’ vs. ‘Real Fast’: How to Choose?

Paul E McKenney
IBM Linux Technology Center
`paulmck@linux.vnet.ibm.com`

Abstract

Although the oft-used aphorism "real-time is not real-fast" makes a nice sound bite, it does not provide much guidance to developers. This paper will provide the background needed to make a considered design choice between "real time" (getting started as quickly as possible) and "real fast" (getting done quickly once started). In many ways, "real fast" and "real time" are Aesop’s tortoise and hare, respectively. But in the real world of real time, sometimes the race goes to the tortoise and sometimes it goes to the hare, depending on the requirements as well as the details of the workload and the enclosing software environment.

1 Introduction

Linux™ has made much progress in the real-time arena over the past ten years, particularly with the advent of the -rt patchset [10], a significant fraction of which has now reached mainline. This naturally leads to the question of which workloads gain improved performance by running on real-time Linux. To help answer this question, we take a close look at the real-time vs. real-fast distinction in order to produce useful criteria for choosing between a real-time and non-real-time Linux.

Section 2 looks at a pair of example applications in order to make a clear distinction between real-time and real-fast, Section 3 examines some factors governing the choice between real-time and real-fast, and Section 4 gives an overview of the underlying causes of real-time Linux’s additional overhead. Section 5 lays out some simple criteria to help choose between real fast and real time, and finally, Section 6 presents concluding remarks.

2 Example Applications

This section considers a pair of diverse workloads, an embedded fuel-injection application and a Linux kernel build.

2.1 Fuel Injection

This rather fanciful fuel-injection scenario evaluates real-time Linux for controlling fuel injection for a mid-sized industrial engine with a maximum rotation rate of 1500 RPM. This is slower than an automotive engine: when all else is equal, larger mechanical artifacts move more slowly than do smaller ones. We will be ignoring complicating factors such as computing how much fuel is to be injected.

If we are required to inject the fuel within one degree of top dead center (the point in the combustion cycle where the piston is at the very top of the cylinder), what jitter can be tolerated in the injection timing? 1500 RPM is 25 RPS, which in turn is 9000 degrees per second. Therefore, a tolerance of one degree turns into a tolerance of one nine-thousandth of a second, or about 111 microseconds.

Such an engine would likely have a rotational position sensor that might generate an interrupt to a device driver, which might in turn awaken a real-time control process. This process could then calculate the time until top dead center for each cylinder, and then execute a sequence of `nanosleep()` system calls to control the timing. The code to actuate the fuel injector might be a short sequence of MMIO operations.

This is a classic real-time scenario: we need to do something before a deadline, and faster is most definitely *not* better: injecting fuel too early is just as bad as injecting it too late. This situation calls for some benchmarking and validation of the `nanosleep()` system call, for example, with the code shown in Figure 1. On each pass through the loop, lines 2-5 record the start time, lines 6-9 execute the `nanosleep()` system call with the specified sleep duration, lines 10-13 record the end time, and lines 14-16 compute the jitter in microseconds and print it out. This jitter is negative if the `nanosleep()` call did not sleep long enough, and positive if it slept too long.

```
for (i = 0; i < iter; i++) {
  if (clock_gettime(CLOCK_MONOTONIC, &timestart) != 0) {
    perror("clock_gettime 1");
    exit(-1);
  }
  if (nanosleep(&timewait, NULL) != 0) {
    perror("nanosleep");
    exit(-1);
  }
  if (clock_gettime(CLOCK_MONOTONIC, &timeend) != 0) {
    perror("clock_gettime 2");
    exit(-1);
  }
  delta = (double)(timeend.tv_sec - timestart.tv_sec) * 1000000 +
    (double)(timeend.tv_nsec - timestart.tv_nsec) / 1000.;
  printf("iter %d delta %g\n", iter, delta - duration);
}
```

Figure 1: Loop to Validate nanosleep()

It is important to use `clock_gettime()` with the `CLOCK_MONOTONIC` argument. The more-intuitive `CLOCK_REALTIME` argument to `clock_gettime()` means "real" as in real-world wall-clock time, *not* as in real-time. System administrators and NTP can adjust real-world wall-clock time. If you incorrectly use `gettimeofday()` or `CLOCK_REALTIME` and the systems administrator sets the time back one minute, your program might fail to actuate the fuel injectors for a full minute, which will cause the engine to stop. You have been warned!

Of course, before executing this validation code, it is first necessary to set a real-time scheduling priority, as shown in Figure 2. Line 2-5 invokes `sched_get_priority_max()` to obtain the highest possible real-time (`SCHED_FIFO`) priority (or print an error) and lines 6-9 set the current process's priority. Of course, you must have appropriate privileges to switch to a real-time priority: either super-user or `CAP_SYS_NICE`. There is also a `sched_get_priority_min()` that gives the lowest priority for a given scheduler policy, so that `sched_get_priority_min(SCHED_FIFO)` returns the lowest real-time priority, allowing applications to allocate multiple priority levels in an implementation-independent manner, if desired.

However, real-time priority is not sufficient to obtain real-time behavior, because the program might still take page faults. The fix is to lock all of the pages into memory, as shown in Figure 3. The `mlockall()` system call will lock all of the process's current memory down (`MCL_CURRENT`), and all future mappings as well (`MCL_FUTURE`).

Of course, hardware irq handlers will preempt this code. However, the -rt Linux kernel has threaded irq handlers, which appear in the `ps` listing with names resembling "IRQ-16". You can check their priority using the `sched_getscheduler()` system call, or by looking at the second-to-last field in `/proc/<PID>/stat`, where "`<PID>`" is replaced by the actual process ID of the irq thread of interest. It is possible to run your real-time application at a higher priority than that of the threaded irq handlers, but be warned that an infinite loop in such an application can lock out your irqs, which can cause your system to hang.

If you are running on a multi-core system, another way to get rid of hardware-irq latencies is to direct them to a specific CPU (also known as "hardware thread"). You can do this using `/proc/irq/<IRQ>/smp_affinity`, where "`<IRQ>`" is replaced by the irq number. You can then affinity your real-time program to some other CPU, thereby insulating your program from interrupt latency. It may be necessary to pin various kernel daemons to subsets of the CPUs as well, and the schedutils `taskset` command may be used for this purpose (though care is required, as some of the per-CPU kernel daemons really do need to run on the cor-

```
sp.sched_priority = sched_get_priority_max(SCHED_FIFO);
if (sp.sched_priority == -1) {
  perror("sched_get_priority_max");
  exit(-1);
}
if (sched_setscheduler(0, SCHED_FIFO, &sp) != 0) {
  perror("sched_setscheduler");
  exit(-1);
}
```

Figure 2: Setting Real-Time Priority

```
if (mlockall(MCL_CURRENT | MCL_FUTURE) != 0) {
  perror("mlockall");
  exit(-1);
}
```

Figure 3: Preventing Page Faults

responding CPU). Of course, this has the downside of prohibiting your real-time program from using all of the CPUs, thereby limiting its performance. This technique is nonetheless useful in some cases.

Once we have shut down these sources of non-real-time behavior, we can run the program on both a real-time and a non-real-time Linux system. In both cases, we run on a four-CPU 2.2GHz x86 system running with low-latency firmware.

Even after taking all of these precautions, the non-real-time Linux fails miserably, missing the mark by up to 3 *milli*seconds. Non-real-time Linux systems are therefore completely inappropriate for this fuel-injection application.

As one might hope, real-time Linux does much better. Nanosleep always gets within 20 *micro*seconds of the requested value, and 99.999% of the time within 13 microseconds in a run of 10,000,000 trials. Please note that the results in this paper are from a lightly tuned system: more careful configuration (for example, using dedicated CPUs) might well produce better results.

If real-time Linux can so easily meet such an aggressive real-time response goal, it should do extremely well for more typical workloads, right? This question is taken up in the next section.

```
tar -xjf linux-2.6.24.tar.bz2
cd linux-2.6.24
make allyesconfig > /dev/null
time make -j8 > Make.out 2>&1
cd ..
rm -rf linux-2.6.24
```

Figure 4: Kernel Build Script

2.2 Kernel Build

Since the canonical kernel-hacking workload is a kernel build, this section runs a kernel build on both a real-time and a non-real-time Linux. The script used for this purpose is shown in Figure 4, featuring an 8-way parallel build of the 2.6.24 Linux kernel given an `allyesconfig` kernel configuration. The results (in decimal seconds) are shown on Table 1, and as you can see, real-time Linux is not helping this workload. The non-real-time Linux not only completed the build on average more than 15% faster than did the real-time Linux, but did so using less than half of the kernel-mode CPU time. Although there is much work in progress to narrow this gap, some of which will likely be complete before this paper is published, there is no getting around the fact that this is a large gap.

Clearly, there are jobs for which real-time Linux is not the right tool!

		Real Fast (s)	Real Time (s)
real	Raw Data	1350.4	1524.6
		1332.7	1574.2
		1314.5	1569.8
	Average	**1332.6**	**1556.2**
	Std. Dev.	14.6	22.4
user	Raw Data	3027.2	2940.9
		3013.1	2982.2
		2996.1	2971.2
	Average	**3012.2**	**2964.7**
	Std. Dev.	12.7	17.5
sys	Raw Data	314.7	644.3
		317.3	660.9
		317.9	665.9
	Average	**316.6**	**657.0**
	Std. Dev.	1.4	9.2

Table 1: Kernel Build Timings

2.3 Discussion

A key difference between these two applications is the duration of the computation: fuel injection takes place in microseconds, while kernel builds take many seconds or minutes. In the fuel-injection scenario, we are therefore willing to sacrifice considerable performance in order to meet microsecond-scale deadlines. In contrast, even on a very fast and heavily tuned machine, handfuls of milliseconds are simply irrelevant on the kernel-build timescale.

The next section will look more closely at these issues.

3 Factors Governing Real Time and Real Fast

In the previous section, we saw that the duration of the work is a critical factor: although there are a few exceptions, real-time response is usually only useful when performing very short units of work in response to a given real-time event. If the work unit is going to take three weeks to complete, then starting the work a few milliseconds late is unlikely to matter much. This relationship is displayed in Figure 5 for work-unit durations varying from one microsecond on the far left to 100 millisecond on the far right, where smaller latencies are better. The y-axis shows the total delay, including the scheduling latency and the time required to perform the unit of work. If the unit of work to be done is quite

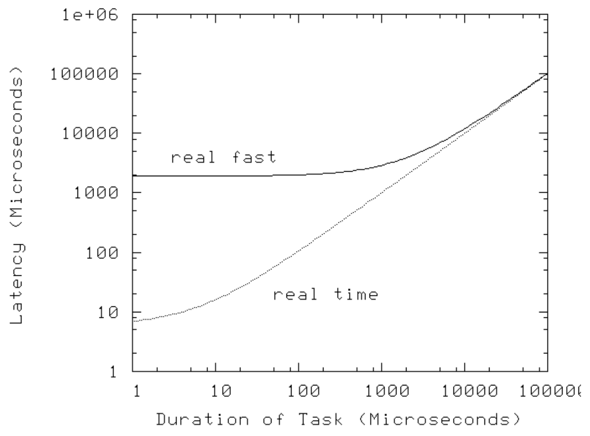

Figure 5: Real Time vs. Real Fast Against Work-Unit Duration for User-Mode Computation

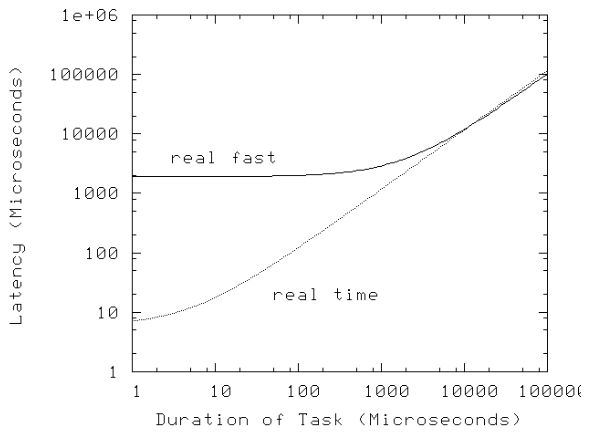

Figure 6: Real Time vs. Real Fast Against Work-Unit Duration for Kernel Build

small, a real-time system will out-perform a non-real-time system by orders of magnitude. However, when the duration of the unit of work exceeds a few tens of milliseconds, there is no discernable difference between the two.

Furthermore, Figure 5 favors the real-time system because it assumes that the real-time system processes the unit of work at the same rate as does the non-real-time system. However, in the kernel-build scenario discussed in Section 2.2, the non-real-time Linux built the kernel 16.78% faster than did the real-time Linux. If we factor in this real-time slowdown, the non-real-time kernel offers slightly *better* overall latency than does the real-time kernel for units of work requiring more than about ten milliseconds of processing, as shown in Figure 6. Of course, this breakeven would vary depending on the type of work. For example, floating-point processing speed

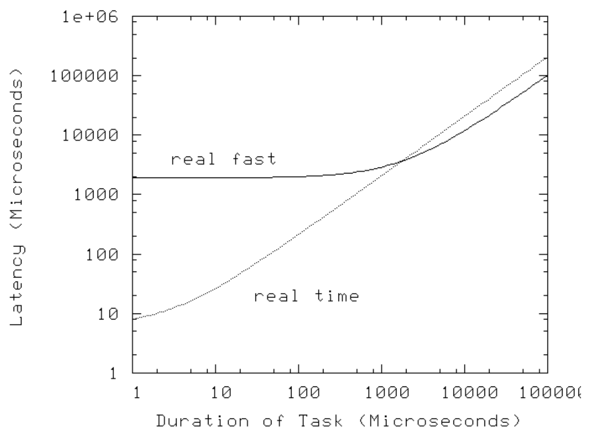

Figure 7: Real Time vs. Real Fast Against Work-Unit Duration for Heavy I/O

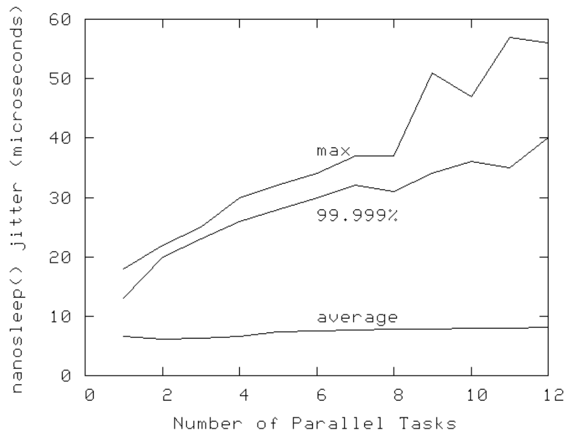

Figure 8: Nanosleep Jitter With Increased Load

would be largely independent of the type of kernel (and hence represented accurately by Figure 5), while heavy I/O workloads would likely be profoundly affected by the kernel type, as shown in Figure 7, which uses the 2-to-1 increase in kernel-build system time as an estimate of the slowdown. In this case, the crossover occurs at about one millisecond.

In addition, of course, a concern with worst-case behavior should steer one towards real time, while a concern with throughput or efficiency should steer one towards real fast. In short, use real-time systems when the work to be done is both time-critical and of short duration. There are exceptions to this rule, but they are rare.

CPU utilization is another critical factor. To show this, we run a number of the `nanosleep()` test programs in parallel, with each program running 100,000 calls to `nanosleep` in a loop (code shown in Figure 1). Figure 8 shows the resulting average, 99.999 percentile delay, and maximum delay. The average jitter changes very little as we add tasks, which indicates that we are getting good scalability from a real-fast viewpoint. The 99.999 percentile and maximum delays tell a different story, as both increase by more than a factor of three as we go from a single task to 12 parallel tasks.

This is a key point: obtaining the best possible real-time response usually requires that the real-time system be run at low utilization. This is of course in direct conflict with the desire to conserve energy and reduce system footprint. In some cases, it is possible to get around this conflict by putting both real-time and non-realtime workload on the same system, but some care is still required. To illustrate this, run four parallel downloads of a kernel source tree onto the system, then unpack one of them and do a kernel build. When the `nanosleep` test program runs at maximum priority concurrently with this kernel-build workload, we see the 99.999% jitter at 59 microseconds with the worst case at 146 microseconds, which is worse than the parallel runs—but still much better than the multi-millisecond jitters from the non-real-time kernel.

Advancing technology can be expected to improve real-time Linux's ability to maintain real-time latencies in face of increasing CPU utilization, and careful choice of drivers and hardware might further improve the situation. Also, more-aggressive tuning might well produce better results. For example, this workload does not control the periodicity of the `nanosleep()` test programs, so that all 12 instances might well try to run simultaneously on a system that has but four CPUs. In real-world systems, mechanical constraints often limit the number of events that can occur simultaneously, in particular, engines are configured so that it is impossible for all cylinders to fire simultaneously. That said, sites requiring the best possible utilization will often need to sacrifice some real-time response.

Similarly, if you need to use virtualization to run multiple operating-system instances on a single server, you most likely need real fast as opposed to real time. Again, technology is advancing quite quickly in this area, especially in the embedded space, so we may soon see production-quality virtualization environments that can simultaneously support both real-time and real-fast operating systems. This is especially likely to work well if either: (1) CPUs and memory can be dedicated to a given operating instance or (2) the hypervisor (e.g., Linux with KVM) gives real-time response, but the guest operating systems need not do so. Longer term, it is quite possible that both the hypervisor and the guest OSes will offer real-time response.

4 Sources of Real-Time Overhead

The `nanosleep()` test program used the `mlockall()` system call to pin down memory in order to avoid page-fault latencies. This is great for this test program's latency, but has the side-effect of removing a chunk of memory from the VM system's control, which limits the system's ability to optimize memory usage. This can degrade throughput for some workloads.

Real-time Linux's more-aggressive preemption increases the overhead of locking and interrupts [2]. The reason for the increased locking overhead is that the corresponding critical sections may be preempted. Now, suppose that a given lock's critical section is preempted, and that each CPU subsequently attempts to acquire the lock. Non-real-time spinlocks would deadlock at this point: the CPUs would each spin until they acquired the lock, but the lock could not be released until the lock holder got a chance to run. Therefore, spinlock-acquisition primitives must block if they cannot immediately acquire the lock, resulting in increased overhead. The need to avoid priority inversion further increases locking overhead. This overhead results in particularly severe performance degradation for some disk-I/O benchmarks, however, real-time adaptive spinlocks may provide substantial improvements [4]. In addition, the performance of the user-level `pthread_mutex_lock()` primitives may be helped by private futexes [5].

Threaded interrupts permit long-running interrupt handlers to be preempted by high-priority real-time processes, greatly improving these processes' real-time latency. However, this adds a pair of context switches to each interrupt even in absence of preemption, one to awaken the handler thread and another when it goes back to sleep, and furthermore increases interrupt latency. Devices with very short interrupt handlers can specify `IRQF_NODELAY` in the `flags` field of their `struct irqaction` to retain the old hardirq behavior, but this is not acceptable for handlers that run for more than a small handful of microseconds.

Linux's $O(1)$ scheduler is extremely efficient on SMP systems, as a given CPU need only look at its own queue. This locality reduces cache thrashing, yielding extremely good performance and scalability, aside from infrequent load-balancing operations. However, real-time systems often impose the constraint that the N highest-priority runnable tasks be running at any given point in time, where N is the number of online CPUs. This constraint cannot be met without global scheduling, which re-introduces cache thrashing and lock contention, degrading performance, especially on workloads with large numbers of runnable real-time tasks. In the future, real-time Linux is likely to partition large SMP systems, so that this expensive global scheduling constraint will apply only within each partition rather than across the entire system.

Real-time Linux requires high-resolution timers with tens-of-microseconds accuracy and precision, resulting in higher-overhead timer management [3, 6]. However, these high-resolution timers are implemented on a per-CPU basis, so that it is unlikely that this overhead will be visible at the system level for most workloads. In addition, real-time Linux distinguishes between real-time "timers" and non-real-time "timeouts", and only the real-time timers use new and more-expensive high-resolution-timer infrastructure. Timeouts, for example, TCP/IP retransmission timeouts, continue to use the original high-efficiency timer-wheel implementation, further reducing the likelihood of problematic timer overheads.

Real-time Linux uses preemptible RCU, which has slightly higher read-side overhead than does Classic RCU [8]. However, the read-side difference is unlikely to be visible at the system level for most workloads. In contrast, preemptible RCU's update-side "grace-period" latency is significantly higher than that of RCU classic [7]. If this becomes a problem, it should be possible to expedite RCU grace period, albeit incurring additional overhead. It may then be possible to retire the Classic RCU implementation [9], but given that Classic RCU's read-side overhead is exactly zero, careful analysis will be required before such retirement can be appropriate.

In summary, the major contributors to the higher overhead of real-time Linux include increased overhead of locking, threaded interrupts, real-time task scheduling, and increased RCU grace-period latency. The next section gives some simple rules that help choose between the real fast non-real-time Linux kernel and the real-time Linux kernel.

5 How to Choose

The choice of real time vs. real fast is eased by considering the following principles:

1. Consider whether the goal is to get a lot of work done (real fast throughput), or to get a little bit of work done in a predictable and deterministic time-frame (real-time latency).

2. Consider whether the hardware and software can accommodate the heaviest possible peak load without missing deadlines (real time), or whether occasional peak loads will degrade response times (real fast). It is common real-time practice to reserve some fraction of resources, for example, to limit CPU utilization to 50%.

3. Consider memory utilization: if your workload oversubscribes memory, so that page faults will occur, you cannot expect real-time response.

4. If you use virtualization, you are unlikely to get real-time response—though this may be changing.

5. Consider the workload: a process that executes normal instructions in user mode will incur a smaller real-time average-overhead penalty than will a process that makes heavy use of kernel services.

6. Focus on work-item completion time instead of on start time: the longer the work item's execution time, the less helpful real-time Linux will be.

The need to focus on deterministic work-item completion cannot be stressed enough. Common practice in the real-time arena is to focus on when the work-item starts, in other words, on scheduling latency. This is understandable, given the historic separation of the real-time community into RTOS and real-time application developers, both working on proprietary products. It is hoped that the advent of open-source real-time operating systems will make it easier for developers to take the more global viewpoint, focusing on the time required for the application to both start *and* finish its work. Please note that it is important to focus on the proper level of detail, for example, event-driven systems should analyze deadlines on a per-event basis.

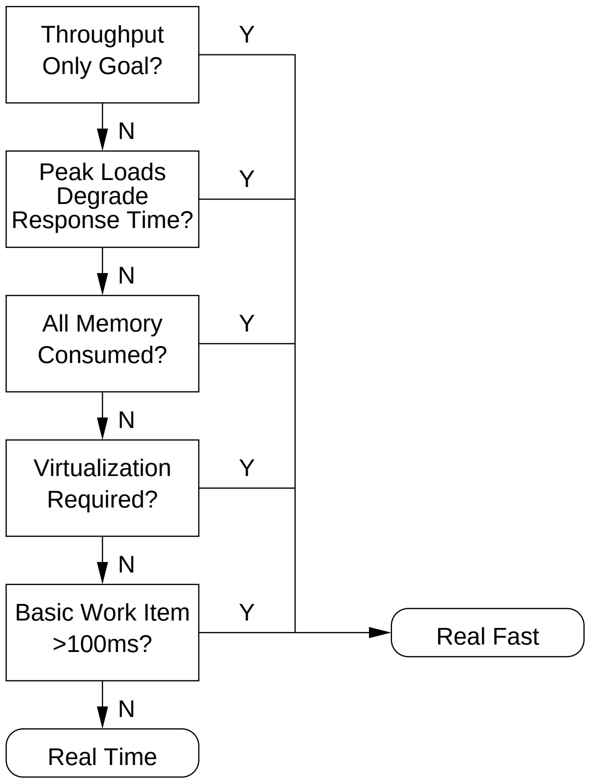

Figure 9: Real Time vs. Real Fast Decision Flow

A rough rule-of-thumb decision flow is shown in Figure 9. If you only care about throughput—the amount of work completed per unit time—then you want real fast. If cost, efficiency, or environmental concerns force you to run at high CPU utilization so that peak loads degrade response times, then you again want real fast—and as a rough rule of thumb, the more aggressive your real-time workload, the lower your CPU utilization must be. One exception to this occurs in some scientific barrier-based computations, where real-time Linux can reduce OS jitter, allowing the barrier computations to complete more quickly—and in this case, because floating point runs at full speed on real-time Linux, this is one of those rare cases where you get *both* real fast *and* real time simultaneously. If your workload will fill all of memory, then the `mlockall()` system call becomes infeasible, forcing you to either purchase more memory or allow the resulting page faults force you to go with real fast. Given the current state of the art, if you need virtualization, you are most likely in real-fast territory—though this may soon be changing, especially for carefully configured systems. Finally, if each basic item of work takes hundreds of milliseconds, any scheduling-latency benefit from real-time Linux is likely to be lost in the noise.

If you reach the real-time bubble in Figure 9, you may need some benchmarking to see which of real time or real fast works best for your workload. Of course, no benchmarking is needed to see that a workload requiring (say) 100 microseconds of processing with a 250-microsecond deadline will require real-time Linux, and there appears to be no shortage of applications of this type. In fact, it appears that real-time processing is becoming more mainstream. This is due to the fact that the availability of real-time Linux has made it easier to integrate real-time systems into enterprise workloads [1], which are starting to require increasing amounts of real-time behavior. Where traditional real-time systems were stand-alone systems, modern workloads increasingly require that the real-time systems be wired into the larger enterprise.

6 Concluding Remarks

If you remember only one thing from this paper, let it be this: "use the right tool for the job!!!"

Of course, ongoing work to reduce the overhead of real-time Linux will hopefully reduce the performance penalty imposed by the real-time kernel, which will in turn make real-time Linux the right tool for a greater variety of workloads.

Might real-time Linux's performance penalty eventually be reduced to the point that real-time Linux is used for all workloads? This outcome might seem quite impossible. On the other hand, any number of impossible things have come to pass in my lifetime, including space flight, computers beating humans at chess, my grandparents using computers on a daily basis, and a single operating-system-kernel source base scaling from cell phones to supercomputers. I have since learned to be exceedingly careful about labeling things "impossible".

Impossible or not, here are some challenging but reasonable intermediate steps for the Linux kernel, some of which are already in progress:

1. Reduce the real-time performance penalty for multiple communications streams.
2. Reduce the real-time performance penalty for mass-storage I/O. (This becomes more urgent with the advent of solid-state disks.)
3. Reduce the preemptable RCU grace-period latency penalty.
4. Where feasible, adjust implementation so that performance penalties are incurred only when there are actually real-time tasks in the system.

It will also likely be possible to further optimize some of the real-time implementations. In any case, real-time Linux promises to remain an exciting and challenging area for some time to come.

Acknowledgements

No article mentioning the -rt patchset would be complete without a note of thanks to Ingo Molnar, Thomas Gleixner, Sven Deitrich, K. R. Foley, Gene Heskett, Bill Huey, Esben Neilsen, Nick Piggin, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese. We all owe Ankita Garg, Vara Prasad, Ananth Mavinakayanahalli, Chirag Jog, and especially Darren Hart and Paul Giangarra a debt of gratitude for their help in making this paper human-readable. I am grateful to Paul Clarke for the use of his machines, and the ABAT team for providing easy access to these machines. Finally, I owe thanks to Daniel Frye and Kathy Bennett for their support of this effort.

Legal Statement

References

[1] Berry, R. F., McKenney, P. E., and Parr, F. N. Responsive systems: An introduction. *IBM Systems Journal 47*, 2 (April 2008), 197–206.

[2] Corbet, J. Approaches to realtime Linux. Available: `http://lwn.net/Articles/106010/` [Viewed March 25, 2008], October 2004.

[3] CORBET, J. A new approach to kernel timers. Available: http://lwn.net/Articles/152436/ [Viewed April 14, 2008], September 2005.

[4] CORBET, J. Realtime adaptive locks. Available: http://lwn.net/Articles/271817/ [Viewed April 14, 2008], March 2008.

[5] DUMAZET, E. [PATCH] FUTEX : new PRIVATE futexes. Available: http://lkml.org/lkml/2007/4/5/236 [Viewed April 18, 2008], April 2007.

[6] GLEIXNER, T., AND MOLNAR, I. [announce] ktimers subsystem. Available: http://lwn.net/Articles/152363/ [Viewed April 14, 2008], September 2005.

[7] GUNIGUNTALA, D., MCKENNEY, P. E., TRIPLETT, J., AND WALPOLE, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal 47*, 2 (May 2008), 221–236. Available: http://www.research.ibm.com/journal/sj/472/guniguntala.pdf [Viewed April 24, 2008].

[8] MCKENNEY, P. E. The design of preemptible read-copy-update. Available: http://lwn.net/Articles/253651/ [Viewed October 25, 2007], October 2007.

[9] MCKENNEY, P. E., SARMA, D., MOLNAR, I., AND BHATTACHARYA, S. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium* (July 2006), pp. v2 123–138. Available: http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184 http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf [Viewed January 1, 2007].

[10] MOLNAR, I. Index of /mingo/realtime-preempt. Available: http://www.kernel.org/pub/linux/kernel/projects/rt/ [Viewed February 15, 2005], February 2005.