

Real Time with a Red Hat Analysis of Standard Linux for Real-Time Simulation

**Robert E. Murray
The Boeing Company
St. Louis, Missouri
bob.murray@boeing.com**

ABSTRACT

There is no doubt that Linux is a full-featured operating system. Its Unix heritage and tuning for use in large server farms has given it traits that any designer of large-scale systems craves: robust, near-crash-proof operation, high performance, and a native software development environment familiar to any Unix developer.

There is also little doubt that standard Linux distributions, such as those from Red Hat and Debian, are unsuitable for hard real-time, deeply embedded applications. Embedded and Real-Time Linux extensions fill that niche but at the expense of many great features of standard Linux such as user/kernel protection, memory protection, a complete set of system services, and full device driver availability.

However, most man-in-the-loop simulation systems do not have the toughest requirements of hard real-time embedded systems. There is no need for 1000 Hz sample rates, 20 microsecond interrupt response, or deeply embedded operation. Can standard Linux possibly meet the hard real-time requirements of simulation without losing any of the many Linux advantages?

The answer is a resounding “Yes”. This paper details the Research and Development effort that resulted in the use of an ordinary Red Hat Linux distribution as the operating system for real-time training simulators.

Real-time determinism is the key. Tests were devised to subject various system configurations to a mix of computation and I/O that is typical in a flight simulator. Determinism test results are presented for these types of system variations: single vs. dual processor, diskless vs. disked system, single process vs. multi-process, CPU affinity, the 2.4 vs. the developmental 2.5 Linux kernel, and the advantages of the new Hyperthreading feature of Intel processors.

The results show that standard Linux on high-end PC hardware can be configured as an exceptionally low-cost simulation computational system to provide training to the warfighter with uncompromised fidelity.

ABOUT THE AUTHOR

Mr. Robert Murray is an Associate Technical Fellow in the Boeing Company with 19 years of experience in flight simulation technology. He is currently the Principal Investigator for the Computational Architecture IR&D program in Boeing Training and Support Systems, conducting research on computer and network technology for training simulators. His past assignments have led to extensive hardware and software experience in applying commercial computer and network products to meet computational, I/O, and distributed real-time simulation requirements. Mr. Murray received an M.S. degree in Electrical Engineering from Washington University, St. Louis in 1993 and a B.S. in Electrical and Computer Engineering from the University of Cincinnati in 1983.

Real Time with a Red Hat Analysis of Standard Linux for Real-Time Simulation

Robert E. Murray
The Boeing Company
St. Louis, Missouri
bob.murray@boeing.com

INTRODUCTION

The emergence of Linux as a serious contender in the operating system marketplace has been both an expected event and a surprise. Its bullet-proof operation, familiarity and ease of use, and outright performance, especially in the server applications, are the natural progression of the success of Open Source Software started almost 20 years ago by the Free Software Foundation and the GNU software tools. Yet, it is surprising at how fast and how well Linux has matured given its peculiar revenue model.

No matter how it got to where it is, its advantages are too numerous to ignore. The important issue for developers of real-time simulations is how best to leverage this groundswell of enthusiasm and plethora of tools and distributions that are free for the taking. This paper will summarize the research that led to the successful use of the completely free and very popular Red Hat Linux distribution in actual real-time training system host computers.

The use of Linux in simulation has been common, with reports of Linux ports dating back to the 1998-1999 time frame. There are several excellent summaries of the issues encountered in porting simulations to Linux. However, little has been said about running hard real-time simulations on standard Linux. Most report on either standard Linux for a CGF (Computer Generated Forces) type of simulation (Burch, 2000) or the use of a Real-Time Linux extension for hard real-time simulation (Nalepka, 2001). Nalepka (2001) also describes a preference to use a standard Linux distribution but found that it required specialized timer hardware and a kernel patch to get adequate determinism from the kernel used at that time. Kern (2001), detailing the porting of a training system to Red Hat Linux, mentions real-time performance but considers it something to be tested at a later date.

This paper shows rigorous testing with quantitative results of a standard Linux distribution in a real-time frame-driven simulation environment that meets the

requirements of high-fidelity flight simulation. This maintains all the advantages of a fully featured Linux installation and avoids the limitations of real-time and embedded variants of Linux, as will be explained further.

The focus of this study is on the use of Linux in the simulation computational system, also called the host computer. The host computer is made up of one or more processors with some form of high-speed communication between them and I/O to the other components of the simulator device such as image generators, crew station I/O, and the Instructor Operator Station.

Graphics are also a large part of training simulation. Fenrich (1999) published one example of the use of a Linux PC as a graphics workstation. Since its publication, much has been improved in Linux distributions, kernels, and device drivers for graphics cards that eliminate most of the problems cited in that paper. Recent experience at Boeing has shown excellent performance and ease of integration for generating cockpit displays in real-time simulations with PCs running standard Linux. However, graphics performance is an issue that is beyond the scope of this study.

Why Linux is an Excellent Solution for Training Simulation

The single biggest advantage of Linux and open source software is exactly that: the source code is open and readily available, greatly easing system design and integration. Linux is easy to configure for diskless booting. This is very important to reduce the number of disk drives that need to be managed in complex systems requiring many computers. It is also desirable for classified environments where all disks have to be regularly removed and locked up. Linux can also run headless, that is, with no monitor or keyboard. A single rack can hold several dozen computers without the need for a monitor attached to each or a KVM (keyboard, video, mouse) switch and all the associated

cabling. Remote login capability allows much easier developing and debugging of a large number of computers simultaneously.

Not to be ignored are the many intangible benefits of working squarely in the mainstream of operating system and software development rather than as a niche customer in the embedded computing market. The limitation of embedded computing is the reason real-time Linux variants as well as many commercial real-time operating systems lose their appeal.

While real-time versions of Linux have been used in training systems, they have some disadvantages. Proctor (2002) gives a comprehensive overview of the various methods used to extend or alter Linux for embedded and real-time use. The best performing Linux extensions work by adding a “microkernel” that runs under the standard Linux kernel and manages interrupts and all real-time tasks. Real-time tasks must run at kernel level, not at the normal user level. This gives absolute control and excellent determinism but memory and user/kernel protection are lost and the system services and network protocol stack are limited. Also, developers are unable to use standard debuggers and other development tools while running at kernel level, and support for Ada and Fortran is unknown. In short, this would be a very risky environment for a large multiprocessor application involving several hundred thousand lines of code, possibly mixing Ada, C, C++, and Fortran with a development team of dozens of programmers.

The other method of improving real-time performance to Linux is to alter (“patch”) the Linux kernel directly. Its open source nature makes it readily available for experimenting. Several real-time patches have been created and are available for download. However, this is not a safe approach for a training system that must be maintained for many years. Each new kernel upgrade must be similarly patched, but patches that work on one kernel version are rarely kept up to date for later versions.

There are commercial companies that provide kernel extensions in a clean, documented, and maintained Linux distribution without resorting to the microkernel approach. Since the kernel is controlled by the GNU Public License, these extensions cannot remain proprietary. The best of these extensions are being added to the standard 2.5 Linux kernel now in development. Better yet, some of these improvements have already been retrofitted into the stable 2.4 kernel series and have already been released in standard Linux distributions. So again, the open source model

benefits everyone by allowing these extensions and improvements to be available with high quality and in a timely manner.

One example of a recent kernel improvement is CPU affinity. This is the means to lock a process to a particular CPU in a multi-processor system. It eliminates timing glitches that can occur because of the overhead imposed by the operating system scheduler and memory cache system when processes migrate between processors. Love (July 2003) gives details on CPU affinity and how it is made available in Red Hat 9.

Extensive rework and improvements have been made to the 2.5 kernel to improve responsiveness for a wide range of applications such as multimedia. The kernel is considered pre-emptible, which means it can be interrupted to run higher priority tasks much more readily. The critical regions of kernel code that cannot be pre-empted are much shorter. In all, it is a better starting platform for real-time use. This is especially true for the more complicated combinations of real-time and non-real-time tasks. The 2.5 kernel is planned to be released as the stable 2.6 Linux kernel by the end of 2003. Love (May 2003) gives an excellent overview of the improvements in the 2.6 kernel.

Test results in this paper are given for both the 2.4.20 kernel delivered in Red Hat 9 and the 2.5.69 developmental kernel. It will be shown that Red Hat 9 with no kernel patching and only minimal change from the standard configuration is suitable for a wide range of frame-driven real-time simulations. Results from the 2.5 kernel tests indicate that the 2.6 kernel promises to be even better.

The *Real* Real-time Requirements for Training Simulators

Man-in-the-loop simulators are generally considered to require embedded real-time host computers. However, under closer scrutiny, they are not really very embedded. They do not require a small footprint in memory or physical space. A minute or two of boot-up time is acceptable. Unlike many embedded systems, the computer does not add significantly to the cost of the overall simulator device. The host computer has to be generally invisible to the end users (the students and instructors) but does not have to be completely free of normal administration by technicians and engineers. This is not a new development. Windows and Unix systems have been used successfully for years as simulation host computers. Standard Linux is no different in that way.

Then there is the hard real-time requirement. The same close scrutiny reveals that on the continuum of hard real-time requirements, training systems fall somewhere in the middle. There is generally no interrupt latency requirement that is common in some hard real-time systems. Even if there is a video sync interrupt used to drive the frame rate, system response in the low 100s of microseconds can easily be tolerated; the low 10s of microseconds that is often given as a requirement for hard real-time systems is not necessary. Frame rates up to 100 Hz are tolerant of jitter on the order of hundreds of microseconds, unlike rates of 1000 Hz or more used in motion control systems, for example. Simulation software can often be designed to handle an occasional overframe event, possibly up to a millisecond or two, with no noticeable artifact as long as it is a singular event and does not occur in several back-to-back simulation frames. However, not all programs allow overframing so a Linux solution has to be configured and tested to ensure worst-case jitter is below frame limits with no exceptions under all conditions.

Requirements for frame rates up to 100 Hz and 200 microseconds of jitter leads to a different philosophy of simulation host computer system design, that of a real-time server. Instead of trying to scale up embedded real-time operating systems to the needs of large, complex training devices, it is easier to start with a robust server-class operating system that was designed from the start for large-scale environments and make it run with real-time determinism. With this philosophy in mind, it becomes feasible to consider using a standard Linux distribution as the basis for a simulation computer system. It is just a matter of testing the possible configurations to see which are suitable for real-time simulations. That testing is described next.

TEST METRICS AND METHODOLOGY

The two common measurements of real-time determinism are interrupt latency and jitter. For simulations running at a fixed frame rate, interrupt latency is not a concern, as stated above. This leaves jitter as the primary metric for real-time determinism.

Two types of jitter were measured for this analysis: the *frame start jitter* and *frame run jitter*. Both types of jitter have the same root cause, namely that the application execution was pre-empted by the operating system, possibly to run some other process. Jitter at the start of the frame is caused when execution is preempted at the time the frame is supposed to start,

thus delaying it. Frame run jitter is variation of timing at the end of the frame and is caused by a pre-emption in the middle of the frame processing.

Minimizing jitter at the start of the frame is usually important to ensure that critical time stamping and I/O occurs at regular intervals. Jitter at the end of a frame usually is not as problematic unless it severe enough to delay beyond the start of the next frame, a condition called *frame overrun*. Figure 1. helps to explain how these terms are defined.

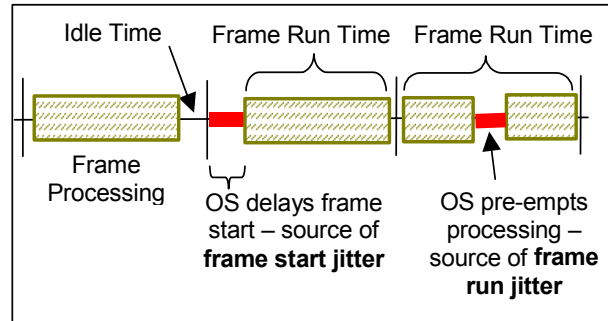


Figure 1. Frame Timing and Sources of Jitter

For these tests, frame run jitter is measured by first calculating the *frame run time*, which is the difference between the time the frame actually started application processing and the time it ended. The variance of the frame run time around its average is the frame run jitter. This defines frame run jitter as relative to the time the frame processing actually started, not to the time it was supposed to start. That way, frame run jitter is measured independently from frame start jitter.

An additional measure of determinism can be obtained by adding the worst-case frame start and frame run times. Over the course of a relatively short test, these two worst-case numbers are unlikely to happen in the same frame. But, since they are measured independently, they can be added to determine the potential for their sum to exceed the frame period, giving an additional quality metric of deterministic operation beyond that of seeing no frame overruns in a test.

The primary variables for the tests are as follows. Every combination of these six variables was configured and tested, resulting in 52 test runs, each one hour in length.

- Single vs. dual CPU
- Intel Xeon Hyperthreading
- Disked vs. diskless systems
- Linux 2.4.20 kernel vs. 2.5 kernel
- CPU affinity enabled vs. disabled
- Number of real-time processes

All tests were run with a 100 Hz frame rate. A representative mix of I/O and both integer and floating point math processing was done. The same amount of frame processing was done in all tests, except as noted in the tests with Hyperthreading enabled. This amount was found in pre-testing to fill 85% to 95% of the frame time (8.5 to 9.5 ms) on average for all test conditions. The minimum, maximum, and average amount of frame processing time and the number of overframe events that occurred in the one-hour test are also given.

The tests were run on 2.4 GHz Intel Xeon PCs. Both single and dual CPU machines were tested. Both were capable of Hyperthreading, which is the ability of Intel Xeons and newer Pentium 4 processors to appear as two logical CPUs managed in hardware. The two logical CPUs share some resources so Hyperthreading does not provide twice the overall performance, but two processes or threads can be executed in parallel without requiring any overhead from the operating system scheduler. When Hyperthreading is enabled in the BIOS, each physical processor appears as two CPUs to the operating system.

For tests involving diskless operation, the PXE (Pre-Execution Environment) feature available in the BIOS of nearly all PCs was used for booting. All boot information comes from a file server via the network. No boot floppy, CD, or ROM is required in the real-time PC. This does require the kernel to be rebuilt for diskless booting. This process is not trivial, but experience has shown that rebuilding the kernel for diskless operation when upgrading to a new version of Linux can be done with minimal effort.

No graphical console X server was run on the real-time test PCs. Of course, real-time nodes can still be X clients to allow graphical debugging through a remote login.

Ethernet I/O was done over a private 100 Mbit network to another device that simply echoed all packets coming out of the test system back to it as input packets. No other network was connected during the tests. No file I/O was performed, even on systems that had disks.

For the tests that set CPU affinity, all non-real-time processes were locked to CPU 0 after boot-up. As the real-time test processes started, they were each locked to a different CPU, thus ensuring that a CPU was dedicated to only that one process and no other. These tests did not attempt to run more real-time processes

than processors, thus allowing a dedicated CPU per real-time process.

The two primary means of controlling the start of a real-time simulation frame are to spin on a timer read operation or to sleep until interrupted, usually by some type of interval timer. The spin-wait is less complicated which gives it more potential for deterministic behavior, but it consumes the entire CPU while spinning. The interval timer allows the process to sleep and let other background processing occur. Its disadvantage is that the operating system now has complete control of the start of each frame, potentially with a loss of some determinism. The spin timing method was used for the 52 tests of the primary variables. Real-time priority was not used for these 52 tests.

Two more tests were added that used an interval timer as the frame timing mechanism. The main real-time process sleeps during the idle time at the end of its frame and the interval timer interrupt brings it out of the sleep at the start of the next frame. A background thread was run during this idle time but was not measured for performance in any way. This required the use of real-time priority for the main process to ensure it got scheduled at the interval timer start time ahead of the background thread.

TEST RESULTS

Disk vs. Diskless

The first tests were done on a single-CPU computer both with and without a disk and no Hyperthreading. Four configurations are possible, the results of each shown in Table 1. Only one process was run for each test. CPU affinity is not meaningful with a single CPU.

In these result tables, a highlighted background is used to point out overframes, max runtimes over 10 ms, the max start plus max run time sums that exceed 10 ms, and start jitter that exceeds 1 ms. Start jitter of less than 200 microseconds is shown in **bold**.

One notable result from this test is that the 5 to 10 ms jitter that is often cited for standard Linux does not appear here. This could be due to the improvements over older kernel schedulers or to the fact that there was no user interface operation (keyboard, mouse, graphics) or disk I/O. It is apparent that diskless systems are better for real-time use than those with disks. However, all tests showed some overframing

Table 1. Disk vs. Diskless, 2.4 and 2.5 kernel for Single CPU, no Hyperthreading

Disk or Diskless	Kernel	Affinity	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
Disk	2.4.20	N/A	1	225	0.901	1.310	9.040	10.350	9.111	11.251
	2.5.69	N/A	1	3	0.726	1.770	8.970	10.740	9.466	11.466
Diskless	2.4.20	N/A	1	1	0.331	1.300	9.040	10.340	9.110	10.671
	2.5.69	N/A	1	1	0.397	1.460	8.950	10.410	9.468	10.807

and the sum of the start jitter and maximum run time is over the 10 ms frame period. The 2.5 kernel doesn't show a definite difference in this test.

As more variables are tested, the sheer number of tests generates large amounts of data. The complete set of results is given in the Appendix. For gleaning meaningful analysis, important sections are repeated here for discussion.

Single vs. Dual CPU

Analysis continues with the test configuration that resulted in the least jitter. The best combination is a dual-CPU with no Hyperthreading running a single real-time process with affinity set to lock that process to the second CPU while all other background and I/O processing is locked to the first CPU. Its results for the 2.4 kernel are shown in Table 2. compared to the single CPU diskless case from the previous section.

The advantage of adding the second CPU and setting CPU affinity is striking. Start jitter is reduced by nearly 300 μ sec and run jitter by almost a millisecond. The sum of the start jitter and maximum run time is comfortably under the 10 ms frame period. This is considered a very safe configuration for real-time use, even though it does have the cost disadvantage of

being able to run only one process per dual-CPU computer. That is mitigated somewhat by the fact that the cost of a dual-CPU over a single-CPU PC is not significant for many real-time simulation systems.

For all tests, the results of start and frame run time for each individual frame was binned for plotting as a histogram. An example for the frame run time of the diskless dual-CPU, non-Hyperthreaded test is shown in Figure 2.

It can be seen how the frame run jitter is calculated by subtracting the lowest bin (8.72 ms) from the highest bin (9.05 ms) to get 0.330 ms. Note that only the lower portion of the vertical scale is shown, cutting off the bars for the first few bins so that the small sample counts in the higher bins can be seen in the plot. Almost all test results fit this pattern, where the vast majority of frame samples fell into the few bins around the average (8.733 ms in this case) at the left side of the plot with just a few frames falling to the extreme right side of the plot. The histograms of frame start times are even less interesting, with almost all frames starting within the first few microseconds and a very few at the right-hand extreme. Because they are so similar, histograms for other test results are not presented here.

Table 2. Single vs. Dual CPU for diskless 2.4 kernel, no Hyperthreading

CPUs	Affinity	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
2	Yes	1	0	0.041	0.330	8.720	9.050	8.733	9.091
1	N/A	1	1	0.331	1.300	9.040	10.340	9.110	10.671

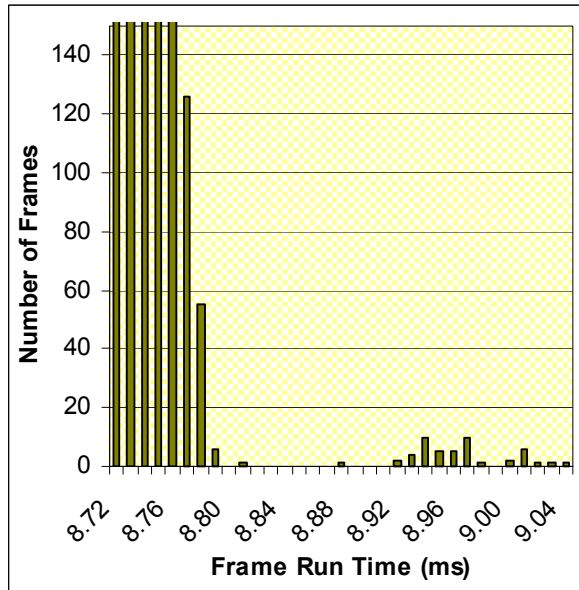


Figure 2. Histogram of Frame Run Time
Diskless, Dual-CPU, 2.4 Kernel, 1 Real-time Process
with CPU Affinity

Hyperthreading

The main advantage of Hyperthreading is to allow more real-time processes on the same number of physical CPUs. It should be noted that for the Hyperthreading tests with more than one process, the amount of work performed in each frame *per process* had to be cut by 25%. That is, each physical CPU could run two processes, each with 75% of the work done by the single process of the non-Hyperthreaded configuration. Thus, an overall *increase* of 50% was achieved in this particular test application. Of course, the amount of increase (or lack thereof) is dependent on the mix of instructions in an application.

The results of the Hyperthreading test are compared in Table 3. to the dual-CPU, one-process non-Hyperthreaded test. In that test, one CPU was unused for real-time processing and was left open for

background and I/O processing. Since Hyperthreading increases the number of CPUs that appear to the system to four, the number of real-time processes could be increased to three while still leaving one (logical) CPU for background and I/O processing.

It can be seen that jitter increases somewhat when Hyperthreading is enabled, but not terribly. In most cases, this would be considered a desirable tradeoff, gaining 50% more processing power in one of the CPUs and the ability to run three processes on a single computer. This is especially true when porting from a multi-processor platform where the simulation application was designed with shared-memory communication. Shared memory is possible between the three processes with Hyperthreading but would require specialized hardware between computers if only one process per computer were allowed.

It is interesting to note how much faster process 1 ran even though it was doing the same amount of work as processes 2 and 3. This is because processes 2 and 3 were locked to the logical processors that share a physical CPU. These are called "sibling" processors in Linux. Process 1 is locked to the logical processor 1 that is a sibling to processor 0 where non-real-time background and I/O tasks are locked. These tasks are not as busy as the real-time process, thus giving it a larger share of actual CPU time.

CPU Affinity

The improvement in determinism gained by setting CPU affinity makes it one of the most important features to be added to the Linux kernel. With a single real-time process on a dual-CPU computer, affinity has less effect because the Linux scheduler tends not to migrate the single busy process. But with Hyperthreading and more processes running, process migration is more likely. Table 4. shows the 3-process tests with Hyperthreading used in the previous section and compares it to a test with no CPU affinity set but identical in every other way.

Table 3. Hyperthreading for a Diskless 2.4 Kernel, Dual-CPU with Affinity

Physical CPUs	Hyper-thread	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
2	No	1	0	0.041	0.330	8.720	9.050	8.733	9.091
	Yes	1 of 3	0	0.142	0.620	6.570	7.190	6.640	7.332
		2 of 3	0	0.188	0.440	8.690	9.130	8.803	9.318
		3 of 3	0	0.141	0.380	8.710	9.090	8.803	9.231

Table 4. Affinity vs. No Affinity for a Diskless 2.4 Kernel, Dual-CPU with Hyperthreading

Physical CPUs	Affinity	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
2	Yes	1 of 3	0	0.142	0.620	6.570	7.190	6.640	7.332
		2 of 3	0	0.188	0.440	8.690	9.130	8.803	9.318
		3 of 3	0	0.141	0.380	8.710	9.090	8.803	9.231
	No	1 of 3	130	1.564	3.930	6.610	10.540	8.612	12.104
		2 of 3	0	0.084	0.600	6.540	7.140	6.607	7.224
		3 of 3	2	0.167	2.220	8.300	10.520	9.152	10.687

These results show the importance of CPU affinity in achieving real-time determinism. Without affinity, both start jitter and frame run jitter are adversely affected, to the point where they may be unacceptable for some real-time simulation applications.

Number of Processes

The effect of running a fourth process on the logical CPU that was previously set aside for background and I/O is shown in Table 5. Process 1 of 4 was given 10% less processing to compensate for sharing the logical CPU that was also used for background and I/O processing.

Processes 2, 3, and 4 are locked to the same logical CPUs as processes 1, 2, and 3 are in the 3-processes test. It can be seen that they have nearly identical results between the two tests. Also, the 10% reduction was about the right amount of work for process 1 in the 4-process test because the average runtime is similar to the other three processes. However process 1 of 4 does not have very good determinism results. This is not

particularly surprising, but it does prove that any process that shares the background and I/O CPU should not be expected to have determinism that is as high as the other real-time processes.

However, that might change when the 2.6 kernel becomes available, as shown in the next section when this test is repeated on the current developmental 2.5 kernel.

The 2.5 Kernel

In general, the 2.5 kernel performed better than the 2.4 in nearly every test. One example is the 4-process test described previously. There, as expected, the process locked to the background and I/O CPU didn't fare as well as the other three. Table 6. shows very different results with the 2.5 kernel.

Note that process 1 of 4 runs much better with the 2.5 kernel. This and the overall 2.5 test results bode well that the 2.6 kernel will be even better than 2.4 for real-time applications when it becomes available.

Table 5. 3 vs. 4 Real-Time Processes for a Diskless 2.4 Kernel, Dual-CPU, Hyperthreaded, with Affinity

Physical CPUs	Kernel	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
2	2.4.20	1 of 3	0	0.142	0.620	6.570	7.190	6.640	7.332
		2 of 3	0	0.188	0.440	8.690	9.130	8.803	9.318
		3 of 3	0	0.141	0.380	8.710	9.090	8.803	9.231
		1 of 4	127	2.077	4.680	5.980	10.660	8.387	12.737
		2 of 4	0	0.116	0.870	8.040	8.910	8.646	9.026
		3 of 4	0	0.124	0.440	8.560	9.000	8.669	9.124
		4 of 4	0	0.034	0.420	8.550	8.970	8.662	9.004

Table 6. 2.4 vs. 2.5 Kernel for a Diskless, Hyperthreaded Dual-CPU with Affinity

Physical CPUs	Kernel	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
2	2.4.20	1 of 4	127	2.077	4.680	5.980	10.660	8.387	12.737
		2 of 4	0	0.116	0.870	8.040	8.910	8.646	9.026
		3 of 4	0	0.124	0.440	8.560	9.000	8.669	9.124
		4 of 4	0	0.034	0.420	8.550	8.970	8.662	9.004
	2.5.69	1 of 4	0	0.149	1.070	7.740	8.810	7.863	8.959
		2 of 4	0	0.038	0.590	8.720	9.310	8.985	9.348
		3 of 4	0	0.031	0.740	8.430	9.170	8.825	9.201
		4 of 4	0	0.126	0.510	8.660	9.170	8.818	9.296

Interval Timer Frame Timing

Some simulation applications require multiple, prioritized threads or processes with the highest priority process running with hard real-time framing and the others running while the main process sleeps. This requires an interrupt-style of interval timer to bring the main process out of its sleep at the beginning of each frame. The results of two tests using that type

of frame timing are shown here, the first for the 2.4 kernel shown in Table 7. and the other for the 2.5 kernel shown in Table 8. These tests consisted of a main process run with real-time priority and a background thread that was not assigned priority.

Table 7. shows that the interval timer makes start jitter slightly worse, but not enough to cause overframing. The sum of the start jitter and maximum run time remains well under the 10 ms frame period. Therefore,

Table 7. Spin vs. Interval Frame Timer for a Diskless 2.4 Kernel, Hyperthreaded Dual-CPU with Affinity

Physical CPUs	Spin or Interval Timer	Kernel	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
2	Spin	2.4.20	1 of 3	0	0.142	0.620	6.570	7.190	6.640	7.332
			2 of 3	0	0.188	0.440	8.690	9.130	8.803	9.318
			3 of 3	0	0.141	0.380	8.710	9.090	8.803	9.231
	Interval	2.4.20	1 of 3	0	0.391	0.300	7.550	7.850	7.632	8.241
			2 of 3	0	0.205	1.460	7.360	8.820	8.517	9.025
			3 of 3	0	0.104	1.240	7.510	8.750	8.498	8.854

Table 8. Spin vs. Interval Frame Timer for a Diskless 2.5 Kernel, Hyperthreaded Dual-CPU with Affinity

Physical CPUs	Spin or Interval Timer	Kernel	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)	Max Start + Max Run Time
2	Spin	2.5.69	1 of 3	0	0.025	0.560	6.660	7.220	6.906	7.245
			2 of 3	0	0.058	0.460	8.650	9.110	8.776	9.168
			3 of 3	0	0.007	1.670	7.420	9.090	8.776	9.097
	Interval	2.5.69	1 of 3	0	1.006	0.300	7.540	7.840	7.642	8.846
			2 of 3	0	1.027	1.680	7.570	9.250	8.801	10.277
			3 of 3	0	1.001	2.250	7.240	9.490	8.581	10.491

this style of frame timing is quite possible with standard Linux for the 2.4 kernel. In fact, several programs at Boeing have successfully integrated standard Linux into simulation applications that require a mix of hard real-time framing and event-driven, multi-threaded processing using real-time priority settings.

The test results in Table 8. show an anomaly in the interval timer in the 2.5 kernel. The frame start has a consistent 1.0 ms jitter, much worse than that of the spin-timer framing in the 2.5 kernel and of the interval timer in the 2.4 kernel for the same test conditions. The 1.0 ms value is significant because it matches the scheduler clock period (1000 Hz clock rate) in the 2.5 kernel. This could be due to a change in the kernel design that requires a different setup and usage of the interval timer or some other irregularity given that this is not yet a stable kernel. Analysis of the cause of this anomaly was not attempted since 2.5 is a developmental kernel and not considered ready for use in real training system programs. Certainly, this must be resolved before using the 2.6 kernel in an operational system that requires an interval timer.

CONCLUSIONS

Standard Linux has proven to be a practical, low-cost, and easy to use solution for real-time simulation. Ordinary Linux distributions are suitable for use in frame-based real-time simulations on common, inexpensive PC hardware and with only minor reconfiguration of the kernel. This is all achieved while maintaining a fully robust memory-protected run-time system, a standard software development environment, support for all required languages, all required system services, and a full set of simulator I/O capabilities.

The upcoming 2.6 Linux kernel has even better potential for real-time determinism under an even wider range of uses with little or no kernel reconfiguration. However, one unresolved issue with the use of an interval timer in the 2.5 kernel was found and requires further study.

The easiest means to achieve real-time determinism is to run one real-time process per processor, use CPU affinity to lock each process to its own CPU, and lock all other system processes to a CPU not used for real-time processes. If an additional process is locked to that shared CPU, it must have lower performance and determinism requirements. Hyperthreading should be used more to save hardware cost than to gain

performance. It is also useful to allow shared-memory communication between a larger number of processes. While some overall performance gain can be achieved with Hyperthreading, determinism will be slightly lower because of contention for resources in the physical CPU such as the floating-point unit.

Of course, the results presented here are no guarantee that all real-time applications will run successfully on regular Linux. Much is dependent on the makeup and number of processes and threads, the system services used, and which of the innumerable types of I/O are required. There is no test result or benchmark that is as good as the application itself for proving the suitability of a computational environment. However, with the results gained from these tests, simulation programs can eliminate many Linux configurations as being unsuitable and focus on the smaller set of variables that are important for testing their application.

Further testing and analysis of a more complicated mix of priority-based, multi-threaded, interrupt-driven applications would be the next step in bringing standard Linux further into the fold of a broader range of simulation applications. The results obtained in these tests show great promise that almost any simulation requirement can be met with standard Linux when it is configured properly. And, that will only get easier with continuing improvements to the ever-evolving Linux kernel.

From a full computational system standpoint, running three real-time processes locked with CPU affinity on a dual-CPU PC with Hyperthreading enabled is the best tradeoff between cost and real-time performance available with current computer technology. Standard Linux is poised to take full advantage of that configuration, continually lowering the cost of full-fidelity training simulation.

ACKNOWLEDGEMENTS

I would like to thank my colleagues Dave Clauson and Tom Brooks, without whom this paper would not have been possible. Dave was indispensable in setting up the myriad Linux kernel and PC configurations, writing scripts to run the tests, and helping to interpret results with his knowledge of the Linux kernel. Tom helped with writing test code and setting up scenarios for testing the large number of variables.

REFERENCES

- Burch, B., Hughly, P., McCulley, G., & Dietrich, D. (2000). CCTT SAF on a PC, *Proceedings from the 2000 Interservice/Industry Training, Simulation and Education Conference*, 1345-353
- Fenrich, C., Campbell, M., & Zuffoletti, M. (1999). Linux on a PC: A viable RT Graphics Workstation, *Proceedings from the 1999 Interservice/Industry Training, Simulation and Education Conference*, 554-560
- Kern, C., Foster, J., & Callahan, B. (2001). The Migration of Close Combat Tactical Trainer to Linux, *Proceedings from the 2001 Interservice/Industry Training, Simulation and Education Conference*, 1300-1308
- Love, R. (May 2003). Introducing to the 2.6 Kernel, *Linux Journal*, Issue 109, 52-57
- Love, R. (July 2003). CPU Affinity, *Linux Journal*, Issue 111, 18-22
- Nalepka, J., Williams, G., Dube, T., Bryant, R. B., & Danube, T. (2001). Real-Time Simulation Using Linux, *AIAA Modeling and Simulation Technologies Conference*, Paper AIAA 2001-4185 (File A0137307.pdf)
- Proctor, F. (2002). *Introduction to Linux for Real-Time Control*. National Institute of Standards and Technology, Intelligent Systems Division, Retrieved June 2003 from <ftp://ftp.isd.mel.nist.gov/pub/isd/RealTimeLinuxReport-2.0.0.pdf>

Appendix – Raw Test Results

These tables contain the results from the 52 tests for the six primary variables. Rows in the table with Process labeled “*m* of *n*” indicate results from the *n* processes that ran simultaneously during a single test.

Single CPU

Hyper-thread	Disk or Diskless	Kernel	Affinity	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)
No	Disk	2.4.20	N/A	1	225	0.901	1.310	9.040	10.350	9.111
		2.5.69	N/A	1	3	0.726	1.770	8.970	10.740	9.466
	Diskless	2.4.20	N/A	1	1	0.331	1.300	9.040	10.340	9.110
		2.5.69	N/a	1	1	0.397	1.460	8.950	10.410	9.468
Yes	Disk	2.4.20	No	1	187	0.571	1.380	8.940	10.320	9.029
				1 of 2	270	1.212	3.690	6.780	10.470	9.058
				2 of 2	0	0.434	1.080	8.180	9.260	8.716
			Yes	1	1	0.002	1.290	8.840	10.130	8.925
				1 of 2	0	1.229	1.640	7.820	9.460	8.131
				2 of 2	0	0.015	0.410	8.530	8.940	8.719
		2.5.69	No	1	0	0.424	1.050	8.830	9.880	9.293
				1 of 2	0	0.408	0.660	8.680	9.340	8.754
				2 of 2	0	0.055	0.880	8.670	9.550	9.084
			Yes	1	0	0.046	0.940	8.790	9.730	9.241
				1 of 2	0	0.386	0.840	7.960	8.800	8.169
				2 of 2	0	0.049	0.370	8.590	8.960	8.760
	Diskless	2.4.20	No	1	1	0.047	1.330	9.060	10.390	9.140
				1 of 2	0	0.031	1.090	8.520	9.610	8.598
				2 of 2	0	0.022	0.350	9.080	9.430	9.200
			Yes	1	0	0.003	0.810	8.760	9.570	8.825
				1 of 2	0	0.031	1.190	8.060	9.250	8.215
				2 of 2	0	0.003	0.430	8.310	8.740	8.573
		2.5.69	No	1	0	0.021	1.070	8.720	9.790	9.179
				1 of 2	0	0.036	0.840	8.580	9.420	8.878
				2 of 2	0	0.050	1.010	8.670	9.680	8.996
			Yes	1	0	0.020	1.030	8.700	9.730	9.170
				1 of 2	0	0.057	1.360	7.380	8.740	8.015
				2 of 2	0	0.024	0.780	8.470	9.250	8.805

Dual CPU, No Hyperthreading

Disk or Diskless	Kernel	Affinity	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)
Disk	2.4.20	No	1	3	1.022	2.640	9.030	11.670	9.137
			1 of 2	1	0.314	1.480	8.850	10.330	8.940
			2 of 2	280	3.412	3.790	8.970	12.760	9.156
		Yes	1	0	0.086	0.780	8.840	9.620	8.892
			1 of 2	145	3.492	3.690	8.400	12.090	8.441
			2 of 2	0	0.096	0.910	8.890	9.800	9.057
	2.5.69	No	1	0	0.356	1.000	8.780	9.780	8.979
			1 of 2	0	0.333	1.250	8.760	10.010	8.966
			2 of 2	0	0.104	0.700	8.880	9.580	9.100
		Yes	1	0	0.098	0.700	8.800	9.500	8.995
			1 of 2	0	0.340	0.930	8.390	9.320	8.588
			2 of 2	0	0.099	0.690	8.880	9.570	9.100
Diskless	2.4.20	No	1	2	0.013	1.420	8.950	10.370	9.048
			1 of 2	0	0.207	0.800	9.050	9.850	9.192
			2 of 2	129	1.243	1.420	8.780	10.200	8.798
		Yes	1	0	0.041	0.330	8.720	9.050	8.733
			1 of 2	132	1.337	1.700	8.690	10.390	8.833
			2 of 2	0	0.162	0.320	8.720	9.040	8.739
	2.5.69	No	1	0	1.159	1.070	8.720	9.790	9.060
			1 of 2	0	0.097	1.280	8.730	10.010	9.095
			2 of 2	0	0.158	0.790	8.820	9.610	9.186
		Yes	1	0	0.019	0.940	8.620	9.560	8.925
			1 of 2	0	0.110	0.960	8.530	9.490	8.893
			2 of 2	0	0.043	0.990	8.610	9.600	8.945

Dual CPU With Hyperthreading

Disk or Diskless	Kernel	Affinity	Process	Over-frames	Start Jitter (ms)	Run Jitter (ms)	Min Runtime (ms)	Max Runtime (ms)	Average Runtime (ms)
Disk	2.4.20	No	1 of 3	18	2.952	9.140	6.640	15.780	8.463
			2 of 3	1	0.713	3.020	7.000	10.020	9.226
			3 of 3	118	2.719	4.220	6.600	10.820	6.947
			1 of 4	0	0.672	1.500	8.060	9.560	8.628
			2 of 4	1	0.799	2.080	8.020	10.100	8.745
			3 of 4	144	2.920	5.170	6.650	11.820	8.722
		Yes	4 of 4	0	0.022	1.220	8.520	9.740	9.430
			1 of 3	0	0.063	0.710	6.550	7.260	6.638
			2 of 3	0	0.005	1.180	8.040	9.220	8.892
			3 of 3	1	0.189	3.300	6.940	10.240	9.013
			1 of 4	144	2.522	5.860	5.910	11.770	8.421
			2 of 4	0	0.132	1.160	7.810	8.970	8.652

