Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux

Steve Brosky and Steve Rotolo
Concurrent Computer Corporation
2881 Gateway Drive, Pompano Beach, FL 33069

Abstract

The low latency and preemption patches provide significant progress making standard Linux into a more responsive system for real-time applications. These patches allow guarantees on worst case interrupt response time at slightly above a millisecond. However, these guarantees can only be met when there is no networking or graphics activity in the system. This paper will describe the implementation of shielded processors in RedHawk Linux and their benefits. It will also present the results of real-time performance benchmarks. Interrupt response time guarantees are significantly below one millisecond and can be guaranteed even in the presence of networking and graphics activity.

1. Introduction

Concurrent Computer Corporation has had more than a decade of experience in utilizing the shielded CPU model for attaining real-time performance under a real-time version of an SVR4 UNIX-based operating system. The key benefit of the shielded CPU approach is that it allows a commodity operating system to be used for applications that have hard real-time deadlines. Commodity operating systems like UNIX or Linux provide a benefit for these applications because they have large numbers of programmers that are familiar with the programming API, and there is a rich set of development tools and other application software available for these operating systems.

Shielded CPUs can provide more deterministic performance because the overhead of the operating system is essentially off-loaded onto a subset of CPUs in the system. A shielded CPU is therefore able to provide a more deterministic execution environment. In applying the shielded processor model to Linux, several nuances were found which affected the expected behavior of processes running on shielded CPUs.

It should be noted that shielded CPUs are just one method for attaining real-time performance under the RedHawk Linux operating system. Shielded CPUs offer the most deterministic performance available under RedHawk. However, RedHawk contains many real-time modifications including the preemption patch, low latency patches, a real-time scheduler, BKL hold time reduction and modifications to the way that *softirqs* are handled. These modifications allow RedHawk to attain real-time performance guarantees even when shielded CPUs are not utilized, for example on a uni-processor system.

2. The shielded CPU model

The shielded CPU model is an approach for obtaining the best real-time performance in a symmetric multiprocessor (SMP) system. This approach does not apply to uniprocessor systems. The shielded CPU model allows for both deterministic execution of a real-time application as well as deterministic response to interrupts. A task has deterministic execution when the amount of time it takes to execute a code segment within that task is predictable and constant. Likewise the response to an interrupt is deterministic when the amount of time it takes to respond to an interrupt is predictable and constant.

When the worst-case time measured for either executing a code segment or response to an interrupt is significantly different than the typical case, the application's performance is said to be experiencing *jitter*. Because of computer architecture features such as memory caches and because of contention for shared resources, there will always be some amount of jitter in measurements of execution times. Real-time applications are defined by the fact that they must respond to real world events within a predetermined deadline. Computations that are completed after this deadline are considered incorrect. This means that the worst-case jitter the operating system allows determines whether that operating system is suitable for hosting a given real-time application. Each real-time application must define the amount of jitter that is acceptable to that application.

In the shielded CPU model, tasks and interrupts are assigned to CPUs in such a way as to guarantee a high

grade of service to certain important real-time functions. In particular, a high priority task and a high priority interrupt are bound to one or more shielded CPUs, while most interrupts and low priority tasks are bound to *other* CPUs. The CPUs responsible for running the high-priority tasks are shielded from the unpredictable processing associated with interrupts and the other activity of lower priority processes that enter the kernel. Thus these CPUs are called *shielded CPUs*.

Some examples of the types of tasks that should be run on shielded CPUs are:

- tasks that require guaranteed interrupt response time
- tasks that require very fast interrupt response time
- tasks that must be run at very high frequencies
- tasks that require deterministic execution in order to meet their deadlines
- tasks that have no tolerance for being interrupted by the operating system

It will be shown that a shielded CPU can be used to guarantee deterministic execution and deterministic interrupt response times using a modified Linux kernel that presents a standard Linux API to the user. The benefit is that real-time applications can be developed using standard Linux interfaces and standard Linux debugging tools while still being able to guarantee very deterministic real-time performance.

3. Implementation of shielded processors

To create a shielded processor, it must be possible to set a CPU affinity for every process and every interrupt in the system. In this way a system administrator can define which processes and interrupts are allowed to execute on a shielded CPU. The Linux kernel already has support for CPU affinity in the form of an entry in the process structure for storing the CPU affinity and code in the scheduler that allows processes to run only on CPUs that are included in their CPU affinity. The only thing lacking in standard Linux is a user interface for setting a process' CPU affinity. Several open source patches provide this capability. Standard Linux does support a CPU affinity for interrupts. In this case, the user interface is already present via the */proc/irq/*/smp_affinity* files.

These two CPU affinity capabilities can allow a system administrator to set up a shielded processor, but it would require all processes and users in the system to honor the shielded processor by not explicitly changing their processor affinity to run on the shielded CPU. A less fragile mechanism for setting up a shielded CPU is desirable.

In addition, there are some interrupts that cannot be assigned a CPU affinity. The local timer interrupt interrupts every CPU in the system, by default at a rate of 100 times per second or once every 10 milliseconds. This interrupt is generally the most active interrupt in the system and therefore it is the most likely interrupt to cause jitter to a real-time application. The local timer interrupt provides functionality such as the accounting of CPU execution time, system profiling and CPU resource limits. The shielded processor mechanism allows this interrupt to be disabled. Some of the functionality, such as CPU time accounting, can be accomplished via other techniques. Other functionality more geared towards debugging and performance analysis, such as profiling, is simply lost when this interrupt is disabled.

A new set of /proc files was added to a new directory, */proc/shield*, to allow the system administrator to specify a bit mask of CPUs that should be shielded. It is possible to shield a CPU from both interrupts and processes. Separate files control shielding a CPU from processes, interrupts that can be assigned to a CPU and the local timer interrupt. It is possible to shield a CPU from all of these activities or just a subset.

Because we do want the ability to have some processes and some interrupts active on a shielded CPU, it was necessary to create a semantic for the interaction of process and interrupt affinity with the shielded CPU mask. In general, the CPUs that are shielded are removed from the CPU affinity of a process or interrupt.

The only processes or interrupts that are allowed to execute on a shielded CPU are processes or interrupts that would otherwise be precluded from running unless they are allowed to run on a shielded CPU. In other words, to run on a shielded CPU, a process must set its CPU affinity such that it contains *only* shielded CPUs.

When one of the /proc files that controls CPU shielding is modified, the shielded CPU is dynamically enabled. This means that the affinity masks of all processes and interrupts are examined and modified accordingly. The processes currently assigned to the shielded processor will no longer be allowed to run on that processor and will be migrated to other CPUs. Because the affinity mask associated with interrupts is also modified, the shielded CPU will handle no new instances of an interrupt that should be shielded. Optionally, the local timer interrupt may also be disabled on a shielded CPU. The ability to dynamically enable CPU shielding allows a developer to easily make modifications to system configurations when tuning system performance.

4. RedHawk kernel

Before describing the test scenarios that were used, it is necessary to describe the RedHawk Linux kernel, which was used for running benchmark tests that show the effect

of shielded processors. The RedHawk kernel used was version 1.4. RedHawk is a Linux kernel based on kernel.org 2.4.21. Various open source patches have been applied to this kernel to augment both real-time functionality and real-time performance including the MontaVista preemption patch [1], Andrew Morton's low-latency patches [2], Ingo Molnar's 0(1) scheduler patch [3] and the POSIX timers patch [4]. Other changes have also been incorporated by Concurrent for improving real-time performance. This includes further low-latency work and the implementation of shielded processors. In addition, support was added for the Concurrent manufactured Real-Time Clock and Interrupt Module (RCIM) PCI card. The RCIM provides the ability to connect external edge-triggered device interrupts to the system and also supports additional high-resolution timers. It will be shown how the RCIM driver is an important part of the RedHawk strategy for supporting deterministic interrupt response under Linux.

5. Determinism in execution

Determinism refers to a computer system's ability to execute a particular code path within a fixed amount of time. The extent to which the execution time for the code path varies from one instance to another indicates the degree of determinism in the system. Determinism applies not only to the amount of time that is required to execute a time-critical portion of a user's application, but also to the amount of time that is required to execute system service code in the kernel.

The standard Linux kernel has already addressed some of the primary causes of non-deterministic execution. For example Linux supports the ability to lock an application's pages in memory, preventing the jitter that would be caused when a program first accesses a page not resident in memory and turning a simple memory access into a page fault. Linux also supports strict priority-based scheduling so that the highest priority real-time processes are guaranteed to get as much CPU time as they require without having their priority eroded by scheduling fairness algorithms. Finally, the 0(1) scheduler, which was adapted in the Linux 2.5 series, provides scheduling overhead which is both constant and minimal.

Previous experience with creating a real-time variant of UNIX showed that the primary remaining cause of indeterminism in program execution is interrupts. Because interrupts will preempt the execution of even the highest priority task, interrupts are essentially the highest priority activity in the system. An interrupt can occur at any point in time because it is asynchronous to the operation of the programs executing in the system. This means that interrupts can cause significant jitter to a real-time application because they cause delays in program execution at unpredictable points in time.

5.1. Execution determinism test

For this test, the system used was a dual processor 1.4GHz Pentium 4 Xeon with 1GB of RAM and a SCSI hard drive.

Because we are measuring CPU execution determinism, it is desirable to have an application which is CPU bound for this measurement. The determinism test simply measures the length of time it takes to execute a function using double precision arithmetic to compute a sine wave. The sine function is called in a loop such that the total execution time of the outer loop should be around one second in length. Before starting this loop, the IA32 TSC register is read and at the end of the loop the TSC register is again read. The difference between these two high-resolution times represents the amount of time required to perform this CPU-bound loop. The test locks its pages into memory and is scheduled under the SCHED_FIFO scheduling policy.

The base time for the determinism test is based on the ideal case of running the CPU-bound loop and was determined by running the test on an unloaded system. Both kernels under test were tried in an unloaded state. The best time was measured under RedHawk on a shielded CPU.

Subsequently, the test was run under various kernel configurations with a load on the system. Any run of the CPU-bound loop that took more time than the ideal case was considered to have been impacted by indeterminism in the system. The difference between the worst-case time it took to run the CPU-bound loop and the ideal case represents the amount of jitter.

To measure jitter, the background workload run on the system should generate significant interrupt traffic. Two shell scripts were used to create Ethernet and disk interrupts. The first script was run on a foreign system and it copies a compressed kernel boot image over the Ethernet to the system being tested:

```
while true
do
     scp bzImage wahoo:/tmp
done
```

The second test generates disk traffic on the system by running a shell script that recursively concatenates files:

```
dir=/tmp/disknoise$$
trap 'rm -rf $dir; exit 1' 1 2 15

mkdir $dir
cd $dir

echo boo >9
cnt=0
```

```
while true
do
     for f in 0 1 2 3 4 5 6 7 8 9
     do
           cat * >$f 2>/dev/null
     done
     cnt=`expr $cnt + 1`
     if [ $cnt -ge 3 ]
     then
           rm *
           echo boo >9
           cnt=0
     fi
done
```

Note that while this workload will generate interrupt traffic, it is not a particularly interrupt-intensive burden on the system.

5.2. Execution determinism results

The determinism test was first run on a standard Linux kernel (kernel.org 2.4.21). The figure below graphs the amount of variance from the ideal case in milliseconds. This means that a deterministic run would have a graph that has the majority of its data points on the left hand side of the graph. Also of interest is the worst-case time observed executing the computational loop. The results for the kernel.org kernel are summarized in Figure 1. The results are also summarized in terms of the ideal time it took to execute the code path, maximum time and the amount of jitter. The jitter reported is the difference between the maximum amount of time it took to run the computational loop and the ideal time it took to run the computational loop, expressed in both seconds and as a percentage of the ideal case.

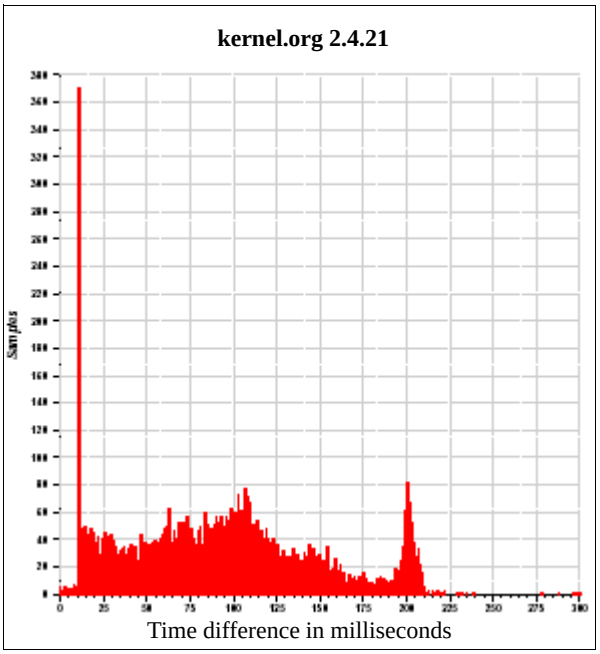

Figure 1. kernel.org 2.4.21 Results

```
ideal: 1.147132sec
  max: 1.447316sec
jitter: 0.300184sec (26.17%)
```

Clearly there was significant variance in the amount of time it took to run the computational loop on a standard Linux kernel when the system is busy with a load that causes interrupt traffic. In the worst case, the computational loop, which should have taken 1.15 seconds, took an additional 300 milliseconds to complete.

The test was next run on the RedHawk 1.4 kernel, on a shielded processor. Figure 2 graphs the amount of variance from the ideal case with a summary of the results in the legend below the graph.

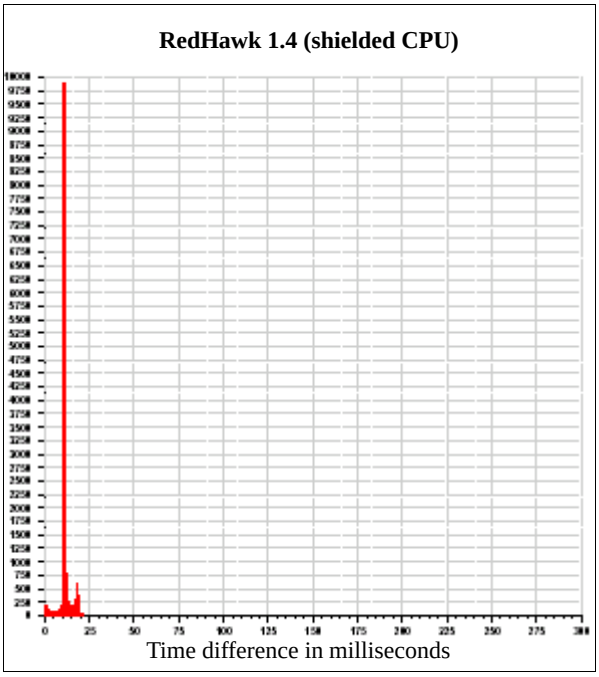

Figure 2. RedHawk 1.4 Shielded CPU Results

```
ideal: 1.147132sec
  max: 1.168630sec
jitter: 0.021498sec (1.87%)
```

As expected, a shielded processor provides a significant improvement in the amount of variance that we see from the ideal case. In the worst case, the computational loop, which should have taken 1.15 seconds, took an additional 21milliseconds to complete. This jitter is assumed to be due to memory contention in an SMP system.

To be sure that the improvement in determinism was due to shielding and not other differences in the system, the test was next run on the RedHawk 1.4 kernel on a non-shielded processor. Figure 3 graphs the amount of variance from the ideal case with a summary of the results in the legend below the graph.

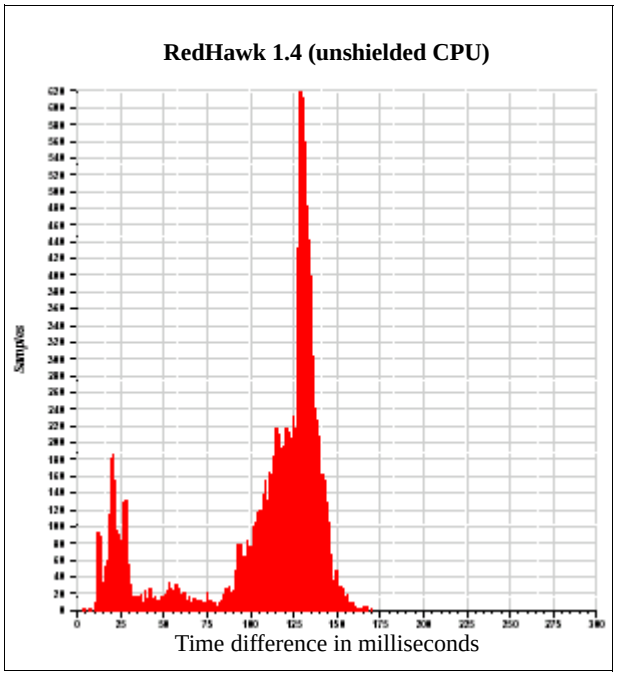

Figure 3. RedHawk 1.4 Unshielded CPU Results

```
ideal: 1.147132sec
  max: 1.317151sec
jitter: 0.170019sec (14.82%)
```

The test confirmed that the interrupt load on an unshielded processor does indeed cause greater jitter in the execution time for executing a computational load.

However, the determinism on a non-shielded CPU was still significantly better than standard Linux. Why were the standard Linux results as bad as they were? It was theorized that the cause was the fact that this version of Linux enables hyperthreading. Note that hyperthreading is disabled by default in RedHawk. A final version of the test was run on the standard Linux kernel with hyperthreading disabled via the GRUB prompt. Figure 4 graphs the amount of variance from the ideal case with a summary of the results in the legend below the graph.

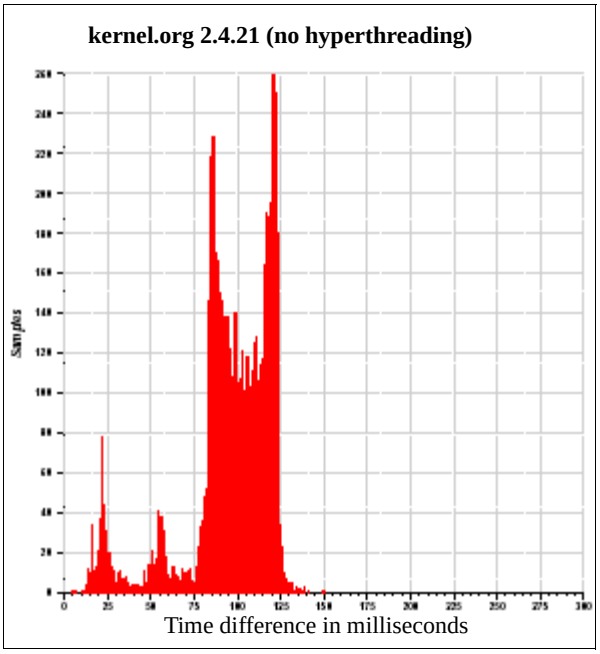

Figure 4. kernel.org 2.4.21 (no hyperthreading) Results

```
ideal: 1.147132sec
  max: 1.298005sec
jitter: 0.150873sec (13.15%)
```

This test clearly identifies hyperthreading as the culprit for even greater non-deterministic execution. While hyper-threading does offer a performance boost for a multi-threaded application by enabling parallelism at the instruction unit level, this chip feature causes another layer of indeterminism for real-time applications. This is because with hyperthreading enabled, the execution unit itself has become a point of contention between the processes that are executing on the virtual processors of a single CPU.

6. Interrupt response

Because real-time applications must respond to real world events and those events are communicated to the computer via an interrupt, determinism in responding to an interrupt is an especially important metric for a real-time operating system.

There are existing open source patches that address some of the issues in the standard Linux kernel for achieving good interrupt response. One such patch is the kernel preemption patch [1]. This patch allows one process to preempt another process currently executing inside the kernel. Prior to this patch when one process did a system call, no other process could execute inside the kernel until that process either blocked or completed its system call. This has the potential to lead to very long delays when trying to wake a high-priority process that is awaiting an interrupt when there is currently a non-preemptible task executing in the kernel.

Even with the preemptible kernel patch there are remaining issues with preempting a process that is executing inside the kernel. When a process makes a system call, that process might enter into one of the Linux kernel's critical sections. A critical section is an area of code that accesses a shared kernel data structure. Because the data structure is shared, it might be simultaneously accessed by another process that is executing inside the kernel. To prevent the shared data from being corrupted, a critical section requires synchronization primitives that allow only one process at a time to access the shared data. The preemptible kernel patch does not allow a process to be preempted while it is inside a critical section, since the preempting process might try to modify the shared data of the pending critical section, causing the shared data to be corrupted.

Because the kernel's critical sections cannot be preempted, the length of the critical sections inside the kernel is significant when considering worst-case interrupt response. In the Linux 2.4 series, there are many very long critical sections. Other open source patches collectively known as the "low-latency patches" [2] address the longest critical sections in the kernel by rewriting the algorithms involved so preemption can be disabled for a shorter period of time. The combination of the preemption patch [1] and the low-latency patch [2] sets was used on a Red Hat based system to demonstrate a worst-case interrupt response time of 1.2 milliseconds [5].

Experience working with a real-time variant of UNIX showed that when trying to guarantee how long it will take to respond to an interrupt, the biggest problem is the critical sections that disable preemption. Consider the case where a low priority process enters the kernel to process a system call and that process enters a critical section where preemption is disabled. If a high priority interrupt becomes active at this point, the system will process that interrupt. But when the interrupt routine wakes the process that is awaiting the interrupt, that process will not be able to run until the execution of the critical section is complete. This means that the worst-case time to respond to an interrupt is going to be at least as long as the worst-case time that preemption is disabled in the kernel.

In a symmetric multiprocessor system that supports CPU shielding, it is possible to prevent low priority processes from running on a CPU where a very fast response to interrupt is required. While this means that some CPU resources will be underutilized, it does allow a very firm guarantee for processes that require a high degree of interrupt response.

6.1. Interrupt response test

The system used for the test was a dual 933MHz Pentium 3 Xeon with 2GB of RAM with a SCSI disk drive. Two different kernels were measured under the same load conditions.

To measure interrupt response time, the *realfeel* benchmark from Andrew Morton's website was initially used. This test was chosen because it would allow results to be compared between a standard Linux kernel and a RedHawk system. This test operates by measuring the response to an interrupt generated by the Real-Time Clock (RTC) driver. This driver is set up to generate periodic interrupts at a rate of 2048 Hz. The RTC driver supports a read system call, which returns to the user when the next interrupt has fired. The clock used to measure interrupt response is the IA32 TSC timer. To measure interrupt response time, the test first reads the value of the TSC and then loops doing reads of /dev/rtc. After each read the test gets the current value of the TSC. The difference between two consecutive TSC values measures the duration that the process was blocked waiting for an RTC interrupt. The expected duration is 1/2048 of a second. Any time beyond the expected duration is considered latency in responding to an interrupt. The test locks it pages into memory and is scheduled under the SCHED_FIFO scheduling policy.

To measure worst-case interrupt response time, a strenuous background workload must be run on the rest of the system. The workload chosen was the same as that used in Clark William's paper on Linux Scheduler Latency [5]. This workload is from the Red Hat stress-kernel RPM. The following programs from stress-kernel are used:

NFS-COMPILE
TTCP
FIFOS_MMAP
P3_FPU
FS
CRASHME

The NFS-COMPILE script is the repeated compilation of a Linux kernel via an NFS file system exported over the loopback device. The TTCP program sends and receives large data sets via the loopback device. FIFOS_MMAP is a combination test that alternates between sending data between two processes via a FIFO and operations on an mmap'd file. The P3_FPU test does operations on floating point matrices. The FS test performs all sorts of unnatural acts on a set of files, such as creating large files with holes in the middle, then truncating and extending those files. Finally the CRASHME test generates buffers of random data, then jumps to that data and tries to execute it. Note that while no Ethernet activity was generated on the system, the system did remain connected to a network and was handling standard broadcast traffic during the test runs.

6.2. Interrupt response results

The first kernel used was a standard Linux (kernel.org 2.4.21). Note that this kernel does not contain the low-latency patches [2] or the preemption patch [1]. After starting the stress-kernel program, realfeel was run for 60,000,000 samples at 2048 Hz. The test was terminated before the full eight-hour run completed because we already had enough data showing poor interrupt latency on this kernel. Figure 5 graphs the interrupt response for a standard Linux kernel. Note that the y axis is a logarithmic scale. This graph is summarized in terms of histogram buckets below the graph.

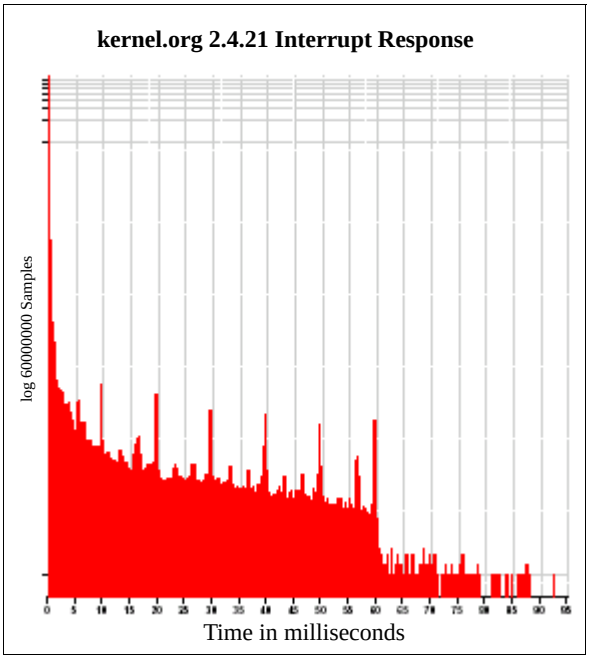

Figure 5. kernel.org 2.4.21 Interrupt Response Results

```
measured 44759417 rtc interrupts
max latency: 92.3ms

44374681 samples < 0.1ms (99.140%)
44594353 samples < 0.2ms (99.631%)
44687849 samples < 1.0ms (99.843%)
44702467 samples < 2.0ms (99.872%)
44719462 samples < 5.0ms (99.910%)
44732301 samples < 10.0ms (99.939%)
44742797 samples < 20.0ms (99.962%)
44748489 samples < 30.0ms (99.975%)
44753080 samples < 40.0ms (99.985%)
44756536 samples < 50.0ms (99.993%)
44759250 samples < 60.0ms (99.999%)
44759363 samples < 70.0ms (99.999%)
```

```
44759402 samples < 80.0ms (99.999%)
44759416 samples < 90.0ms (99.999%)
44759417 samples < 100.0ms (100%)
```

While the majority of the responses to the interrupts occurred in less than 100 microseconds, for a real-time application the most important metric is the worst-case interrupt response. This graph shows that standard Linux, without the patches that implement minimal real-time performance gains, has very poor guarantees on interrupt response. At 92 milliseconds, the worst-case interrupt response is completely unacceptable for the vast majority of real-time applications. These results are expected.

The second kernel tested was the RedHawk 1.4 kernel described above. After starting the stress-kernel program, realfeel was run for 60,000,000 samples at 2048 Hz. This run was approximately 8 hours in length. While the length of this run may seem like overkill, early results showed us that on a shielded CPU the worst-case numbers might not occur until several hours into the run.

In this test, CPU 1 was set up as a shielded processor. The RTC interrupt and realfeel have their CPU affinity set such that they run on shielded CPU 1. The stress-kernel test is run without any CPU affinity set. The results of the interrupt response test for a RedHawk shielded processor are presented in Figure 6. Again, the results are also summarized in histogram form below the graph.

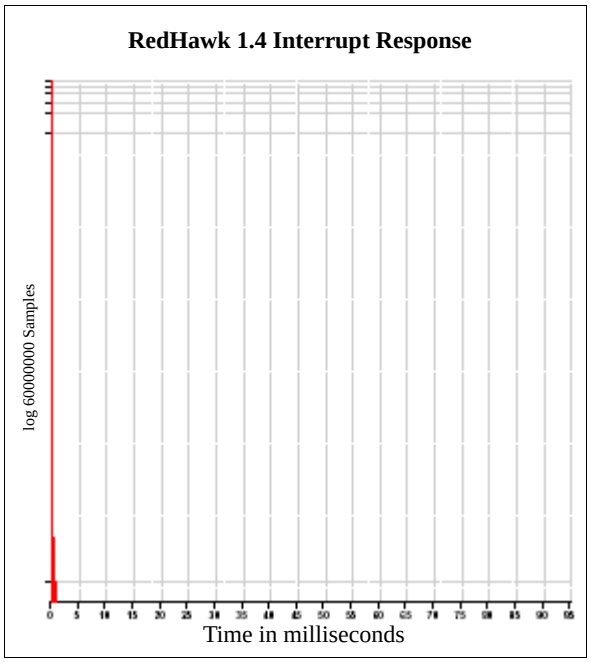

Figure 6. RedHawk 1.4 Interrupt Response Results

```
measured 60000000 rtc interrupts
max latency: 0.565ms
```

```
59999983 samples < 0.1ms (99.99997%)
       8 samples < 0.2ms
       5 samples < 0.3ms
       2 samples < 0.4ms
       1 samples < 0.5ms
       1 samples < 0.6ms
```

The initial tests run under RedHawk on shielded CPUs showed worse results than expected. The problems discovered resulted in several additional fixes to the Linux kernel to allow us to achieve a more optimal interrupt response on a shielded processor. The primary problem was due to critical sections that are protected by spin locks that do not disable interrupts. It is not necessary for these spin locks to disable interrupts because the critical section is never locked at interrupt level. When interrupts are not disabled, it is possible for an interrupt routine to preempt a critical section being protected by a spin lock. Because interrupt routines are relatively short, this should not be a big issue. The problem was in the bottom-half interrupt routines that would run on return from interrupt. These interrupt bottom halves sometimes executed for several milliseconds of time. If the process used to measure interrupt response on the shielded processor attempts to lock the contended spin lock (which had been preempted by interrupts and bottom-half activity) during the read of /dev/rtc, then the response to the interrupt could be delayed by several milliseconds.

Because the /dev/rtc mechanism works via the read() system call, a process that wakes up after the interrupt fires must now exit the kernel through various layers of generic file system code. Embedded in this code are opportunities to block waiting for spin locks. The /dev/rtc interface is therefore not ideal for achieving a guaranteed interrupt response.

6.3. A second interrupt response test

While the initial experiment did succeed in reducing interrupt latency below one millisecond, the results were not as good as expected for shielded CPUs. It was theorized that these mediocre results were due to the fact that the realfeel test uses /dev/rtc, whose API is considered less than optimal, as described above. Therefore a new interrupt response test was designed. In this test the real-time timer on the Real-Time Clock and Interrupt Module (RCIM) PCI card was utilized as the interrupt source.

To block waiting for the RCIM's timer to interrupt, the user makes an ioctl() call rather than a read() system call. In addition, the Linux kernel was modified to correct one of the issues found with this interrupt response test. Linux locks the Big Kernel Lock (BKL) spin lock before entering a device driver's ioctl routine. This is to protect legacy drivers that are not properly multithreaded from having issues on an SMP system. The problem is that the BKL

spin lock is one of the most highly contended spin locks in Linux and attempting to lock it can cause several milliseconds of jitter.

A change was implemented to the generic ioctl support code in Linux so it would check a device driver specific flag to see whether the device driver required the BKL spin lock to be held during the driver's ioctl routine. This allows a device driver that is fully multithreaded to avoid the overhead of the BKL. Since the RCIM driver is multithreaded, it does not require the BKL to be locked during its ioctl routine.

Like realfeel, the RCIM interrupt response test measures the amount of time it takes to respond to an interrupt generated by a high-resolution timer. When the RCIM is programmed to generate a periodic interrupt, the length of the periodic cycle is stored in the RCIM's count register. The count register is decremented until it reaches zero, at which time an interrupt is generated. When the count reaches zero, the RCIM will also automatically reset the count register to its initial value and begin decrementing the count register for expiration of the next periodic cycle.

The RCIM interrupt response test operates by initiating a periodic interrupt on the RCIM and then, in a loop, issuing the ioctl to block until an interrupt is received. When the test is awakened after receipt of the interrupt, it immediately reads the value of the count register on the RCIM. Because this register can be directly mapped into the program, the overhead of this read is minimal. The test can then calculate the time since the interrupt fired by subtracting the current value of the counter register from the initial value loaded into the count register at the beginning of each periodic cycle. The test locks its pages into memory and is scheduled under the SCHED_FIFO scheduling policy.

In this test scenario the workload was significantly increased from that used during the realfeel benchmarking above. The same stress-kernel load was used, but in addition, the X11perf benchmark was run on the graphics console and the ttcp network performance benchmark was run, reading and writing data across a 10BaseT Ethernet connection.

The test was run on a dual processor 2.0 GHz Pentium 4 Xeon with 1GB of RAM and SCSI disks. The Ethernet controller is the 3Com 3c905C-TX/TX-M. The graphics controller is the nVidia GeForce2 MXR.

Because this interrupt response test requires the RCIM driver, which is not a part of standard Linux, no numbers were gathered for a standard Linux kernel. The results for running this test on RedHawk 1.4 are shown in Figure 7. Note that the numbers in this thin bar histogram represent microseconds, not milliseconds. The y axis is a logarithmic scale.

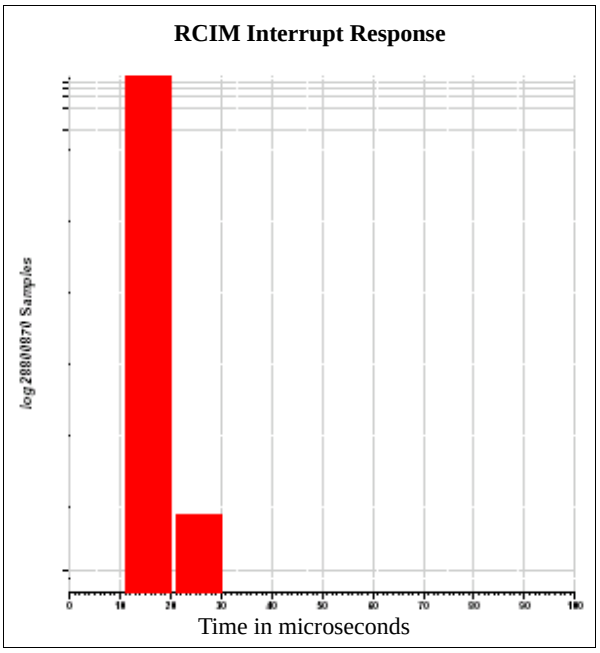

Figure 7. RCIM Interrupt Response Results

```
measured 28800882 RCIM interrupts
minimum latency: 11 microseconds
maximum latency: 27 microseconds
average latency: 11.3 microseconds

28800870 samples < 0.02ms (99.99999%)
      12 samples < 0.03ms
```

This test demonstrates that the issues seen with the realfeel test have to do with the multithreading issues of /dev/rtc. When the RCIM driver is used to generate a high-resolution timer interrupt, a shielded processor is able to provide an absolute guarantee on worst-case interrupt response time of less than 30 microseconds.

7. Conclusion

It has been demonstrated that processes executing on shielded processors on a standard Linux kernel substantially improve the determinism in the time it takes to execute a user-level application. Enabling hyperthreading on the Xeon chip causes another level of contention between the processes that are executing on the virtual CPUs provided by hyperthreading and causes a decrease in program execution determinism.

It has also been demonstrated that when an interrupt and the program that responds to that interrupt are run on a shielded processor, it is possible to guarantee interrupt response of less than 30 microseconds. This guarantee can be made even in the presence of heavy networking and graphics activity on the system. This interrupt response

guarantee rivals the guarantees that can be made by much smaller and much less complex real-time kernels. There are remaining multithreading issues to be solved in the Linux kernel to achieve this level of interrupt response for other standard Linux application programming interfaces which generate interrupts.

References

[1] Robert Love, “preemptible kernel patches,” 4 March 2002, <http://www.kernel.org/pub/linux/kernel/people/rml/preempt-kernel/> (23 January 2003).

[2] Andrew Morton, “scheduling latency,” 7 January 2001, <http://www.zipworld.com.au/~akpm/linux/schedlat.html> (23 January 2003)

[3] Ingo Molnar, “0(1) scheduler,” 4 July 2002, <http://people.redhat.com/mingo/O(1)-scheduler/> (23 January 2003)

[4] George Anzinger, “posix timers,” 31 July 2001, <http://sourceforge.net/projects/high-res-timers> (23 January 2003)

[5] Clark Williams, 2002, Linux Scheduler Latency, Red Hat web cast.

RTLit0015-0504