A Concurrent Real-Time White Paper

800 NW 33rd
Pompano Beach, FL 33064
(954) 974-1700
www.concurrent-rt.com

Tuning a RedHawk™ Linux® System

Maximizing real-time application performance
on Concurrent's RedHawk Linux

By: Mark Slater

Concurrent Systems Analyst

June 2020

Abstract

This paper discusses tuning techniques that will aid the user in taking full advantage of Concurrent's RedHawk Linux real-time operating system. It is intended as a starting point for a user who is new to RedHawk, and it provides a broad look at the capabilities available to optimize the performance of real-time applications. Those users serious about maximizing performance on their RedHawk Linux systems should also review the relevant sections of the RedHawk Linux Release notes and the RedHawk Linux Users Guide.

Overview

RedHawk is a real-time Linux distribution that includes the Red Hat® Enterprise Linux, CentOS Linux or Ubuntu Linux user environment and a Linux kernel from kernel.org specifically modified by Concurrent for real-time applications. The kernel contains some open source patches and Concurrent supplied modifications that provide low-latency, highly deterministic performance. Taking full advantage of RedHawk's capabilities requires some basic knowledge of the system architecture and the methods available to harness system resources and put them to the best use in executing a real-time application with maximum determinism and lowest latency. The user should be aware of three areas where deviation from default settings/methods may be beneficial to real-time applications:

System Tuning:

- Kernel tunables that can be modified to create a custom, special-purpose kernel.
- CPU shielding that isolates CPU cores from certain system activities.
- Vectoring of interrupts that are important to the application, to shielded CPUs for handling.

Execution Control:

- Vectoring processes to shielded CPUs.
- Picking a scheduling policy and priority.
- Taking advantage of shielded memory on Non-Uniform Memory Architecture (NUMA) nodes where the architecture permits.

Programming Practices:

- Good programming practices that help an application execute in the most efficient manner.

This paper will discuss these techniques and the various ways they can be implemented using the command line, scripts, NightStar™ debugging tools and programming APIs.

System Tuning

System tuning refers to the configuration of the system prior to the start of any application executables. Some aspects of system tuning can be accomplished dynamically, for example, the shielding of processor cores and vectoring interrupts to or from those shielded CPU cores. Other system tunables can be built into a custom kernel configuration.

Kernel Tunables

Static system tuning is accomplished via modification of kernel tunables and the building and booting of custom kernel configurations. The mechanics of modifying kernel tunables and building custom kernels can be found in the RedHawk Linux Users Guide, Chapter 11. A certain amount of system tuning is performed upon installation of the RedHawk kernel. A short list of non-essential services found to be detrimental to low-latency, deterministic performance is turned off, and several modifications are made to parameters found in the sysctl.conf and rsyslog.conf files.

Ultimately, it will be the demands of the user's application that dictate whether any additional kernel tuning is necessary. Table B-1 Kernel Tunables for Real-Time Features found in the RedHawk Users Guide, Appendix B, is a comprehensive list of kernel tunables that are relevant on a RedHawk system. They are grouped by functionality along with notes about where they can be found in the kernel configuration tool, their default values in the prebuilt RedHawk kernels, and where in the Concurrent documentation set they are discussed.

CPU Shielding

Shielding CPU cores and NUMA memory from normal Linux activity and reserving them to service high-priority components of the user application has long been a powerful and easily implemented strategy for improving real-time performance. The benefits of shielding are described extensively in Concurrent white paper Real-Time Linux: The RedHawk Approach.

There are a number of ways for the user to shield system resources: from the command line using the shield(1) command, from the NightTune™ (ntune) tool which provides a GUI interface to system shielding, and using cpuctl(3) which provides programmatic access to CPU shielding and up/down control. From the command line, shield(1) can be used at any time to dynamically change the state of system CPU and NUMA memory shielding. The shield(1) command can also be embedded in a system startup script like /etc/rc.local to execute at boot time and shield the system in a given way.

shield(1) has an extensive man page but its capabilities can be summed as follows. shield(1) can protect physical or virtual system CPUs and NUMA memory nodes from normal Linux system activity. This activity falls into three categories: processes in general, interrupts in general and core-specific interrupts and system services like the local timer interrupt. In the case of NUMA memory nodes, the shield(1) command can be used to reserve the memory associated with that node for the use by applications running on cores local to that node. The following examples demonstrate the use of shield(1).

Query the current state of shielding:

```
bash-4.1$ shield

CPUID   irqs   ltmrs   procs

  0      no      no      no
  1      no      no      no
  2      no      no      no
  3      no      no      no
  4      no      no      no
  5      no      no      no
  6      no      no      no
  7      no      no      no
```

Shield cores 1, 2, 3 and 7 from processes, interrupts and the local timer interrupt, and then display the results:

```
bash-4.1$ shield -a 1-3,7 -c

CPUID    irqs    ltmrs    procs

  0       no      no       no
  1       yes     yes      yes
  2       yes     yes      yes
  3       yes     yes      yes
  4       no      no       no
  5       no      no       no
  6       no      no       no
  7       yes     yes      yes
```

Modify this configuration by shielding CPU 4 from interrupts, CPU 5 from local timers and CPU 6 from processes and display the results. Then remove local timer shielding from CPU 1 and display them again:

```
bash-4.1$    shield   -i+4   -l+5   -p+6   -c   -l-1   -c

CPUID    irqs    ltmrs    procs

  0        no      no       no
  1        yes     yes      yes
  2        yes     yes      yes
  3        yes     yes      yes
  4        yes     no       no
  5        no      yes      no
  6        no      no       yes
  7        yes     yes      yes




CPUID    irqs    ltmrs    procs
```

0	no	no	no
1	yes	no	yes
2	yes	yes	yes
3	yes	yes	yes
4	yes	no	no
5	no	yes	no
6	no	no	yes
7	yes	yes	yes

The cpu(1) command is a helpful command line utility for querying the state of hyper-threading and shielding on the system. cpu(1) with no arguments displays the current state:

```
bash-4.1$ cpu

cpu   chip   core   ht   ht-sibs   state   shielding
 0     0      0     0       4        up       none
 1     0      1     0       5        up       none
 2     0      2     0       6        up       none
 3     0      3     0       7        up       none
 4     0      0     1       0        up       none
 5     0      1     1       1        up       none
 6     0      2     1       2        up       none
 7     0      3     1       3        up       none
```

By marking down CPU 6, we leave its hyper-threaded sibling CPU 2 alone on core 2, maximizing the determinism of any process running there:

```
bash-4.1$ cpu -d 6

cpu   chip   core   ht   ht-sibs   state   shielding
 0     0      0     0       4        up       none
 1     0      1     0       5        up       none
 2     0      2     0       6        up       none
 3     0      3     0       7        up       none
 4     0      0     1       0        up       none
 5     0      1     1       1        up       none
 6     0      2     1       2       down      none
 7     0      3     1       3        up       none
```

Shielding and CPU “up/down” control is also provided via the “CPU Shielding and Binding” dialogue in the NightTune (ntune) tool. Right clicking in that window will display a “Change Shielding” pulldown option as shown in Figure 1.

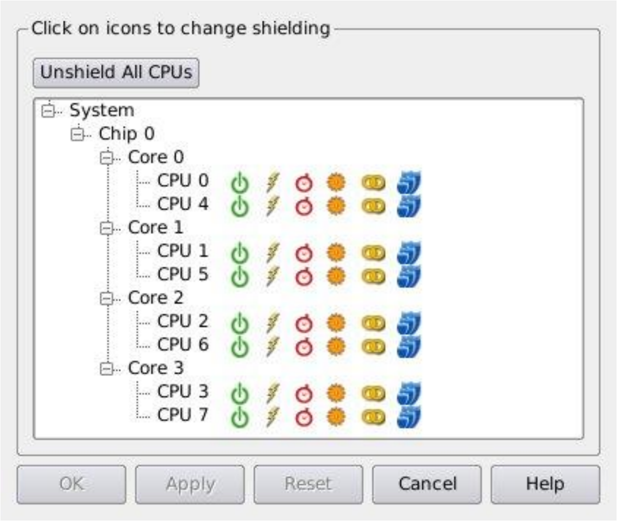
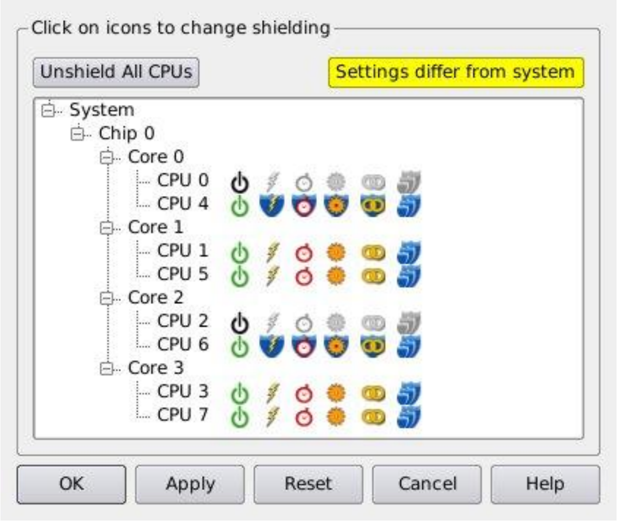


Figure 1

This dialogue lists CPUs by chip with hyper-threaded sibling CPUs grouped together. The first five symbols from left to right are toggles. The first is the 'Active' button - green means the CPU is up and black means the CPU is down and can be activated. The next three symbols represent the current state of shielding for general interrupts, local timer interrupts and processes. Individually toggling these icons will apply or remove the shield icon from that item. The shield icon means that the core has been shielded from that type of system activity.

The fifth icon is another way of turning off or marking down a hyper-threaded sibling. The sixth icon on the far right is a collection of three shields that specifies total shielding of that CPU. As shown in Figure 2, selecting that icon for CPU 6 automatically marks down the hyper-threaded sibling and shields the CPU from interrupts in general, the local timer interrupt and processes in general.



Settings differ from system

Figure 2

The last method of shielding control is via the cpuctl(3) API. cpuctl(3) provides the user with the ability to programmatically query and alter the state of CPU shielding and the ability to mark CPUs up or down. Used in conjunction with the mpadvise(3) and cpuset(3) API calls, users can construct subroutines whose purpose is to query available resources, shield those resources and change the CPU affinity of real-time processes to take advantage of those shielded resources. These APIs require that the user run as the root user or have the CAP_SYS_NICE privilege (see questions 10 and 11 in the RedHawk Linux FAQs) and link in the Concurrent Real-Time library libccur_rt (-lccur_rt). Appendix A of this paper lists a short example of how a user might utilize these calls.

Hyper-threading

The previous section referred to the user's ability to affect the state of hyper-threading of available cores. It is worth a moment to explain what hyper-threading is and how it may affect the performance of a real-time application — for better or worse. Platforms that support Intel® hyper-threading in the BIOS allow the user to configure a given core to be viewed as two virtual CPUs. These hyper-threaded "siblings" share the processors execution resources, but maintain separate instruction pipelines. Since modern processors can execute instructions much faster than those instructions can be fetched from memory, a processor can find itself with nothing to do for a considerable time. Instead, with hypertheading enabled, the processor can execute instructions from one pipeline or another, in effect acting as two virtual CPUs.

There are a number of considerations when deciding whether or how to utilize hyper-threading. Generally, processes that run well in parallel can take advantage of the increased throughput that hyper-threading provides. In general, the sharing of resources by hyper-threaded siblings introduces potential latency to the execution of a real-time process which can affect determinism. A process will execute with the most determinism when running on a CPU that does not have a hyper-threaded sibling. Understanding how hyper-threading works and understanding the requirements of the application can lead to some good strategies for maximizing performance on hyper-threaded siblings.

One such strategy is interrupt isolation. In the next section we will see that an interrupt deemed important to an application can be vectored to the same CPU that is running the application. That CPU can be shielded from all other interrupts assuring that the important interrupt will be handled immediately. Kernel tracing, however, may reveal that the application is delayed for a short time while the interrupt handler is completing the interrupt service. It may turn out to be a better decision to vector the important interrupt to the sibling CPU of the one running the application. The sibling CPU can be shielded from all other interrupts and processes, meaning the interrupt routine will stay in the CPU's L1 cache, thus improving its performance.

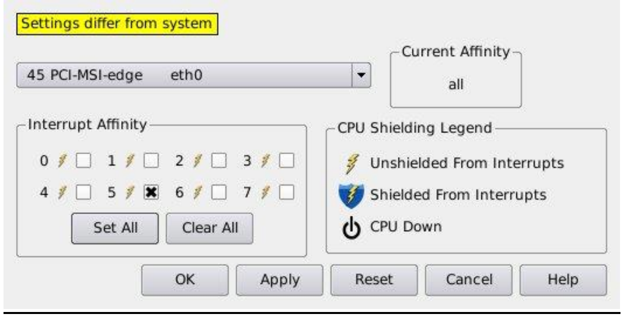
Because hyper-threaded siblings share execution resources including cache, client/server applications may benefit by sharing the cached data on the same physical CPU core. The cpu(1) command with the –C or --cache option can be used to view the CPU layout and which caches are shared between CPUs. If an application can use different threads to process the majority of its floating point calculations versus its integer calculations, those different threads can efficiently utilize the separate computational resources of hyper-threaded cores.

Interrupt Vectoring via CPU Affinity

In our discussion of shielding, we have seen how CPUs can be protected from system activity including interrupts. In many applications, interrupt response time is an integral and important part of the real-time determinism and low-latency requirements of the application. It is important to realize that interrupts are subject to preemption by interrupts of a higher priority. When looking at kernel trace data on an unshielded CPU, it is not unusual to see interrupt execution nested three deep. That is, a low-priority interrupt is interrupted by a middle-priority interrupt which is itself interrupted by a high-priority interrupt. When the high-priority interrupt service completes, the middle resumes and completes, finally leaving the low-priority interrupt to resume and complete. Interrupts important to the user's real-time application should be vectored to CPU cores that have been shielded from other interrupt activity.

The user should also be aware that hardware interrupts will often have a 'bottom half' associated with them. These bottom halves are softirq tasks that run after the hardware interrupt service and complete processing that doesn't need to occur at interrupt level. Kernel tunables make it possible to give these softirq tasks a priority 1 less than the maximum SCHED_FIFO priority, allowing a user's real-time process to preempt these softirqs. (See the discussion of SOFTIRQ_PRI and SOFTIRQ_PREEMPT_BLOCK in section "Softirqs and Tasklets" of Chapter 14 of the RedHawk Linux Users Guide.) The softirqs will execute on the same CPU that handles the hardware interrupt routine.

NightTune provides an easy-to-use interface for controlling interrupt vectors. From the "Monitor" pulldown, select "Interrupt Detail Activity" and then "Text Pane". This will display a list of all system interrupts along with current activity. On the left are IRQ number/names. Select the IRQ to be vectored by left-clicking and then right click to access the "Set CPU Affinity" dialogue as shown in Figure 3. From this dialogue you can select which CPU or collection of CPUs is permitted to service the particular IRQ.



Settings differ from system

45 PCI-MSI-edge eth0

Current Affinity: all

Interrupt Affinity

0 □ 1 □ 2 □ 3 □

4 □ 5 ☒ 6 □ 7 □

Figure 3

Altering the shielding of resources or vectoring of IRQs in this way will only persist until the next reboot of the system. NightTune has a mechanism that allows any tuning changes, including interrupt and process vectoring and CPU shielding, to be saved as a system state file and then restored at will via the "Restore System State" dialogue or automatically at boot time in batch mode. To take full advantage of this capability, the user must understand which attributes can be set at boot time and which can only be implemented post boot.

For example, if the user has used ntune to shield CPUs and vector interrupts, these modifications can be implemented immediately upon boot by running ntune as a one-time batch job. Let's assume the system state was saved as /home/state_1.config. The user could then guarantee the system was always booted to that state by adding the following to any system startup script:

```
/etc/rc.local:
ntune --wait=0 --config=\
/home/state_1.config
```

Now let's assume that in addition to any CPU shielding and interrupt vectoring, the user has directed specific processes or threads of processes to run on specific CPUs. Those processes or threads may not be active until sometime after booting is complete. In this case, the user will want to run ntune in daemon mode so that the daemon will remain actively looking for processes or threads to move. From the same startup script:

```
ntune --restore-state  --config=\ /home/state_2.config my_sys_config
```

where “my_sys_config” provides a handle by which later commands can be applied to the daemon. For example, after application startup, you may want to terminate the daemon with:

```
$ ntune --quit my_sys_config
```

Execution Control

In the previous section we discussed putting the system in the most favorable configuration to host real-time applications. Now we will discuss taking advantage of those changes when executing those processes. CPUs may have been shielded as a destination for real-time threads or important interrupts. NUMA architectures enable the “on-node” location of a process address space.

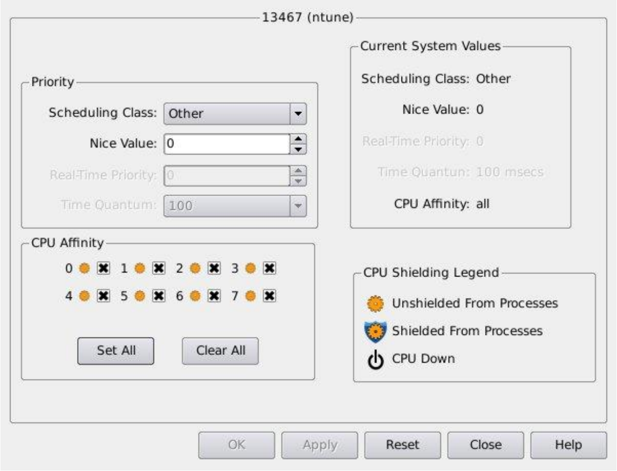
As in shielding, process vectoring can be accomplished via the command line, through an ntune GUI or performed programmatically via an API. From the command line, RedHawk’s run(1) command can be used to start a process with a given set of parameters or to adjust the run-time parameters of an existing process or thread. In general run(1) can control any run-time attribute of a process including CPU bias, scheduling policy, priority and memory policy. For example, if you have shielded CPUs 4 and 5, and you want two real-time processes to share CPU 4 with one having a higher priority than the other, and also a third process on CPU 5, you might use these run commands:

```
$ run -b4 -sfifo –P45 my_app_1
$ run -b4 -sfifo –P40 my_app_2
$ run -b5 -sfifo –P45 my_app_3
```

To move my_app_2 to CPU 6 leaving its other attributes the same:

```
$ run -b6 -n  my_app_2
```

Once processes and threads are established, NightTune provides one method to change scheduling policy and associated parameters and two ways to affect CPU affinity. From the “Process List” page, the user can highlight a given process or thread and right-click to access the “Process Scheduler” dialogue as shown in Figure 4. From here the user can alter both Policy and Priority-related parameters as well as the CPU mask of the process.



13467 (ntune)

Priority

Scheduling Class: Other

Nice Value: 0

Real-Time Priority: 0

Time Quantum: 100

Current System Values

Scheduling Class: Other

Nice Value: 0

Real-Time Priority: 0

Time Quantum: 100 msecs

CPU Affinity: all

CPU Affinity

0 ● ☒ 1 ● ☒ 2 ● ☒ 3 ● ☒

4 ● ☒ 5 ● ☒ 6 ● ☒ 7 ● ☒

CPU Shielding Legend

● Unshielded From Processes

● Shielded From Processes

⏻ CPU Down

Figure 4

From the same “Process List”, the user can drag a process or thread to the “CPU Shielding and Binding” page and drop the process onto a given CPU to bind the process to that CPU. Finally, process and thread scheduling parameters, CPU affinity, scheduling policy and priority can be controlled programmatically using mpadvise(3) and the pthread API. Appendix B of this paper provides an example of an application that creates threads which then set their own CPU affinity, scheduling policy and priority at start up. Before moving on, it is worth mentioning that any performance-critical process or thread should be run with the SCHED_FIFO scheduling policy. SCHED_FIFO dictates a different execution path through some kernel timers. Processes *not* run using SCHED_FIFO are subjected to some built-in slack designed to make the timers more efficient when multiple processes are utilizing them.

NUMA

NUMA refers to a system design that features separate memory nodes, each locally attached to a CPU socket that contains a multi-core CPU. The obvious consequence of this architecture is that a memory reference from a given CPU will either be to a local pool of memory or to a remote one. A process which has requirements for low latency and high determinism will want to make sure all of its memory references are to the local memory pool.

By default, a process started on a CPU of a given NUMA node will allocate pages for its address space from the memory pool local to the CPU as long as the page requests can be satisfied from that adjacent pool. Pages may be allocated from remote pools if necessary. For a process requiring real-time determinism, it is possible to use run(1) to demand that all pages be allocated from the local pool, and if they can't be, the process will terminate. It is also possible, using the shield(1) command to ensure that a given NUMA node's memory pool is shielded from page allocations from processes running on remote nodes thereby maximizing the amount of memory available to the real-time process. Furthermore, RedHawk will automatically identify any shared pages that the real-time process may be using, (system library text, read-only data), and replicate them on the shielded NUMA node's memory pool so that the real-time process can access all of its libraries and data from local, adjacent memory.

For example, the cpu(1) command on a four-socket system with 12-core CPUs (48 CPUs total) shows that CPUs 12-23 are all hosted on chip (socket) 1.

```
 bash-4.1$  cpu

cpu  chip  core  ht  ht-sibs  state  shielding
  0    0     0    -     -      up      none
  1    0     1    -     -      up      none
  2    0     2    -     -      up      none
  3    0     3    -     -      up      none
  4    0     4    -     -      up      none
  5    0     5    -     -      up      none
  6    0     6    -     -      up      none
  7    0     7    -     -      up      none
  8    0     8    -     -      up      none
  9    0     9    -     -      up      none
 10    0    10    -     -      up      none
 11    0    11    -     -      up      none
 12    1     0    -     -      up      none
 13    1     1    -     -      up      none
 14    1     2    -     -      up      none
 15    1     3    -     -      up      none
 16    1     4    -     -      up      none
 17    1     5    -     -      up      none
 18    1     6    -     -      up      none
 19    1     7    -     -      up      none
 20    1     8    -     -      up      none
 21    1     9    -     -      up      none
 22    1    10    -     -      up      none
 23    1    11    -     -      up      none
 (Output truncated)
```

This output also shows that there is currently no shielding of any kind in place. Let's shield CPU 13 from all activity then run my_app on CPU 13 with a scheduling policy of SCHED_FIFO, a priority of 90 and a memory policy of MPOL_BIND. This will ensure that my_app's pages will be allocated from the memory pool associated with the second NUMA node (chip 1).

```
bash-4.1$ shield -a 13
bash-4.1$ $ run -b13 -sfifo -P90 -Mb ./myapp
```

```
Myapp running
Myapp running
Myapp running
```

Now we will run my_app in a similar manner, but first we will shield the memory pool, reserving it for the pages that my_app allocates and any system library or read-only data pages that may need to be copied onto this node. Note that in order to shield a given memory pool, all of the CPUs in that NUMA node need to be shielded from processes. It is up to the user whether and to what extent those CPUs are also shielded from interrupts and local timers.

```
bash-4.1$ shield -p 12-23 -a+13 -m1 –c

CPUID   irqs   ltmrs   procs   mem
  0      no     no      no      no
  1      no     no      no      no
  2      no     no      no      no
  3      no     no      no      no
  4      no     no      no      no
  5      no     no      no      no
  6      no     no      no      no
  7      no     no      no      no
  8      no     no      no      no
  9      no     no      no      no
 10      no     no      no      no
 11      no     no      no      no
 12      no     no      yes     yes
 13      yes    yes     yes     yes
 14      no     no      yes     yes
 15      no     no      yes     yes
 16      no     no      yes     yes
 17      no     no      yes     yes
 18      no     no      yes     yes
 19      no     no      yes     yes
 20      no     no      yes     yes
 21      no     no      yes     yes
 22      no     no      yes     yes
 23      no     no      yes     yes
```

```
bash4.1$ run -b13 -sfifo -P90 -Mb ./myapp

Myapp running
Myapp running
```

Use the run(1) command with the –Mc option to list the NUMA nodes, their memory sizes and what CPUs are in each node.

Programming Practices

To this point, we have focused on setting up the system environment so that the demands of the operating system are isolated from the demands of the user's real-time application as far as available resources permit. It is important to realize that *how* the real-time application is written can also affect its real-time performance. The RedHawk Linux Users Guide has a wealth of detailed information on good programming practices, particularly in Chapter 2 "Real-Time Performance". It is worth mentioning a few of these techniques to give the user a flavor of what is possible when porting an application or developing a new application on a RedHawk system.

Memory Locking

One of the most basic techniques available is the ability to lock down your process' address space so that no paging will occur during run-time. A family of mlock(2) calls can be used to lock a part or all of a process' current and/or future address space requirements into memory. Memory locking is easily implemented when the user has access to application source and can recompile the application. RedHawk also allows the address space of a running executable to be locked down when source is not available via the run(1) command's --lock (-L) option. For example, to lock down all current and future pages of the running application graphics_app:

```
$ run -La -n graphics_app
```

The mlock(2) family of calls can also be used to memory lock binary tasks by invoking the binary task from a master process. In order to implement all the features of memory locking, the user will need the CAP_IPC_LOCK and the CAP_SYS_NICE capability.

Postwait

Processes often have the need to synchronize with other processes or threads of processes. A very efficient synchronization mechanism is provided by RedHawk via the postwait(2) family of system calls. A thread or collection of threads can voluntarily block on the pw_wait() call. A posting thread can wake up any given thread with a pw_post() call or wake up a collection of threads with the pw_postv() call. Appendix C provides a short example of using the postwait(2) calls.

Rescheduling Variables

Cooperating processes often need to share a system resource and they accomplish this by protecting that resource with a mechanism like spinlocks. Any process acquiring a spinlock needs to take precautions against being preempted while in possession of the lock or other cooperating processes could suffer. One way to do this is to use a system call to raise the process' priority to the highest in the system to preclude preemption, but this requires a non-trivial amount of overhead.

RedHawk Linux provides a better method called a rescheduling variable. A process may register a rescheduling variable with RedHawk at the beginning of execution, and then manipulate it using supplied macros when entering critical regions of code like the acquisition of a spinlock. Whenever the rescheduling variable is set, the process is temporarily immune to rescheduling and cannot be preempted by any other process. When the spinlock is returned, the rescheduling variable is cleared and the process assumes its normal relationship with processes competing for the same CPU. Spinlocks used by Ada applications developed with Concurrent's MAXAda™ compiler automatically use rescheduling variables, however POSIX® spinlocks or other types of spinlocks do not. Therefore, any application using the SCHED_FIFO or SHED_RR scheduling policy and spinlocks should use rescheduling variables to avoid the risk of a priority inversion deadlock. A short example of the use of rescheduling variables can be found in Appendix D of this paper.

Cache Thrashing

A real-time code developer should be aware of the possibility of a condition known as cache thrashing. This occurs when different CPUs constantly update variables on the same cache line. If two variables are both frequently used and on the same cache line, then the common cache will need to be updated anytime either variable is modified. It is a more desirable condition that frequently-used variables be located on different cache lines. A discussion of programming techniques that can assure this can be found in the RedHawk Linux Users Guide, Chapter 2 section on "Avoiding Cache Thrashing".

Conclusion

Achieving maximum performance, low latency and high determinism in a real-time application requires a basic understanding of both the system's architecture and the features of RedHawk Linux.

RedHawk Linux, like any other Linux or Unix® system, by default will run non-real-time applications without any special handling required of the user. In these cases, RedHawk is designed to take advantage of all system resources and make sure that all applications get their fair share of time on those resources. Using the special features and utilities provided by Concurrent's RedHawk, the user can easily alter the system configuration to favor real-time applications over general Linux activity and the activities of other users.

The Linux kernel itself can easily be customized with Concurrent's GUI-based customization tool to build a custom kernel configured exactly to the needs of the application. Using shielding, system resources can be protected from non-real-time activity and reserved for those processes requiring real-time performance. Shielding is easily implemented and can be done dynamically while experimenting with various configurations in real-time. Finally, maximum performance improvements can be made by reviewing the RedHawk Linux Users Guide for optimum programming strategies.

About Concurrent Real-Time

Concurrent Real-Time is the industries' foremost provider of high-performance real-time Linux computer systems, solutions and software for commercial and government markets. The Company focuses on hardware-in-the-loop and man-in-the-loop simulation, data acquisition, and industrial systems, and serves industries that include aerospace and defense, automotive, energy and financial. Concurrent Computer Corporation is headquartered in Atlanta with offices in North America, Europe and Asia. Concurrent Real-Time is located in Pompano Beach, Florida. For more information, please visit Concurrent Real-Time at www.concurrent-rt.com.

Appendix A

```
/*
 * Using cpuctl() to shield system CPUs
 * cc -o examp examp.c -lccur_rt
 */
#include <stdlib.h>
#include <cpuset.h>
#include <cpuctl.h>
#include <mpadvise.h>

void main()
{
 int CMD = CPUCTL_SHIELD, i=0;
 int FLAGS = CPUCTL_SHIELD_IRQ_FLAG|CPUCTL_SHIELD_PROC_FLAG|CPUCTL_SHIELD_LTMR_FLAG;
 cpu_t top_cpu;
 cpuset_t *cpuset_mask;
/*
 * Locate the highest number CPU on the system
 */
 if((top_cpu=cpuset_max_cpu())==0)
   {perror("Max CPU Failure");exit(-1);}
/*
 * Allocate a cpu set pointer
 */
 if((cpuset_mask=cpuset_alloc()) == 0)
   {perror("Set Allocation Failure");exit(-1);}
/*
 * Add the last four cpu cores to our cpu set
 */
 for (i=top_cpu;i > (top_cpu-4);i--)
   {
     if((cpuset_set_cpu(cpuset_mask,(cpu_t)i, 1))!= 0)
       {perror("Cpu set failure");exit(-1);}
   }
/*
 * Shield the list of cores according to FLAGS
 */
 if((cpuctl(CMD, FLAGS, cpuset_mask))!=0)
   {perror("Cpuctl failure");exit(-1);}
/*
 * Change the cpu set list to cores 2 & 3 using a string
```

Appendix A (Continued)

```
 * (This _does_ write over the previous set contents)
 */
 if((cpuset_set_list(cpuset_mask,"2,3"))!=0)
   {perror("Set List Failure");exit(-1);}
/*
 * Only shield cores 2 & 3 from IRQs
 */
 if((cpuctl(CMD, CPUCTL_SHIELD_IRQ_FLAG, cpuset_mask))!=0)
   {perror("Cpuctl failure");exit(-1);}
/*
 * And just for fun, move this process to the lowest number
 * completely shielded core
 */
 cpuset_init(cpuset_mask);          /* Clear the set */
 if((cpuset_set_cpu(cpuset_mask,(cpu_t)(top_cpu-3), 1))!= 0)
   {perror("Cpu set failure");exit(-1);}
/*
 * Migrate myself to the desired CPU
 */
 if((mpadvise(MPA_PRC_SETBIAS,MPA_TID,0,cpuset_mask )) == MPA_FAILURE)
   {perror("MPADVISE failure");exit(-1);}
 sleep(10);  /* So I can verify the migration */
}
```

Appendix B

```
/*
 * Thread self determination; CPU affinity, Scheduling Policy and Priority
 * cc -o examp examp.c -lccur_rt lpthread
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <cpuset.h>
#include <mpadvise.h>
 struct
 {
  pthread_t thrd_id;
  pthread_attr_t attr;
  cpuset_t *cpuset_mask;
  struct sched_param param;
 } thrd_stuff[3]    /*  thrd_stuff is global.  Main() and the_thread() both see it */
static void * the_thread(void *arg)
{
 Long loc_i=(long)arg;
 int j;
/*
 * Allocate and initalize a cpu_mask pointer
 */
 if((thrd_stuff[loc_i].cpuset_mask=cpuset_alloc()) == 0)
  perror("Set Allocation Failure");
  cpuset_init(thrd_stuff[loc_i].cpuset_mask);
/*
 * Migrate the threads to cpus 5,4 and 3
 */
if((cpuset_set_cpu(thrd_stuff[loc_i].cpuset_mask,(cpu_t)(5-loc_i), 1))!= 0)
   perror("Cpu set failure");
 if((mpadvise(MPA_PRC_SETBIAS,MPA_TID,0,thrd_stuff[loc_i].cpuset_mask )) == MPA_FAILURE)
   perror("MPADVISE failure");
/*
 * Set our threads Scheduling Policy to FIFO and give it a real-time priority
 */
   thrd_stuff[loc_i].param.sched_priority =95-loc_i;
   if((pthread_attr_init(&thrd_stuff[loc_i].attr))!=0)
     perror("Attribute init");
   if((pthread_setschedparam(thrd_stuff[loc_i].thrd_id,SCHED_FIFO,&thrd_stuff[loc_i].param))!=0)
```

Appendix B (Continued)

```
  perror("Policy & Priority set");
/*
 * Now do whatever this thread does
 */
 for(j=0;j<10;j++)
    { printf("Thread %d is cycling\n",arg);sleep(1);}
}
main()
{
 long i;
/*
 * Create three default threads. The threads themselves will set run parameters
 */
 for(i=0;i<3;i++)
  {
   if((pthread_create(&thrd_stuff[i].thrd_id,&thrd_stuff[i].attr,the_thread,(void *)i)) != 0)
      perror("Thread creator");
   sleep(1);
  }
/*
 * Let main() wait for our last thread to complete
 */
  if((pthread_join(thrd_stuff[2].thrd_id,NULL))!=0)
   perror("Thread join");
}
```

Appendix C

```
/*
 * Post / Wait, efficient thread wake up mechanism
 * cc -o examp examp.c -lccur_rt
 */
#include <stdio.h>
#include <sys/pw.h>
 ukid_t child, parent;

 void do_child_initialization(void) {
     printf("child initialization \n");
 }
 void do_child_work(void) {
     printf("child work \n");
 }
 void do_parent_initialization(void) {
     printf("parent initialization \n");
 }
 void do_parent_work(void) {
     printf("parent work \n");
 }

 void do_child(void) {
     do_child_initialization();

     /* implement a barrier with the parent */
     pw_post(parent, NULL);
     pw_wait(NULL, NULL);
     do_child_work();
 }
 void do_parent(void) {
     do_parent_initialization();
     /* implement a barrier with the child */
     pw_post(child, NULL);
     pw_wait(NULL, NULL);

     do_parent_work();
 }

 int main(void) {
     setbuf(stdout, NULL);
```

Appendix C (Continued)

```
    pw_getukid(&parent);

    switch (child = fork()) {
    case -1:
        perror("fork");
        exit(1);
    case 0:
     /* This is the child process.  gets its ukid_t */
        pw_getukid(&child);
        do_child();
        break;
    default:
     /* this is the parent process.  fork loaded child
      * with the child’s ukid_t  */
        do_parent();
        break;
    }
    return 0;
}
```

Appendix D

```
/*
 * Using Rescheduling variables and spin locks
 * cc -o examp examp.c -lccur_rt
 */
#include <stdio.h>
#include <spin.h>
#include <sys/rescntl.h>

static struct resched_var my_rv;  /* Define my rescheduling variable */
static struct spin_mutex my_spin; /* Define my spin_lock */
/*
 * Note that in a more comprehensive example, my_spin would be in a shared
 * memory area where other programs that compete for the resource would
 * see it.
 */
/* These macros properly acquire and release
 * a spin_lock while using a rescheduling variable
 * to protect from preemption
 */
#define spin_acquire(_m,_r) \
{ \
resched_lock(_r); \
while (!spin_trylock(_m)) { \
resched_unlock(_r); \
while (spin_islock(_m)); \
resched_lock(_r); \
} \
}
#define spin_release(_m,_r) {\
 spin_unlock(_m);\
  resched_unlock(_r); }

int main()
{
 /* Register this rescheduling variable with the kernel */
 resched_cntl(RESCHED_SET_VARIABLE, (char *) &my_rv);
 /* Initialize the spin lock */
 spin_init(&my_spin);
```

Appendix D (Continued)

```
/* We want to get our spin lock so that we may touch a shared resource.
   Let's use the macro that will set our rescheduling variable and acquire
   the lock all at once  */
  spin_acquire(&my_spin,&my_rv);

/* We may have spun, but we haven't been preempted, and now we have the lock.
   Go ahead and manipulate the resource. */

/* With resource manipulation complete, we can now return the spin_lock and unset the
   rescheduling variable */

  spin_release(&my_spin,&my_rv);
return 0;
}
```