Five Debugging and Analysis Tools Every Developer Needs for Time-critical Applications

Technology Overview
NightStar™ Software Development Tools for RedHawk™ Linux®

Mark Slater
August 2020
# Table of Contents

Abstract  ........................................................................................................... 3

Introduction ................................................................. 3

NightView  ......................................................................................... 4-10

NightTrace ................................................................. 10-16

NightProbe ................................................................. 16-18

NightSim  ......................................................................................... 19-22

NightTune ................................................................. 23-26

Conclusion ................................................................. 27
Abstract
This Technology Brief introduces Concurrent Real-Time’s NightStar suite of development tools. These five tools enable RedHawk Linux users to quickly and easily debug, monitor, schedule, analyze and tune applications in real-time.

Introduction
Concurrent Real-Time has been providing a real-time computing environment to customers with very specialized needs since 1966. The Company has migrated from a proprietary hardware and Operating System, to the more open combination of UNIX, VME & PowerPC, to today’s Linux, PCI & x86 and ARM systems.

The RedHawk real-time Linux Operating System running on customized iHawk PCI, x86 and ARM-based hardware provides customers with a computer system designed for applications that demand low latency and deterministic execution.

More than 50 years of experience in government and commercial applications worldwide uncovered common developer needs:

1) Software development
2) Scheduling software to run in particular ways and at particular rates
3) Run-time access to the memory space of the application for monitoring and recording
4) Application tuning to take maximum advantage of the architecture
5) System performance visibility at the sub-microsecond level.

In response, Concurrent Real-Time created the NightStar suite of development tools. These five tools address each need noted above and allow developers to seamlessly utilize the tools across OS releases and hardware architectures.

NightStar Development Suite Overview

NightView – a source level, full-featured debugger that provides special capabilities with RedHawk Linux.

NightTrace – a tool for analyzing the dynamic behavior of applications, the Linux Operating System and the interaction between them on a sub-microsecond timeline.

NightProbe – a tool for monitoring, modifying and recording data values from any of multiple sources: executing programs, shared or mapped memory segments or PCI registers.

NightSim – an interface to Concurrent Real-Time’s Frequency Based Scheduler (FBS) that provides a mechanism by which multiple programs or program threads can be scheduled cyclically and independently to run at user-specified rates.

NightTune – an interface to tuning and monitoring application and system performance.
NightView

A certain amount of software development is typically required for most Concurrent Real-Time customer applications. Since moving to UNIX in the late 1980s, users have had access to the GNU free and open debuggers, gdb and ddd, but Concurrent Real-Time developers immediately saw the need for a more feature-rich debugger.

Some of NightView's highlights include the ability to debug multiple processes and multiple process threads simultaneously and without stopping the process. NightView also supports the debugging of multiple applications written in any combination of C / C++, Ada and FORTRAN. All variables and expressions in each program are referenced in their own language syntax.

NightView's integration with RedHawk Linux provides for dynamic patching of eventpoint conditions so that these conditions are patched directly into an application and executed at full application speed. This patching capability can be used to alter program execution by any combination of skipping lines of existing code; adding new lines of code; replacing subroutine calls; or adding calls to previously non-existent subroutines.

NightView is able to debug an application's Heap memory without recompilation or re-linking. It watches for heap memory leaks, monitors the amount of memory an application uses, and tracks how it allocates and frees memory. This tool has been designed to debug 'kernels' executing on GPUs. It provides a variety of ways to display GPU context such as GPU threads, blocks, warps and lanes. Filters based on user defined expressions aid in locating and isolating massively parallel debugging issues.

A typical debugging session can be seen here in Figure NView-1.
Note that all the information areas in this view are independent windows that can be moved, re-sized, detached from the main window or removed entirely. At the top of the window is a collection of icons that represent some of the most often utilized debugger commands.

These include control icons like `resume`, `run to here`, `next`, `step` etc., as well as quick access to eventpoint definitions like `breakpoint`, `patchpoint`, `monitorpoint` and others. Finally, there are icons for displaying expressions; control of source code display -- source, assembly or both; and control over how threads should execute (all or singly).

Below the icons in the work area we see a number of sub windows, each dedicated to a type of debugger information. On the left, we see the source code window that has been configured to show the C source code only; no assembler. In the upper right, we have the eventpoints sub-window which will show us any eventpoints that we have set and their status. In this case, we have set a breakpoint, a monitorpoint and a watchpoint (address trap). Below that is a tabbed collection of three sub windows: locals that will allow us to track stack variables in the present context; monitorpoints that will show us the variable values requested with our monitorpoint; and the context sub-window where we have access to context details of all active threads being debugged. Below the source window is the local shell where applications can be started as they normally would, by name or even via a shell script. Finally, at the bottom is the command line interface to NightView where any of the many dozens of available debugger commands can be invoked.

To demonstrate the basic mechanism of applying eventpoints, we highlight line 85 by left clicking on it. Next, we select the breakpoint dialogue at the top of the page, *Figure NView-2*. Notice the location of the breakpoint. A.c:85 is predefined as a result of highlighting that line. We can choose to make this a one-time breakpoint in the options section. We could have accomplished the same goal by highlighting line 85 and selecting our `run to here` icon. The bottom half of the dialogue allows us to create conditions, ignore counts and other metadata associated with the eventpoint. By stepping into the subroutine at line 85, we are presented with a short segment of code where we can demonstrate the use of other eventpoints.

*Figure NView-2*
After stepping into the Asub1() routine, clicking on line 21 and using the monitorpoint icon, we can create a tab that will capture the values of the two variables \textit{incr} & \textit{delayed} every time execution proceeds past line 21 (Figure NView-3).

The monitor tab in Figure NView-4 provides access to any program variable including stack variables, expressions, or even the results of function calls because the monitorpoint is placed at a specific point in the executing code.

We can also use the patch icon to affect how the code executes in a significant way.
Figure NView-5 shows the source code for a routine arbitrarily named patch(). You'll notice that this code references a variable found in the existing subroutine, delayed, as well as defines two new variables; newint and newfloat.

The patchpoint dialogue in Figure NView-6 is capable of supporting (multiple) single line patches and code skipping gotos. In this example, we're implementing a more involved patch by simply putting the new code in a subroutine, compiling it to object, loading that object into the debugger, and finally choosing the point at which you want to execute those new lines and using a patchpoint to call that new routine.

Debugging dynamic memory problems can be difficult and extremely time-consuming. The word heap refers to a collection of allocated and freed memory typically controlled by the malloc() and free() utilities in the C language. NightView provides the ability to monitor and detect memory allocations, frees, and sets of user errors without requiring a non-standard allocator to be compiled or linked into your program. One advantage of this is that when you switch to a debugging allocator, the way blocks are often allocated and freed changes, hiding the very bugs you're trying to find. NightView offers a variety of settings and debugging levels that are useful in catching common heap-related errors.
In order to debug a heap problem, we need only use the **Process** pulldown and select **Debug Heap** prior to executing the first line of code in the program. Doing so produces the dialogue shown in **Figure NView-7** below.

![Figure NView-7](image)

From here, we can set a number of specifiers or simply choose a debugging level which will specify a default set for that level. Suspecting a possible issue with the heap in A.c line 38, we will put a heappoint there and run the program.

The messages sub window, **Figure NView-8**, indicates that a heap error has occurred.

![Figure NView-8](image)

The Locals tab, **Figure NView-9**, shows us that mptr has incurred a heap error, and provides error details. All eventpoints used in a session will be automatically retained when the restart icon is used to restart the program being debugged.
NightView is capable of debugging both a main program that uses CUDA API calls to execute a 'kernel' on a GPU, and the 'kernel' code itself, executing on the GPU.

Figure NView-9 shows such a debugging session. In the left source window, we see the kernel code which will be called below in the main program. A breakpoint has been placed in the kernel code and the debugger has stopped at that line. On the right are the CUDA-specific sub windows. CUDA code is arranged into contexts that are presented as siblings of host threads.

Figure NView-10

Physically, they are organized onto the computational resources of the CUDA devices which are broken down into devices, symmetric processors, warps and lanes. Because the number of CUDA threads can be very high, they are presented as parts of either a physical or logical hierarchy to ease understanding. In the upper right is a display showing the active Block and lanes.

Below that is a Coordinate tool bar which will either display the current Physical and Logical location of context or can be set to search for a particular location. Below that is the Warp Locals display which shows the local variables from each active thread.

In summary, NightView is a powerful debugger that can be used from the command line or a
GUI. It supports programs written in multiple languages running across multiple CPUs and even across multiple systems via remote shell dialogues. NightView's special relationship with RedHawk Linux allows it to evaluate eventpoint conditions at execution speed, debug Heap memory with no special compilation or linking and show the contexts of CUDA kernels running on GPUs.

**NightTrace**

A vast majority of Concurrent-Real-Time customers are developing and running applications that have firm and unyielding requirements for high determinism and low latency. Often, the application uses interprocess communication calls, calls to physical I/O, external interrupts and other mechanisms which complicate timing. Since most applications involve multiple processes and processes with multiple threads running across multiple CPUs, inconsistencies in timing are very difficult to track down and eliminate by examining source code.

NightTrace is a tool designed to function as a software logic analyzer. It dynamically captures tracepoints in user and kernel code while the application is running and displays the data on a sub-microsecond timeline. RedHawk Linux supplies a trace-enabled kernel which is populated with trace eventpoints in all interesting areas like context switches, CPU exceptions, system calls and interrupts. NightTrace supplies an API that developers can use to populate their own source code (C/C++, FORTRAN, Ada, Java) with tracepoints in or around anything of interest to them.

There are also linking options that can automatically populate user code with tracepoints around all calls. The user can edit the parameters associated with the daemons that NightTrace uses to collect this user and kernel data. The user can stream data to the tool for real-time monitoring or have the data written to disk for post run analysis.

Since the amount of data collected can grow quickly, the user has a number of choices to manage the data. One option is to run NightTrace from a non-real-time system, target the runtime system, and collect and store the data on the non-real-time system. Another option is to collect the data on the target system but limit the size of the data file. This file can wrap, meaning newest data overwrites oldest data.

NightTrace also supports the concept of triggers so a user can let a trace run, waiting for a set of conditions to occur that will stop the trace and save the trace data around the triggering condition. For a look at the captured data, refer to *Figure NTrace-1* which is a default, kernel data display page.
Figure NTrace-1

This page is created automatically upon collection of kernel data and requires no configuration by the user. The top portion of the display is an Events list where every kernel event captured in the trace is listed. They are listed by offset — offset 0 being the first event captured in the trace. Additional columns list the specifics of the event (type, time and place it occurred, etc.)

Below that is a timeline where each event is drawn on the logical CPU where it occurred. You will notice objects that appear to have width on the timeline. These objects are called states and are defined as beginning with an event of one type or condition and ending with an event of another type or condition. To the left of this timeline is a series of text boxes that dynamically update descriptions of what the cursor-positioned timeline is sitting on. This page is created by the NightTrace tool and automatically creates a timeline display for each of the logical CPUs configured when the data was collected. For example, if hyper-threading were turned off on each core of this quad core system, then displays for CPUs 0-3 would be created. As hyper-threading is enabled for each of these cores, we see 8 logical CPUs, and timelines for CPUs 0-7. Reading the kernel timeline is not difficult. The text and data areas for a given CPU are identical to the other CPU displays. Each has six sections. From the top down are graphs for user events, interrupts, CPU exceptions, system calls and PID / TID graphs, and a list of all kernel events is at the bottom. The middle four are displaying states, the top and bottom sections are displaying each individual user or kernel event.
Continue to refer to Figure NTrace-1 as we look at some typical data across CPUs 0 through 4. Our cursor placed timeline is positioned near the left edge of the data. The groups of data representing the activity on cpus 1, 2 and 3 shows an (fbs) interrupt being handled.

We can see that a number of kernel events seen in the bottom graph for those cpus are occurring during the handling of this interrupt. The text box on the left shows us that the event just to the left of our timeline is a SCHED_WAKEUP event. This makes perfect sense because that fbs interrupt is being used by our NightSim tool to wake up the processes we see running on CPUs 1, 2 and 3.

![NTrace-1 diagram](image)

**Figure NTrace-2**

Moving to NTrace-2, having moved our timeline slightly right, notice that on CPU 0, no process is currently running, and so we see the idle process showing up in the PID/TID text box. On CPUs 1-3, however, there are processes running, and we can see they are all separate threads of a process called wd. This view also lets us see a vertical line at the very beginning and at the very end of each process. These are context switches and they delineate when a CPU goes from running the idle process to another process and back, or when a CPU goes from running one process to a different process as we see happening on CPUs 0, 2 and 3. All of the processes running on CPUs 1, 2 and 3 were blocked in the system call, fbsched. This is our custom system call, accessed by the API supplied fbswait(3) and used to schedule processes with our NightSim tool. If we move our timeline to the end of
any of those processes, we see that each process has completed its run time loop and is blocking on the `fb sched` call to wait for NightSim to wake it up again.

Refer now to *Figure NTrace-3*. This view shows us a tab that has been customized. A custom kernel page was created that only shows the CPUs interesting to us, CPUs 0-3. Above that kernel data, a series of five state graphs have been created to show us when we are between event points that we have placed strategically in our own source code.

In this case, we have placed event points around some subroutines in the process `wd:thread_sib_A` running on CPU 2, and we have used event points in two different threads, `thread_sib_A` and `thread_sib_B` to time A waking B with a semaphore (`futex`). While approximate timings can be gleaned from the two time scales at the bottom of the display, the top one being for the visible data set and the bottom representing the entire trace, precise times can be viewed by zooming in with the hover time feature of the mouse. Looking at the lower left corner of the display we can see that “Hover time from the current timeline = .000289738” which means that at the moment of this screen capture, the cursor was hovering 289.7 microseconds to the right of the timeline. Quick and accurate timings can be obtained using this method without resorting to consulting the precise time stamps associated with events up in the *Events* display.

One last thing to note in *Figure NTrace-3*, between the data graphs for CPUs 0 and 1, we see three data boxes labeled Pressure=, Alpha=, and Gamma=. These data box values change as the timeline is moved around the data. They reflect the last recorded value of the variable at that location. These values were recorded using a variation of the trace event points used
above to show when we were at the beginning or end of a particular piece of code. These events were logged as floating point values with the API call trace_event_two_flt(). Other API calls allow you to record integers, arrays, structures, and strings. NightTrace provides a variety of Profile templates to help the user locate areas of interest within a data set. Profiles are simply a set of events with conditions.

Figure NTrace-4 shows a search dialogue. In this case the user simply chose to look for “Interrupt Enter Events” from a pulldown menu of Profile templates and then chose the “Nvidia” interrupt from a resulting pulldown menu. The rest of the fields were populated automatically; however, the user is free to modify those fields as desired to narrow the search parameters.

Another very useful dialogue is the Summary tool. Using a menu similar to the one used to configure our search, we can ask NightTrace for a Summary of a particular condition or state, or perhaps a particular Profile that we have defined previously.

![Table of values](image)

**Figure NTrace-5** is the result of a summary of a profile we have defined to match our subroutine, Sub_A2. Based on our criterion, the summary tool has scanned our whole trace and represented each occurrence of Sub_A2 as a vertical red line. The spacing of the red lines across the timeline is an indicator of the frequency at which the routine is called. The height of the red line is an indication of the time spent running on the CPU. On the left side of Figure NTrace-5 we see that our routine Sub_A2 running very deterministically. The red lines are evenly spaced and all the same height. It is very clear though, that at about the five second point in the trace, an anomaly has occurred. Our Sub_A2 has run long and missed a frame.
By simply selecting the Custom kernel Timeline tab, Figure NTrace-6, we change views so that we are looking at our kernel and user data at the same place as our summary.
Then by zooming into the area of interest, Figure NTrace-7, we can diagnose the problem by looking at the kernel data.

NightProbe Timing isn't the only concern when developing software for RedHawk systems. Sometimes the user needs visibility into a running program to see if the software is functioning as designed. The user often has requirements to modify program variables like flags or gains while evaluating software logic or performance. Many applications also call for recording data synchronously or asynchronously and storing it for later analysis. Other times it is helpful to monitor or visualize data via spreadsheets or graphs.

NightProbe was designed to do all of those things and more utilizing a non-intrusive technique of mapping the memory of selected variables directly. NightProbe can sample and modify data without interrupting or otherwise affecting the target resource. By resource, we mean that NightProbe can access anything with an address — a running program, areas of shared memory or mapped memory, and even memory pools on a PCI card.
In Figure NProbe-1 below, we have configured NightProbe to look into the address space of a program called "wd".

Once the resource is chosen, a 'Browse' tab is provided that lets us explore the various scopes of this particular process.

![NightProbe interface screenshot]

Figure NProbe-1

Note that we have opened subroutines called Asub1, Asub3 and a structure called rqtp. NightProbe gives us read/write access to any of the resulting displayed variables. To be clear, we are seeing variables with fixed addresses — variables that have been declared as either **global**, or **static**. However, you can indirect through pointer variables in the Browse panel, freezing the base address at that time – thus even dynamically allocated variables can be probed if their base address is static or global. Our program only needed to be compiled with symbols (debug information) to make this access possible. In addition to this browsing page, there are four other data display options.

Looking just under and to the right of the Help pulldown, we see icons for scrolled text view, scrolled table view, spreadsheet view and graph. To be recorded in any way, or used in one of these displays, a variable must be identified as one of interest. We can see in NProbe-1 that **delta** in Asub1 and **gamma** in Ausb3 have been double clicked; their icons turn to red, and therefore made available for recording, graphing etc.
Figure NProbe-2 below shows the graphing tab when delta and gamma have been plotted.

![Figure NProbe-2](image1)

Figure NProbe-2

Figure NProbe-3 below shows a spreadsheet where a number of variables have been placed. Cells can be configured to change color based on settable upper or lower thresholds and of course can be easily saved to be brought up again in subsequent sessions. To the right of the spreadsheet panel, we can see that some data recorded to disk has been brought up using a scrolled table.

![Figure NProbe-3](image2)

Figure NProbe-3
NightSim

Most of the applications created by RedHawk users can be described as either event-driven or frequency based. An event-driven application is often of the producer/consumer type, where the application waits, blocked for data to become available. This data may come over a network, reflective memory, from another process or some other way; and the data uses one of the inter-process communication mechanisms (IPCs) or an interrupt to announce their availability.

In the case of a frequency-based application, one or more processes represent some physical system and its behavior over time. Such a process will calculate the changes of the system for a specific time step and will internally integrate that time step so that subsequent cycles of the process will represent a realistic prediction or simulation of that system over time.

If the rate at which we cycle the model exactly matches our integration time step, then we would be said to be running in 'real-time'. In other words, if a particular model calculated the changes that would occur in a physical system over the course of one millisecond, and we were able to schedule that process to run and complete once per millisecond, then that model would be running in real-time.

RedHawk customers often have multiple or even dozens of processes that model complex systems. These models all need to run at some frequency, and often different models need to run at different frequencies.

Prior to NightSim, a user was forced to architect their own 'Executive' program that would have to locate and control a timing source on the computer, and program that timing source to wake up processes at the desired frequency.

Over time, if newer OS versions were installed or there were changes to the physical architecture used by the project, a certain amount of porting of the Executive would be required to keep it in working order. There is also personal familiarity with software to consider as team members move to different projects or leave the organization.

By using NightSim, the problem boils down to one system call \texttt{fbswait(3)} and inclusion of two libraries.

\textit{Figure NSim-1} below shows the only code modifications required to control program execution with NightSim. In the program mymain.c, we have included the file /usr/include/fbsched.h, and at the top of our runtime loop we have added the \texttt{fbswait(3)} system call.
In the Makefile we have linked the program with two Concurrent Real-Time libraries: ccur_rt and ccur_fbsched. With this small addition to the source code, a process, or a process thread can be put under the control of Concurrent Real-Time's Frequency Based Scheduler (FBS) and woken up at regular, programmable intervals.

Using the NightSim Scheduler tab, Figure NSim-2, we begin in the Definition box where we select some of the basic parameters of our scheduler such as an arbitrary key and the cycle/frame structure we want. To the right, in the Interrupt box, we will select one of the many interrupt sources provided by the Real-Time Clock and Interrupt Module (RCIM), and in the case of a clock, set the clock to tick at our base rate. With the scheduler set up, we can then proceed to schedule one or more processes on it.

Figure NSim-1

Figure NSim-2
Figure NSim-3 is the Add process dialogue, FBS Schedule tab where we are selecting a process and making decisions about what cycle it will start on and with what period it will run. We are also specifying what to do in case this process overruns its frame.

Figure NSim-4 is the Runtime Properties tab where scheduling class, priority and CPU affinity is set.

Finally, Figure NSim-5 (below) is the I/O and Debug tab where debug sessions can be requested and standard in and out are managed. With the process scheduling parameters in place, running the process is a simple matter of returning to the Scheduler Control tab, Figure NSim-2 and pressing the Set up button. At that point, any and all processes on the scheduler will begin running. They will begin at the 'main' entry point and proceed down through any initialization code that may exist until they reach the fbswait(3) call where they will block.
When the scheduler has been created, connected to an interrupt source and all processes are scheduled, the Start button becomes enabled, and you are free to start the processes cycling. From that point on, cycling can be controlled with the Pause and Resume buttons.

With the processes cycling, you will find it useful to monitor their performance using the Monitor tab, Figure NSim-6. Each scheduled process is listed along with various performance metrics such as percent of CPU used and average execution time in microseconds.

The fields shown are default choices. The user can configure the tool to show these and other additional statistics. The graph at the bottom of the display provides a historical look at process timing, including indications where a process or thread overran its scheduled time. Once a scheduler is configured, it can be saved and reopened at will through NightSim or the user can choose to save the configuration to a script file that uses the FBS' rtcp scripting language.

Figure NSim-5

Figure NSim-6
NightTune

Optimizing the performance of an application goes beyond smart coding practices. The best written code can run poorly on a system when no consideration has been given to the interaction of the running program and the computer systems’ resources. To maximize performance and determinism and minimize latencies, a user should be aware of how the application is utilizing system resources and have the tools necessary to monitor and modify that interaction.

One of the fundamental methods of guaranteeing application performance on a RedHawk Linux system is the concept of protecting and reserving some system resources for the dedicated use of the user’s application. This concept, called shielding, lets the user control what resources should be used in service of the Linux operating system, and which will be reserved for the user’s application. Having shielded resources for their own use, the user also needs a means by which to schedule their application to take advantage of those shielded resources.

Tuning topics are explored briefly in the white paper, Tuning a RedHawk Linux System, and thoroughly in the RedHawk Linux Users Guide. Most of those techniques can be applied using the NightTune GUI tool. NightTune is capable of displaying many system metrics and offers mechanisms for control where applicable.

*Figure NTune-1* is the default page provided by NightTune and contains three information windows.
On the left, is the Process List which displays a “by user” listing of all running processes. In this list are metrics that would be provided by the ls(1), or top(1) commands, as well as specifics about how and where the process is running. Note the highlighted area where the threads of the process wd are being displayed. Not shown are a long list of options and controls revealed by right clicking on any of the processes or process threads. These controls provide options for how much and what kind of data to display with each process and offer dialogues used to modify scheduling and CPU-bias parameters and pages of in-depth process information.

Figure NTune-2 is the Process Details page for the wd process. The visible tab shows the Memory Maps for the process, but other tabs show additional per process information like signals used, environment variables, capabilities, etc. In the upper right of Figure NTune-1 is the CPU Shielding and Binding view which helps the user visualize the machine’s CPU architecture; how many separate sockets, cores per socket, any hyper-threading that is enabled, and how the resulting logical CPU set is shielded from interrupts in general, core specific interrupts like the local timer interrupt, and processes in general.

Right clicking in this window will provide a control dialogue which will enable the user to modify the state of shielding dynamically. Finally, in the bottom right is a CPU Usage view. As shown by the list of tabs in Figure NTune-1, there are many system metrics that can be monitored from NightTune, but for the purposes of this paper, we will only examine a few of them in any detail.
Figure NTune-3 is the Interrupt Detail Activity tab from a machine with four logical CPUs. By right clicking an IRQ name, an affinity control dialogue is provided so the user can specify where specific IRQs should be handled. Note that the eth0 interrupt has been assigned to CPU 1, and bound to that CPU (the chain link icon). CPU 1 may be shielded from all other interrupts, guaranteeing that when an eth0 interrupt occurs, it will be handled without any delay.

Line and bar graph displays are available for most metrics as well, giving you a graphical view as well as history of the metrics over time.

The CUDA and CUDA Configuration tabs, Figure NTune-4, are provided for those utilizing available GPUs for application programming using the CUDA API. NightTune offers a GUI representation of the parallel computing capabilities of each CUDA device on the system and provides detailed device information including GPU configuration and capabilities, block configuration, and memory capacity.
NightTune provides dynamic usage data including GPU usage, memory activity and usage, as well as power, fan speed and temperature metrics. A user can change the configuration settings as needed. Having used NightTune to analyze and modify the state of system shielding and the possible vectoring of processes or threads to specific CPUs, you may want to save that work so that it can be recreated at the next system boot. From the *File* pulldown, select **Save System State**.

The resulting dialogue, *Figure NTune-5*, lets you selectively choose or ignore modifications for saving in an ASCII file. That file can be subsequently loaded in the future so all saved modifications are recreated or NightTune can be set up to run as a daemon at boot time and implement those changes as they become possible.

![Figure NTune-5](image-url)
Conclusion

NightStar is a set of five powerful tools for developing time-critical CPU and GPU applications on RedHawk Linux. These tools run with minimal intrusion, thus preserving application execution behavior and determinism. NightStar handles the complexities of multiple processors and cores, multitask interaction and multi-threading, and the tools are constantly evolving to address customer needs. With NightStar, users can quickly and easily debug, monitor, schedule, analyze and tune applications in real-time which reduces test time, lowers development costs and increases productivity.

About Concurrent Real-Time

Headquartered in Pompano Beach, FL, Concurrent Real-Time is the industry’s foremost provider of high-performance real-time computer systems, solutions and software for commercial and government markets worldwide. Its real-time Linux solutions deliver hard real-time performance in support of the world’s most sophisticated hardware in-the-loop and man-in-the-loop simulation, high-speed data acquisition, process control and low-latency transaction processing applications. With over 50 years of experience in real-time solutions, Concurrent Real-Time provides sales and support from offices throughout North America, Europe and Asia.

©2020 Concurrent Real-Time, Inc. Concurrent Real-Time and its logo are registered trademarks of Concurrent Real-Time. All other Concurrent Real-Time product names are trademarks of Concurrent Real-Time, while all other product names are trademarks or registered trademarks of their respective owners. Linux® is used pursuant to a sublicense from the Linux Mark Institute.

More Information

www.concurrent-rt.com
info@concurrent-rt.com
(800) 666.4544 or (954) 974-1700