KernInst User's Manual

Version 2.0.1

Ariel Tamches, Barton P. Miller, Alexander Mirgorodskii, Michael Brim, Igor Grobman
University of Wisconsin-Madison

tamches@veritas.com
{bart, mirg, mjbrim, igor}@cs.wisc.edu
Bug reports to mjbrim@cs.wisc.edu

1 Overview

This section presents a quick overview of the KernInst software.

1.1 KernInst System Components

KernInst has several components:

- High-level (presumably with a GUI) clients that interact with the user and talk to the lower-level component (*kerninstd*) to perform kernel instrumentation. These clients are linked to the KerninstAPI library (*libkerninstapi*), an application programming interface that abstracts all interaction with *kerninstd* and provides an easy to use set of C++ classes for writing high-level instrumentation code. Presently, there is one such client: *kperfmon* ("kernel performance monitor").
- *kerninstd*, the KernInst daemon process. Kerninstd is the instrumentation server at the heart of KernInst. GUI processes, such as *kperfmon*, that wish to insert instrumentation code into the kernel will ask kerninstd (via a remote procedure call interface) to carry out an instrumentation request. Note that although kerninstd is "just" a user process, it must be run by root.
- */dev/kerninst*, the KernInst pseudo-device driver, which can be considered a part of kerninstd. Kerninstd will invoke the driver's assistance for certain operations that require running from within the kernel's address space.

1.2 Platform requirements

Presently, the kerninst components run on the following platforms:

- Sun UltraSparc-I, II or III systems[1] running Solaris 7 or 8 (both 32- and 64-bit)
- Intel Pentium II, III, or 4 and AMD Athlon systems running Linux 2.4.x
- Uni- or multiprocessors

Kerninstd and the driver are installed and run on the machine whose kernel you wish to instrument. kperfmon can be run on a different machine; in fact, this is encouraged, because it lightens the load on the machine being instrumented.

Since they're "just" GUI's, higher-level clients of kerninstd such as kperfmon can (in theory) be compiled to run on any platform. Presently, however, kperfmon is only distributed as platform-specific binaries (i.e., kperfmon for IA-32 systems is distributed as an IA-32 binary).

1. Running on an older SuperSparc cpu will not work, since both kerninstd and kperfmon access hardware registers, such as the cycle counter, that are only present on UltraSparcs.

2 Installation

This section describes installation of the various KernInst components.

2.1 Fetch a distribution

The KernInst version 2.0.1 release provides a source distribution, kerninst-2.0.1.src.tar.gz, and two binary distributions: kerninst-2.0.1-sparcv9.tar.gz and kerninst-2.0.1-ia32.tar.gz. A third binary distribution, kerninst-2.0.1-sparc.tar.gz, will be made available upon request. Of the "sparc" distributions, the former should be used on machines running the 64-bit version of Solaris, the latter on 32-bit Solaris machines. To determine which file you need to download, run the "isainfo -n" command on the target machine (the machine on which you plan to run kerninstd). If the output mentions "sparcv9" use the sparcv9 tar file, otherwise, use the sparc tar file. The "ia32" distribution can be used on any IA-32 machine running Linux 2.4.x.

You can run kperfmon from either of the "sparc" distributions on any Solaris platform: kperfmon from the 64-bit distribution can run on a 32-bit machine and vice versa. However, make sure that all software components that you use (kperfmon, kerninstd and /dev/kerninst) are from the same distribution. For example, 64-bit kerninstd will not work with the 32-bit kperfmon.

In the following subsections, we use the variable $PLATFORM to refer to a specific architecture and operating system combination. Example platforms are "sparcv9-sun-solaris2.8" (for 64-bit Solaris 8), "sparc-sun-solaris2.7" (for 32-bit Solaris 7), and "i386-unknown-linux2.4" (for IA-32 Linux 2.4). To install KernInst on a specific platform, follow the directions below, but substitute the appropriate platform string for the $PLATFORM variable.

After the usual "tar xzvf kerninst-2.0.1-<arch>.tar.gz" unpacking sequence, you should see the kerninst-2.0.1 directory with ten sub-directories: core, docs (containing this file), kerninstapi, kerninstd, kerninstdriver, kperfmon, scripts, tcl, util, and visiClients. Steps for installing these components are enumerated below.

2.2 Install the device driver

Since installing the device driver, as well as running kerninstd, may print some debug messages to the console, it is recommended that *xconsole* be already running, to capture the output in an easy to read format.

- Get a root shell.
- Change to the kerninstdriver/$PLATFORM subdirectory.
- Type "./install_script" to perform the installation.

If for any reason this script fails, you should be able to install the driver manually; examining the install_script file will show that it is pretty trivial.[2] Some debugging messages in the console are expected.

2. Of course, if manual installation is required, please file a bug report to mjbrim@cs.wisc.edu.

IA-32/LINUX NOTE: Due to the fact that there are numerous 2.4 kernels that may be running at various user sites, we do not provide precompiled binaries for the kerninst driver. Instead, we provide the source code and build files, allowing users to build the driver for their specific kernel. Building the driver is included as a step in the 'install_script'. If this step fails, it is likely due to an incorrect variable setting in the 'Makefile', and you should edit the file to match your environment. The most common problem in building the driver is a missing 'kallsyms.h' include file. This file is part of the modutils package, available from http://www.kernel.org/pub/linux/utils/kernel/modutils. If you encounter this error, please download the latest modutils package and set the KALLSYMS_INCLUDE variable in the 'Makefile' to the path to 'modutils-<version>/include'.

2.3 Run kerninstd

IA-32/LINUX NOTE: Please see the README.Linux file for additional information on how to provide better kernel symbol information to kerninstd before running it.

After installing the device driver successfully, you should next run kerninstd:

- Get a root shell
- Change to the kerninstd/$PLATFORM subdirectory
- Type "./kerninstd" to run it

Although kerninstd does not contain a GUI, startup is a significant task (involving, among other things, reading the kernel's symbol table, parsing its code into basic blocks, creating a control-flow graph, and performing live register analysis).

If anything goes wrong, such as an assertion failure, it is likely to be in this stage. If this occurs, please re-run kerninstd with one flag: "-v" (verbose) and email the output[3] to mjbrim@cs.wisc.edu. Furthermore, if you have built your own binaries from the source distribution, please re-run kerninstd from within the gdb debugger (kerninstd is presently compiled with the -g flag and no optimization, to assist this) and send a stack trace (i.e., type "where" from within gdb after the crash[4]).

If everything goes right, the startup phase will take about a minute.[5] There will be a line such as "The chosen accept port number is 41647". Remember the port number; clients (e.g., kperfmon) need to know it in order to connect to kerninstd.

To exit kerninstd cleanly, type Ctrl-C (the signal will be caught and cleanup will be done).

---

3. The output from running kerninstd with the '-v' option is quite large. Please compress the output file with gzip or bzip before emailing.
4. Optionally, to prevent much superfluous gdb output, type "set print static-members off" within gdb before getting the stack trace.
5. It's much less when kerninstd is compiled with optimization (about twenty seconds).

2.4 Run kperfmon

Preferably on another machine (to lighten the load on the kernel being measured), log in as a regular (not root) user and run kperfmon:

- Make sure that the binaries for the visualization processes (rthist, barChart, and tableVisi) are in the current search path, so kperfmon will be able to find them. Double-check that they can be found by typing "which rthist" (for example) before running kperfmon itself.
- Make sure the environment variables TK_LIBRARY and TCL_LIBRARY are set. See the following paragraph for more info.
- Change to the "kperfmon/$PLATFORM" directory.
- Type "./kperfmon <kerninstd-machine-name> <port-num>", where <port-num> is the "chosen accept port number" as reported by kerninstd's startup.

Note that kperfmon uses the Tcl and Tk toolkits for GUI purposes. kperfmon should have the necessary libtcl and libtk libraries statically linked, so you do not need to install them.[6] However, tcl/tk binaries are not stand-alone; certain ".tcl" files are needed for tcl/tk to start up, and there must be environment variables TCL_LIBRARY and TK_LIBRARY which contain the directory names of these startup files. If you already have tcl/tk installed on your system, then these variables should already be set and you can ignore the rest of this paragraph. If not, we have shipped the necessary (freeware) files. Set the TCL_LIBRARY environment variable to 'kerninst-2.0.1/tcl/tcl8.4' (use the full path name), and set TK_LIBRARY to 'kerninst-2.0.1/tcl/tk8.4' (again, using the full path name).

If all goes well, kperfmon will establish a connection to kerninstd, retrieve the analyzed kernel's state, and put up its GUI (this takes about twenty seconds).

With this step successfully completed, installation is done! Assuming for the moment that the user is interested in running existing kerninstd clients such as kperfmon (as opposed to writing new ones), there is no need to explain kerninstd or its driver any further, and all that is left to do is discuss the features of kperfmon.

6. We are presently using tcl and tk version 8.4.5. If your site already has tcl/tk installed, you may prefer a dynamically linked (and thus smaller) kperfmon binary. Let us know if this is the case.

3 Quick Kernel Instrumentation Tutorial

The best way to understand kperfmon is through an example. In this example, we will count the number of procedure calls being made to the function “kmem_alloc” (which resides in the module “genunix”) on a Sparc/Solaris machine. Instrumenting a particular piece of *kernel code* (presently, a function or basic block) using a certain performance *metric* is done with the following steps:

- Choose one or more performance metrics by single-clicking from among the various “Metrics” checkbuttons in the upper-left corner of the kperfmon window. For this example, we choose “entries to”.

Choose the code that you want to instrument by navigating among the “Kernel Code” portion of the kperfmon window. Note from Figure 1 that “Code” has a small triangle after its name. This indicates that there are more detailed code resources available for choosing, which can be revealed by double-clicking on it. (Single-clicking on “Code”

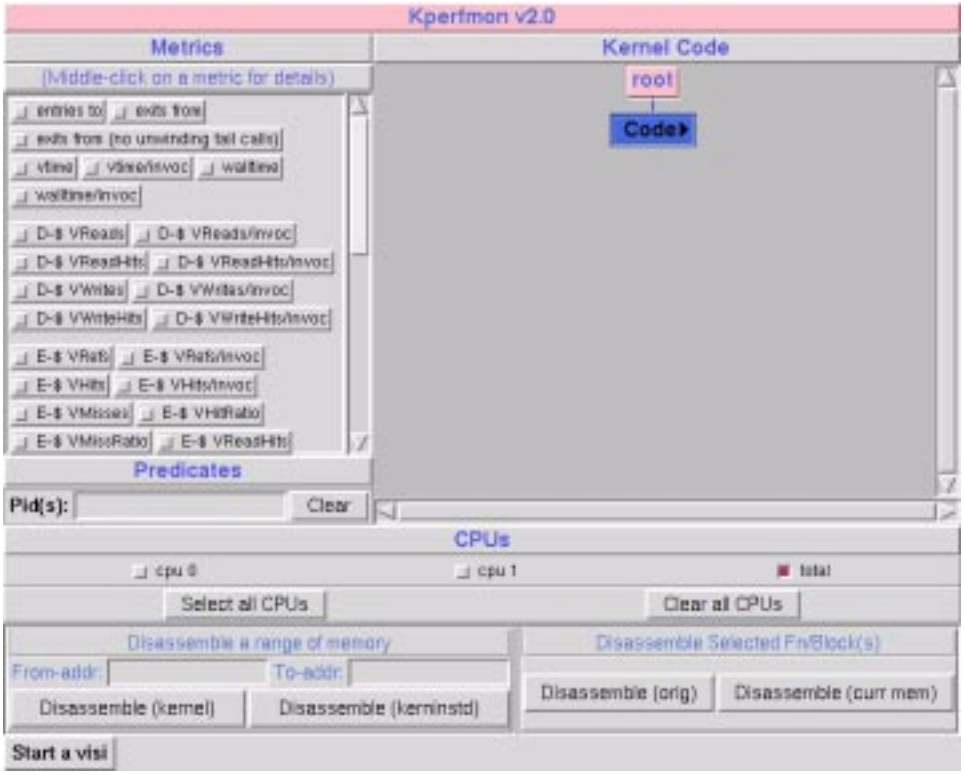

Figure 1: The kperfmon GUI on startup

would select it for instrumentation[7]; double-clicking on it will not select it. If you accidentally select a code resource, you can un-select it by single-clicking on it again.) Double-clicking on “Code” expands its children in the resource tree, namely, all of the kernel modules, as shown in Figure 2. To get to kmem_alloc, we next need to expand the mod-

7. We do not want to instrument “Code”, because (being the root of the code tree), this resource indicates “all kernel code”. In this example, we want to choose a specific kernel function.

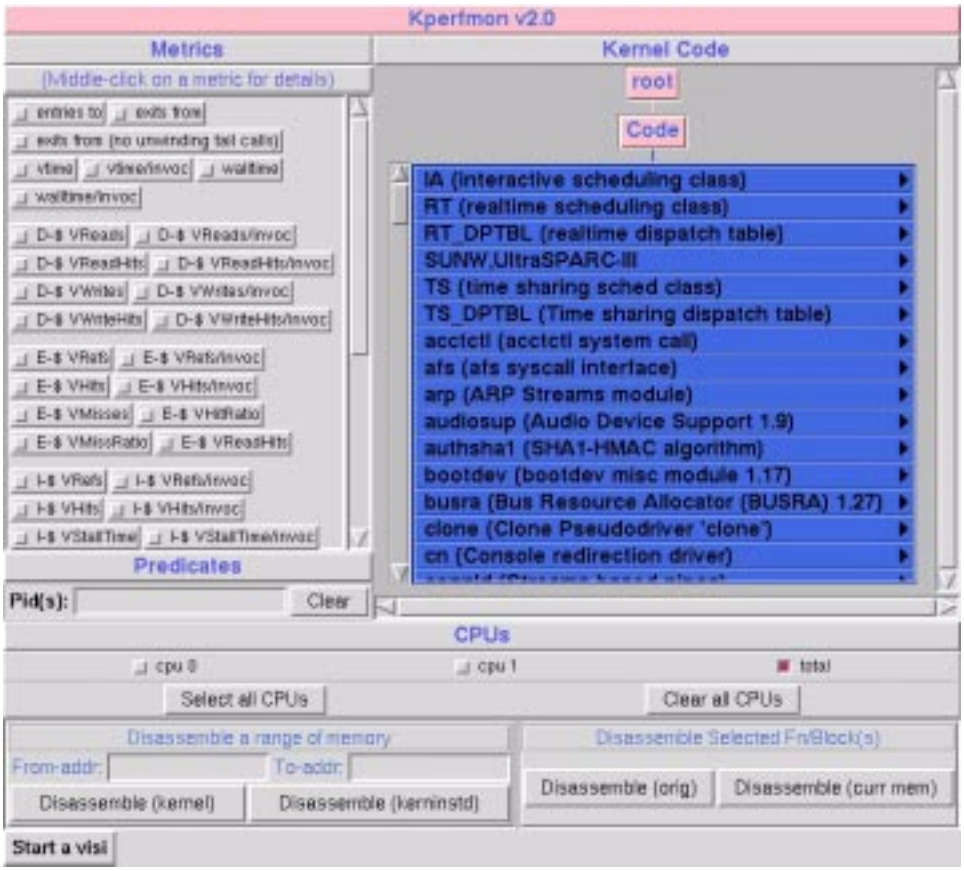

Figure 2: kperfmon after expanding "Code" to its children (all of the kernel modules)

ule "genunix" by double-clicking on it. After doing this, the functions of module genunix will be shown in their own list box. Next, we scroll in this list box (each list box has its own scroll bar) until we find the function kmem_alloc; we then select this function by single-clicking on it. The window will now appear as in Figure 3.

- If kerninstd runs on a multiprocessor system, you can select CPUs of interest by single-clicking on the corresponding check boxes in the CPU panel. For example, if you want to monitor the number of cache misses incurred by CPU0, you should select the "CPU0" checkbox. The "total" check box represents metric values for the whole system and is selected by default. Notice that if kerninstd runs on a uniprocessor, the CPU panel is not displayed.
- Start a "visi" (visualization process) from kperfmon by choosing an item from the "Start a visi" pull-down menu (near the bottom of the kperfmon window). The visualization process will be the GUI that displays the data collected by the instrumentation code. For this example, we choose the *histogram* visi from the pull-down menu, and its window (Figure 4) is shown on the screen.
- Next comes the step that will instrument the kernel! From the histogram's window, pull down the "Curve" menu and choose "Add..." This causes the histogram to tell kperfmon to (1) dynamically instrument the kernel for whatever metric/code

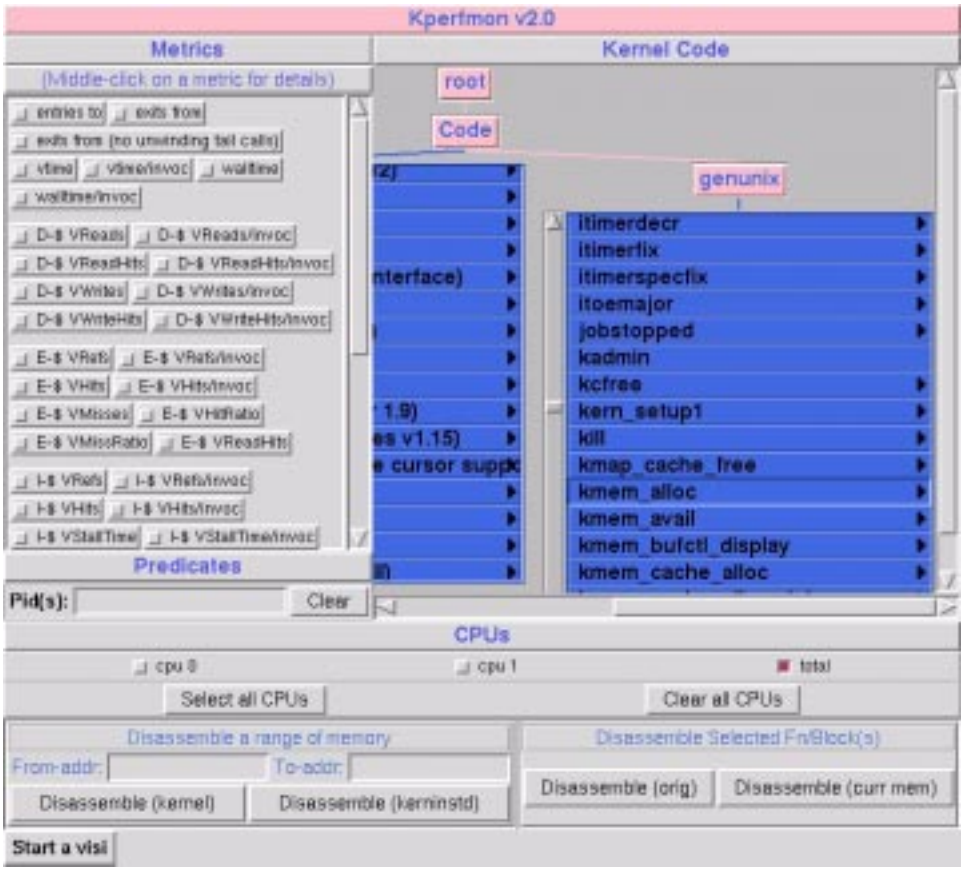

Figure 3: kperfmon window after expanding the "genunix" kernel module, scrolling its list box until we find the function "kmem_alloc", and selecting that function

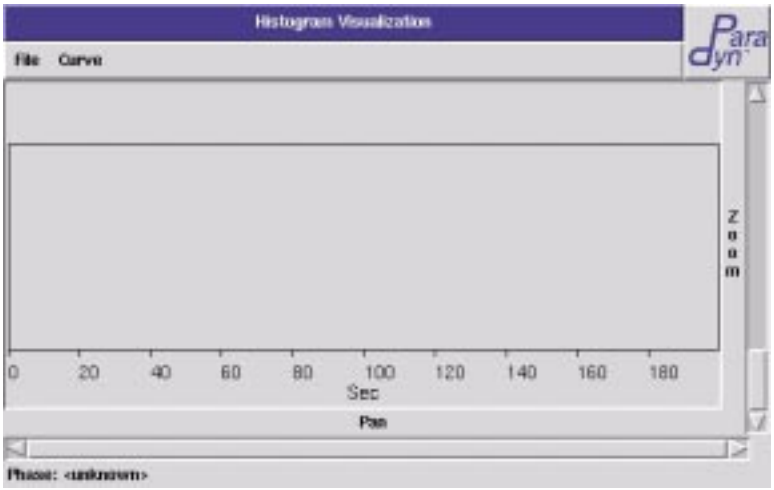

Figure 4: The histogram visi window

resource combinations are presently selected, and (2) to periodically sample the

data collected by instrumentation code and ship it to the histogram for GUI display. Collected data should immediately begin appearing in the histogram, as shown in Figure 5.

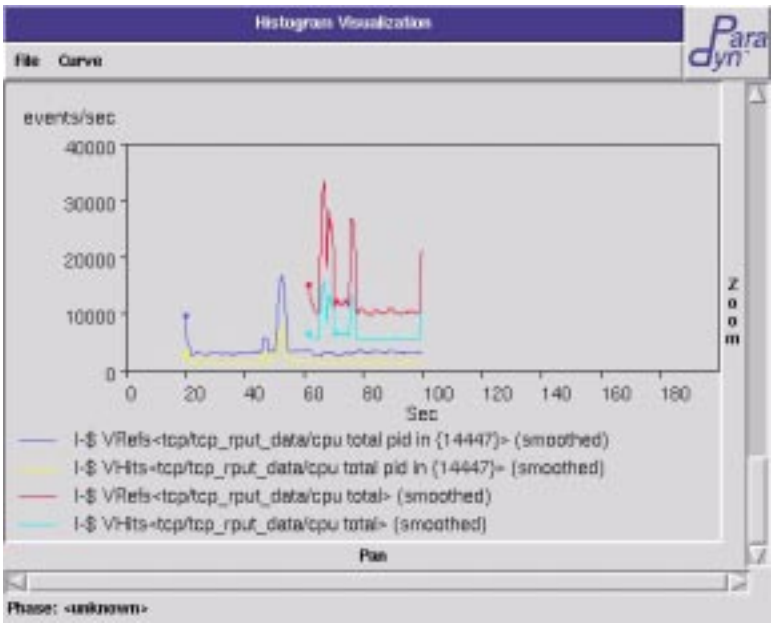

Figure 5: Histogram visi window after some data has been collected

- This completes our kernel instrumentation example. To stop collecting data (and more importantly, to un-instrument what had been put into the kernel), simply close the histogram visi.

Important note: In order to run one of the "visualization tools" such as the "histogram", you'll need to have the 'visiClients/$PLATFORM' directory in your path before running kperfmon; otherwise, kperfmon will not be able to find it (Section 2.4).

Like kperfmon, the visualization tools use tcl/tk.

Note that you can exit and re-run kperfmon multiple times without exiting and restarting kerninstd.

4 Kperfmon User's Guide

This section presents a more complete description of kperfmon than in the quick tutorial.

4.1 Making Instrumentation Requests (Selections)

An instrumentation request is a coupling of a metric (Section 4.1.1), a code resource (Section 4.1.2), a CPU resource (only on multiprocessors, see Section 4.1.3) and optionally extra predicates (Section 4.1.4).

At any given time, a large number of metric/code resource/cpu resource/predicate combinations can be selected. The set of selections is the cross product of all selected metrics, applied to all selected resources, with the optional predicate. For example, if two metrics are selected (say "entries to" and "walltime"), three code resources are selected (say, "kmem_alloc", "kmem_free", and "kmem_reap") and two CPUs are selected (say, "cpu3" and "cpu5"), then 2x3x2=12 instrumentation combinations have been selected. Choosing "add curve(s)" from the histogram visi will then add twelve curves to the histogram. With this is mind, it is also worthwhile to note that once selected, metrics and resources *stay selected* until they are explicitly unselected (by single-clicking on the given metric or resource a second time).

4.1.1 Metrics

As mentioned in the tutorial, each instrumentation request requires at least one metric to be selected. A given metric contains the logic for kernel instrumentation; when given a code resource, it generates that machine code snippet(s) to be inserted at specific kernel addresses.

To select a metric, single-click on it (its check button will then turn blue). Once selected, a metric stays selected until it is single-clicked on again to unselect it. Metric selection can also be done using kperfmon's tcl scripting interface; see Section 4.4.1.

Documentation on each metric is available on-line: clicking the middle mouse button over any metric will pop up a window that gives useful information on what it measures, what code resources it can be combined with, and (for the curious) an overview of the instrumentation code that it generates. An example is shown in Figure 6.

4.1.2 Resources: Kernel Code

kperfmon supports two classes of kernel resources: kernel code and CPUs[8]. These selections are chosen by manipulating the "Kernel Code" and "CPU" panels of the kperfmon GUI. In this section, we descibe the "Kernel Code" panel.

Single-clicking on a resource will select it; the resource will be drawn recessed in the GUI. Like a metric, a selected resource stays selected until explicitly unselected by single-clicking on it again.

8. We hope to support other kinds of resources in the future. Consider for example, being able to couple a metric "time blocked on" with a *mutex* (not code) resource "my_mutex".

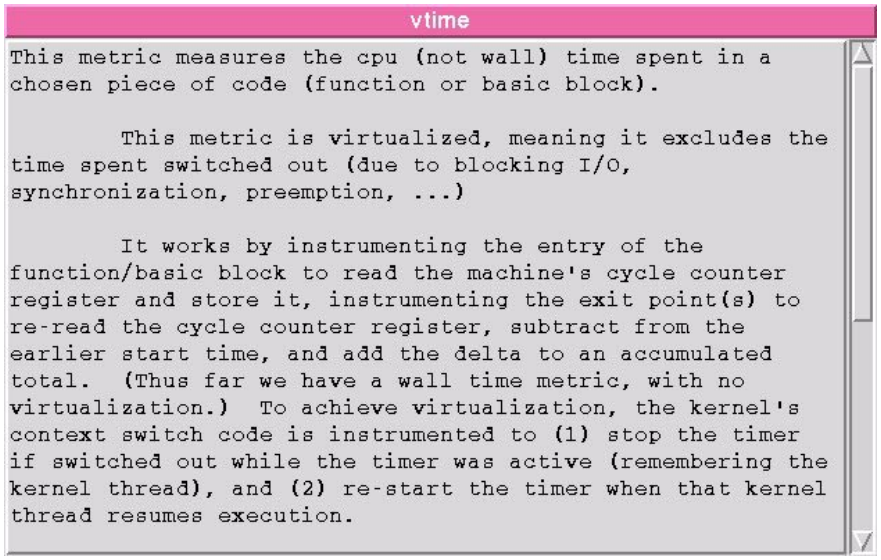

Figure 6: The description of the metric vtime

Since the number of code resources is overwhelming (on my machine, for example, there are 92 kernel modules, 9752 kernel functions, and 132250 basic blocks), it is challenging to present a GUI to allow for their selection. The solution in kperfmon organizes resources as a tree (root, modules, functions, basic blocks). Since there is not enough screen real estate to display 140000+ names, we hide entire sub-trees until they are *expanded* with a double-click. For example, initially (Figure 1) there is only a single code resource displayed ("Code"). The small triangle to the right of the name indicates that it can be expanded to show further resources (in this case, the next tree level being the various kernel modules) as in Figure 2. To maximize use of screen real estate, the various kernel modules are in a single, scrollable list box. Double-clicking on any one kernel module will then expand the next level of that module (only), which would be the module's individual functions, as in Figure 3. Functions that were successfully parsed into basic blocks by kerninstd can be expanded to show their basic blocks.

It is important to note that at least conceptually, a given metric can be coupled with *any* code resource to create instrumentation. For example, coupling the "entries to" metric with the resource "kmem_alloc" (of module "genunix") gives the number of calls to that function. Taking the next step, the same metric can be coupled with an individual basic block to give the number of executions of that block. And conceptually, "entries to" can be coupled with an entire kernel module (such as "genunix") to give the number of calls to *any* function in that module. However, in this case, the metric "entries to" will disallow this combination, because the amount of kernel instrumentation required would be overwhelming. The same applies to the base code resource, "Code", which would conceptually give the number of calls to *any* function in the kernel.

To review the hierarchy of code resources:

- *Root* or *Code*: conceptually means "the entire kernel". Presently, all kperfmon metrics refuse to generate code when presented with this resource, because the amount of instrumentation code to realize the desired measurement is prohibitive.

- Kernel Module: these appear in the list box underneath *Code*. Kperfmon receives the kernel modules from kerninstd on startup.[9] As with the "root" resource, all kperfmon metrics refuse to generate the (prohibitively large amount of code) when coupled with such a resource.
- Kernel Functions: a given module's functions will appear in a list box underneath the module. The functions will be shown sorted by kernel address.
- Basic blocks: a given function's basic blocks will appear in the list box underneath the function. The basic blocks will be shown sorted by kernel address. Unfortunately, the only identifying information for a given basic block is its starting address, so it is often difficult to know which basic block to choose! However, viewing a disassembly of a function or basic block (Section 4.3) can help.

A Note About Fonts Used

If the fonts used in the resource hierarchy do not look good on a given installation, they can be easily changed. Edit the file kperfmon.tcl (located in the same directory as the kperfmon executable), change one of the lines beginning "set helv14 ...", "set helv12 ...", "set helv10 ...", or "set helv9 ..." (these lines will be near the beginning of the file), and re-run kperfmon as before.

Resource Selection and Unselection Using the Keyboard

It will not take long to tire of scrolling through a module's list box to find a specific function. If you know the name of the kernel function, and the module that it belongs to, you can select and unselect resources from the keyboard using kperfmon's tcl scripting interface; see Section 4.4.2.

4.1.3 Resources: CPUs

If kerninstd is run on a multiprocessor, kperfmon displays the "CPUs" panel. This panel contains checkboxes for individual CPUs on the target system as well as the "total" checkbox which represents the system as a whole. Selecting an individual CPU defines a metric constrained to this CPU --- only events occurred on this CPU will be reported. Selecting the "total" checkbox defines a metric aggregated across all CPUs. For example, the "total" cpu resource applied to the "D-cache hits" metric will add up the individual numbers of cache hits on each CPU and report a single number for the whole system.

4.1.4 Predicates

The standard coupling of a metric with a code resource can be augmented with a *predicate*. Presently, there is a single supported predicate: process id(s). Consider the example of the metric "entries to" coupled with the code resource "kmem_alloc()", which will measure the number of calls made to kmem_alloc(). To measure the number of calls

9. Presently, if a kernel module is loaded into the kernel after starting up kerninstd, it will not be recognized. This feature may be added in the future. In the meantime, the desired effect can be achieved by restarting kerninstd, to force it to re-examine the contents of the kernel's runtime symbol table.

made to "kmem_alloc()" *just from a given process*, you can enter its pid in "pid(s)" entry widget of the kperfmon window. As always, it is up to the chosen metric / resource / predicate. Presently, every metric in kperfmon allows a pid to be entered.

Note also that more than one pid can be entered, so for example you can ask for the number of calls to a function made by process X *or* Y *or* Z. To do so, enter a list[10] of pids in the "pid(s)" entry widget.

For those interested in technical details, the current pid is determined in instrumentation code by examining the current kernel thread's process data structure. On Solaris, kernel threads not belonging to a user process are always assigned a pid of 0, which is a valid pid to enter in the "pid(s)" entry widget.

4.2 Visis

Visualization processes can be started in one of two ways: by selecting from the "start a visi" kperfmon pull-down menu (as introduced in the tutorial), or through kperfmon's tcl scripting interface (Section 4.4.3).

Presently, kperfmon is shipped with three visualization processes (all, as you will no doubt notice from the GUI logos, coming verbatim from KernInst's sister project, Paradyn):

- **Histogram**. This is probably the most useful visi. For each metric / resource / predicate selection made, a single curve is drawn showing the value (y-axis) over time (x-axis). It is the only visi process that shows how samples change over time. The downside is that the Histogram display can get cluttered after, say, 5 curves have been added. Furthermore, average or total values cannot be displayed using the Histogram.
- **Barchart**. This visi can display many metric / resource / predicate combinations efficiently. Each resource / predicate combination is drawn, top to bottom. To the right of a given resource / predicate combination will be one or more colored bars whose length indicates the value (x-axis); each color corresponds to a specific metric. The chief benefit of using the Barchart is that screen real estate is efficient, especially when many metrics are chosen. The downside is that you cannot see values changing over time, as in the Histogram. You can, however, customize the display in several ways by selecting from among the Barchart's pull-down menus; the most interesting being instantaneous ("current") values vs. average values vs. total values.
- **Table**. This visi is similar to the Barchart, except that exact values are displayed on screen. (Think of Table as a digital watch, and the Barchart as an analog one.)

The visis must be in the user's search path. So, for example, add <base directory> / visiClients / $PLATFORM to the $path environment variable before running kperfmon.

10. Both comma-separated and space-separated lists are supported.

As mentioned in the tutorial, instrumentation does not take place when a visualization is started. Kernel instrumentation takes place when the command is given to add the current metric and resource selections to a given visi, which can be done in one of two ways: (1) From within a visi's GUI, select the menu item "Add curve(s)", or (2) from kperfmon's tcl scripting interface, with the addCurrSelectionsToVisi command (see Section 4.4.4). Whichever method is chosen, the kernel will get instrumented with the currently selected metrics and resources, the visi will be informed of these pairings (so it can update its GUI), and periodic sampling (from within kerninstd, which sends its data to kperfmon, who then forwards it to relevant visi(s) for processing) will begin to take place.

4.3 Debugging kperfmon and kerninstd

If you suspect a bug in kperfmon and/or kerninstd, the following widget in the kperfmon window can help track down a problem. Viewing a machine code disassembly of the function(s) being instrumented is a good way to understand how KernInst works. The two buttons in the kperfmon window under the label "Disassemble Selected Fn/BasicBlock" are a good place to start. These buttons will disassemble every function or basic block that is presently selected in the "Kernel Code" tree. In Figure 7, we see a disassembly of kmem_alloc() achieved by selecting kmem_alloc() in the code tree and then clicking on "Disassemble (orig)".

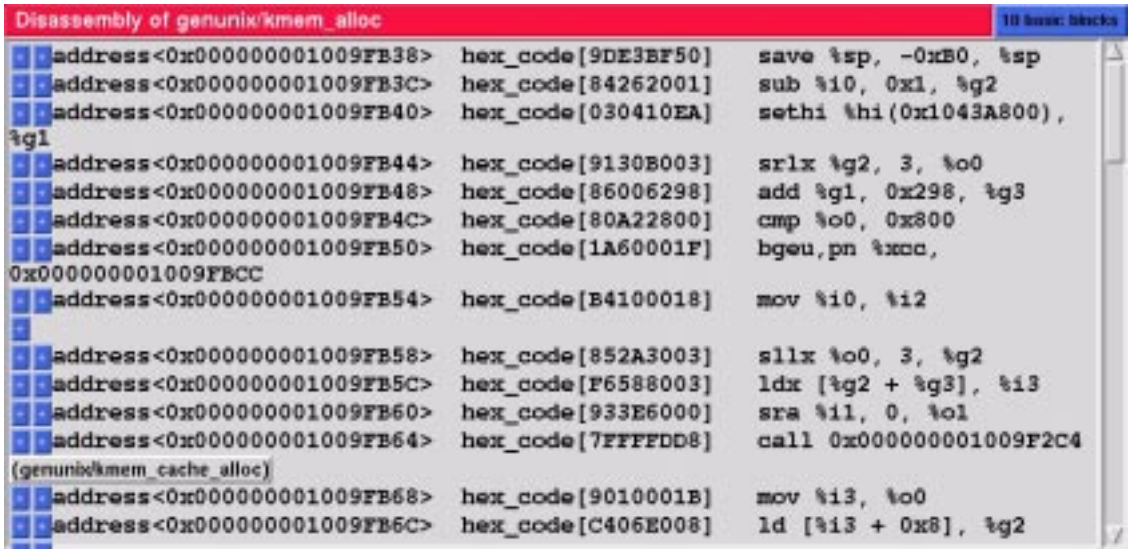

Figure 7: Disassembly of kmem_alloc()

The disassembly window shows, on each line, the instruction's address, a hex representation of its raw byte code, and its disassembly. Calls to fixed-addresses (i.e., excluding calls via a register) will have the callee's name as the title of a pull-down menu that will allow the *callee* to be disassembled, selected, and unselected, respectively. To the left of each instruction are two "+" buttons, which when clicked on, will give a live register analysis for this instruction. Clicking on the left most button will show which registers

kerninstd thinks are killed (and thus are available for scratch usage by instrumentation code), and which are potentially holding live values. Often on the Solaris kperfmon, there are a pair of "killed/made live" lines; this indicates two different Sparc register windows. Clicking on the right most "+" button will show a live register analysis for this instruction *in isolation*.

The button "Disassemble(orig)" differs from "Disassemble (curr mem)" in one key manner: the latter will disassemble from the present contents of kernel memory (and thus, will show the effects of any dynamic instrumentation); the former will always present a disassembly of the instructions at the moment kerninstd originally parsed them. The current memory disassembly also does not provide live register analysis buttons.

The disassembly window can help debug kerninstd/kperfmon in several ways: if you believe that instrumentation code is overwriting a register that is not free for scratch usage, you might be able to determine that by viewing the live register analysis and the instrumentation code itself. Viewing the instrumentation code is fairly easy: a disassembly (curr mem) of a function will show that certain instruction(s) have been overwritten with a branch to instrumentation code; simply follow that branch to determine where instrumentation code begins.[11] Once that is done, you can disassemble a specific range of memory addresses: enter the starting address in the "From-addr:" entry widget, and an ending address in the "To-addr:" entry widget and click on "Disassemble (kernel)". This will present a basic disassembly of this address range (no basic blocks boundaries are shown, and no register analysis is done).

The "Disassemble (kerninstd)" button, only available in the Solaris kperfmon, is similar, but it disassembles an address range in kerninstd's, not the kernel's, memory space. This is less often needed, but can be useful because the Solaris kperfmon often downloads code into kerninstd; the classic example being code to periodically sample the counter(s) and/or timer(s) that are updated by instrumentation code.[12] For the Linux kperfmon, downloading of code is not used, and thus parts of kperfmon that used downloading in the Solaris version have been rearchitected to use new KerninstAPI interface functions. It is expected that a future release of kperfmon on Solaris will also use these new interfaces, as they are much less complicated to use than generating custom downloadable code.

---

11. As mentioned in our OSDI '99 paper, a single branch instruction usually cannot reach the instrumentation code, so a two-level jump scheme is usually needed: the branch instruction branches to an intermediate locale called a *springboard*, which in turn takes several (usually three) instructions to jump to the instrumentation code. Thus, a few disassemblies will be needed in practice before getting to the instrumentation code.
12. On Solaris, kerninstd mmap()s the part of the kernel space where counters and timers are allocated, to make periodic sampling possible without the kernel's knowledge or assistance. On Linux, where mapping of kernel memory is unsupported, the counters and timers are read from the kernel memory device file, /dev/kmem.

4.4 Scripting kperfmon actions

Kperfmon has a tcl command line interface that can be accessed by typing into the shell window from which kperfmon was started (not the kperfmon GUI window). This section describes tcl commands that can drive kperfmon.

4.4.1 Scripting Metrics

The tcl command "metrics" can be used to select or unselect metrics within kperfmon.

`metrics select "`*`metricname`*`"` where *metricname* is, for example, "entries to" or "vtime", will select the appropriate metric. The spelling of the metric name must be exactly the same as in the checkbuttons at the upper-left of kperfmon's GUI window. If a metric name contains spaces, then quotes around *metricname* are mandatory for correct behavior.

`metrics unselect "`*`metricname`*`"` will unselect that metric.

`metrics clear` will unselect all metrics at once.

As always, remember that a metric, once selected, stays selected until explicitly unselected.

4.4.2 Scripting Resources

The tcl command "resources" can be used to select or unselect resources within kperfmon.

`resources select` *`modname fnname`* where *modname* is, for example, `genunix`, and where *fnname* is, for example, `kmem_alloc`, will select a particular resource.

`resources unselect` *`modname fnname`* will unselect a particular resource.

`resources clear` will unselect every resource currently selected.

`resources select` *`hexaddress`* will select a function when given any address that lies within it. For example, `resources select` *`0x100014ce`*.

As always, remember that a resource, once selected, stays selected until explicitly unselected.

4.4.3 Scripting Starting/Closing Visis

kperfmon provides a tcl interface to starting and closing visis.

`newVisi` *`visiname`* will launch the visi *visiname*. Note that the current metric and resource selections are not added to the visi by this command, nor does starting a visi cause the kernel to be instrumented. *visiname* is the name of the visi; if it is not in the current search path, you can use a full path name for *visiname*.

The *newVisi* command returns an integer identifier for this visi that will need to be passed to the *closeVisi* and *addCurrSelectionsToVisi* commands, below. To remember this identifier in a tcl script, one can assign the result integer to a tcl variable, as in

```
set myvisiid [newVisi rthist]
```

For the curious, you can view the value of this id by dumping it to stdout:

```
puts stdout $myvisiid
```

A visi can be closed with the *closeVisi* command, as in:

```
closeVisi $myvisiid
```

As always, closing a visi will uninstrument the kernel (assuming some instrumentation had been activated and was being sent to the visi).

4.4.4 Adding Current Metric and Resource Selections to a Visi

In a scripted session, between opening and closing a visi, you will in all likelihood want to instrument the kernel and cause some values to be sent to the visi. Selecting metrics and resources in a script were discussed above in Section 4.4.1 and Section 4.4.2. The *addCurrSelectionsToVisi* command will gather the current metric and resource selections, instrument the kernel, inform a visi so it may update its GUI, and begins the process of sampling data values, which will be sent to a visi. The command takes in a single parameter, the visi id, as returned by the *newVisi* command. An example is:

```
addCurrSelectionsToVisi $myvisiid
```

4.4.5 Saving Data Values Collected by a Visi

Before closing a visi, a scripting session will probably wish to save the data values that have been collected by the visi to some file. This is not presently implemented in kperfmon at the scripting level. However, this functionality is available by selecting “Save ...” from the visi “File” menu, which will bring up a dialog box allowing you to select the specific metric/resource combinations currently enabled in the visi for which you wish to save values. After indicating the values you wish to save, click the “Export” button to bring up another dialog box that allows you to choose a file name and location in which to save the selected data.

4.4.6 Launching a Script

The above tcl commands can be typed directly into kperfmon’s command line interface (which is in the shell that launched kperfmon, not kperfmon’s GUI window). A scripted sequence of these commands can be put in a “.tcl” file, and be launched by typing the tcl *source* command, as in:

```
source myfile.tcl
```

Of course, at least a rudimentary knowledge of the tcl language will be required to get much use out of scripting. The best place to start learning tcl is:

```
http://www.tcl.tk
```