NAME

collect − command used to collect program performance data

SYNOPSIS

collect *collect-arguments target target-arguments*
collect
collect −V
collect −R

DESCRIPTION

The **collect** command runs the target process and records performance data and global data for the process. Performance data is collected using profiling or tracing techniques. The data can be examined with a GUI program (**analyzer**) or a command-line program (**er_print**). The data collection software run by the **collect** command is referred to here as the Collector.

The data from a single run of the **collect** command is called an experiment. The experiment is represented in the file system as a directory, with various files inside that directory.

The *target* is the path name of the executable, Java(TM) **.jar** file, or Java **.class** file for which you want to collect performance data. (For more information about Java profiling, see JAVA PROFILING, below.) Executables that are targets for the **collect** command can be compiled with any level of optimization, but must use dynamic linking. If a program is statically linked, the **collect** command prints an error message. In order to see annotated source using **analyzer** or **er_print**, targets should be compiled with the **-g** flag, and should not be stripped.

In order to enable dataspace profiling, executables must be compiled with the **-xhwcprof -xdebugformat=dwarf -g** flags. These flags are valid for the C, C++ and Fortran compilers, but only on SPARC[R] platforms. See the section "DATASPACE PROFILING", below.

The **collect** command uses the following strategy to find its target:

- If there is a file with the name of the target that is marked executable, the file is verified as an ELF executable that can run on the target machine. If the file is not such a valid ELF executable, the **collect** command fails.
- If there is a file with the name of the target, and the file is not executable, **collect** checks whether the file is a Java[TM] jar file or class file. If the file is a Java jar file or class file, the Java[TM] virtual machine (JVM) software is inserted as the target, with any necessary flags, and data is collected on that JVM machine. (The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java[TM] platform.) See the section on "JAVA PROFILING", below.
- If there is no file with the name of the target, your path is searched to find an executable; if an executable is found, it is verified as described above.
- If no file of the current name is found, the command looks for a file with that name and the string **.class** appended; if a file is found, the target of a JVM machine is inserted, with the appropriate flags, as above.
- If none of these procedures can find the target, the command fails.

OPTIONS

If invoked with no arguments, print a usage summary, including the default configuration of the experiment. If the processor supports hardware counter overflow profiling, print two lists containing information about hardware counters. The first list contains "aliased" hardware counters; the second list contains raw hardware counters. For more details, see the "Hardware Counter Overflow Profiling" section below.

Data Specifications

−p *option*

Collect clock-based profiling data. The allowed values of *option* are:

Value	Meaning

off Turn off clock-based profiling

on Turn on clock-based profiling with the default profiling interval of approximately 10 milliseconds.

lo[**w**] Turn on clock-based profiling with the low-resolution profiling interval of approximately 100 milliseconds.

hi[**gh**] Turn on clock-based profiling with the high-resolution profiling interval of approximately 1 millisecond.

n Turn on clock-based profiling with a profiling interval of *n*. The value *n* can be an integer or a floating-point number, with a suffix of **u** for values in microseconds, or **m** for values in milliseconds. If no suffix is used, assume the value to be in milliseconds.

If the value is smaller than the clock profiling minimum, set it to the minimum; if it is not a multiple of the clock profiling resolution, round down to the nearest multiple of the clock resolution. If it exceeds the clock profiling maximum, report an error. If it is negative or zero, report an error. If invoked with no arguments, report the clock-profiling intervals.

An optional + can be prepended to the clock-profiling interval, specifying that **collect** capture dataspace data. It will do so by backtracking one instruction, and if that instruction is a memory instruction, it will assume that the delay was attributed to that instruction and record the event, including the virtual and physical addresses of the memory reference.

Caution must be used in interpreting clock-based dataspace data; the delay might be completely unrelated to the memory instruction that happened to precede the instruction with the clock-profile hit; for example, if a memory instruction hits in the cache, but is in a loop executed many times, high counts on that instruction might appear to indicate memory stall delays, but they do not. This situation can be disambiguated by examining the disassembly around the instruction indicating the stall. If the surrounding instructions also have high clock-profiling metrics, the memory delay is likely to be spurious.

Clock-based dataspace profiling should be used only on machines that do not support hardware counter profiling on memory-based counters.

See the section "DATASPACE PROFILING", below.

If no explicit **−p off** argument is given, and no hardware counter overflow profiling is specified, turn on clock-based profiling.

−h *ctr_def*...[**,***ctr_n_def*]

Collect hardware counter overflow profiles. The number of counter definitions, (*ctr_def* through *ctr_n_def*) is processor-dependent. For example, on an UltraSPARC III system, up to two counters can be programmed; on an Intel Pentium IV with Hyperthreading, up to 18 counters are available. You can ascertain the maximum number of hardware counters definitions for profiling on a target system, and the full list of available hardware counters, by running the collect command without any arguments.

This option is now available on systems running the Linux OS. You are responsible for installing the required perfctr patch on the system; that patch can be downloaded from:

http://user.it.uu.se/~mikpe/linux/perfctr/2.6/perfctr-2.6.15.tar.gz

Instructions for installation are contained within that tar file. The user-level *libperfctr.so* libraries are searched for using **LD_LIBRARY_PATH**, and then in */usr/local/lib*, */usr/lib/*, and */lib/* for the 32-bit versions, or */usr/local/lib64 /usr/lib64/*, and */lib64/* for the 64-bit versions.

Each counter definition takes one of the following forms, depending on whether attributes for hardware counters are supported on the processor:

1. [+]*ctr*[*/reg#*][**,***interval*]
2. [+]*ctr*[**~***attr*=*val*]...[**~***attrN*=*valN*][*/reg#*][**,***interval*]

The meanings of the counter definition options are as follows:

Value	Meaning
+	Optional parameter that can be applied to memory-related counters. Causes **collect** to collect dataspace data by backtracking to find the instruction that triggered the overflow, and to find the virtual and physical addresses of the memory reference. Backtracking works on SPARC processors, and only with counters of type **load**, **store**, or **load-store**, as displayed in the counter list obtained by running the **collect** command without any command-line arguments. See the section "DATASPACE PROFILING", below.
ctr	Processor-specific counter name. You can ascertain the list of counter names by running the **collect** command without any command-line arguments. On most systems, even if a counter is not listed, it can still be specified by a numeric value, either in hexadecimal (0x1234) or decimal. Drivers for older chips do not support numeric values, but drivers for more recent chips do.
attr=*val*	On some processors, attribute options can be associated with a hardware counter. If the processor supports attribute options, then running **collect** without any command-line arguments specifies the counter definition, *ctr_def*, in the second form listed above, and provide a list of attribute names to use for *attr*. Value *val* can be in decimal or hexadecimal format. Hexadecimal format numbers are in C program format where the number is prepended by a zero and lower-case x (0x*hex_number*).
reg#	Hardware register to use for the counter. If not specified, **collect** attempts to place the counter into the first available register and as a result, might be unable to place subsequent counters due to register conflicts. If you specify more than one counter, the counters must use different registers. The list of allowable register numbers can be ascertained by running the **collect** command without any command-line arguments.
interval	Sampling frequency, set by defining the counter overflow value. Valid values are as follows:

Value	Meaning
on	Select the default rate, which can be determined by running the **collect** command without any command-line arguments. Note that the default value for all raw counters is the same, and might not be the most suitable value for a specific counter.
hi	Set interval to approximately 10 times shorter than **on**.
lo	Set interval to approximately 10 times longer than **on**.
value	Set interval to a specific value, specified in decimal or hexadecimal format.

An experiment can specify both hardware counter overflow profiling and clock-based profiling. If hardware counter overflow profiling is specified, but clock-based profiling is not explicitly specified, turn off clock-based profiling.

For more information on hardware counters, see the "Hardware Counter Overflow Profiling"

section below.

−s *option*
Collect synchronization tracing data.

The minimum delay threshold for tracing events is set using *option*. The allowed values of *option* are:

Value	Meaning
on	Turn on synchronization delay tracing and set the threshold value by calibration at runtime
calibrate	Same as **on**
off	Turn off synchronization delay tracing
n	Turn on synchronization delay tracing with a threshold value of *n* microseconds; if *n* is zero, trace all events
all	Turn on synchronization delay tracing and trace all synchronization events

By default, turn off synchronization delay tracing.

Record synchronization events for Java monitors, but not for native synchronization within the JVM machine.

−H *option*
Collect heap trace data. The allowed values of *option* are:

Value	Meaning
on	Turn on tracing of memory allocation requests
off	Turn off tracing of memory allocation requests

By default, turn off heap tracing.

Record heap-tracing events for any native calls. Treat calls to **mmap** as memory allocations.

Heap profiling is not supported for Java programs. Specifying it is treated as an error.

Note that heap tracing might produce very large experiments. Such experiments are very slow to load and browse.

−M *option*
Specify collection of an MPI experiment. (See MPI PROFILING, below.) The target of **collect** should be **mpirun**, and its arguments should be separated from the user target (that is the programs that are to be run by **mpirun**) by an inserted **--** argument. The experiment is named as usual, and is referred to as the "founder experiment"; its directory contains subexperiments for each of the MPI processes, named by rank. It is recommended that the **--** argument always be used with **mpirun**, so that an experiment can be collected by prepending **collect** and its options to the **mpirun** command line.

The allowed values of *option* are:

Value	Meaning
MPI-version	Turn on collection of an MPI experiment, assuming the MPI version named
off	Turn off collection of an MPI experiment

By default, turn off collection of an MPI experiment. When an MPI experiment is turned on, the

default setting for **-m** (see below) is changed to **on**.

The supported versions of MPI are printed when you type **collect** with no arguments, or in response to an unrecognized version specified with **-M**.

−m *option*

Collect MPI tracing data. (See MPI PROFILING, below.)

The allowed values of *option* are:

Value	**Meaning**
on	Turn on MPI tracing information
off	Turn off MPI tracing information

By default, turn off MPI tracing, except if the **-M** flag is enabled, in which case MPI tracing is turned on by default. Normally, MPI experiments are collected with **-M**, and no user control of MPI tracing is needed. If you want to collect an MPI experiment, but not collect MPI trace data, you can use the explicit flags:

-M on -m off.

−c *option*

Collect count data, using bit(1) instrumentation. This option is available only on SPARC-based systems.

The allowed values of *option* are:

Value	**Meaning**
on	Turn on count data
static	Turn on simulated count data, based on the assumption that every instruction was executed exactly once.
off	Turn off count data

By default, turn off count data. Count data cannot be collected with any other type of data. For count data or simulated count data, the executable and any shared-objects that are instrumented and statically linked are counted; for count data, but not simulated count data, dynamically loaded shared objects are also instrumented and counted.

In order to collect count data, the executable must be compiled with the **-xbinopt=prepare** flag.

−I *directory*

Specify a directory for bit(1) instrumentation. This option is available only on SPARC-based systems, and is meaningful only when *-c* is specified.

−N *libname*

Specify a library to be excluded from bit(1) instrumentation, whether the library is linked into the executable, or loaded with dlopen. This option is available only on SPARC-based systems, and is meaningful only when *-c* is also specified. Multiple **-N** options can be specified.

−r *option*

Collect thread-analyzer data.

The allowed values of *option* are:

Value	**Meaning**

on	Turn on thread analyzer data-race-detection data
all	Turn on all thread analyzer data
off	Turn off thread analyzer data
dt1,...,dtN	Turn on specific thread analyzer data types, as named by the *dt** parameters.

The specific types of thread analyzer data that can be requested are:

Value	**Meaning**
race	Collect datarace data
deadlock	Collect deadlock and potential-deadlock data

By default, turn off all thread-analyzer data.

Thread Analyzer data cannot be collected with any tracing data, but can be collected in conjunction with clock- or hardware counter profiling data. Thread Analyzer data significantly slows down the execution of the target, and profiles might not be meaningful as applied to the user code.

Thread Analyzer experiments can be examined with either **analyzer** or with **tha**. The latter displays a simplified list of default tabs, but is otherwise identical.

In order to enable data-race detection, executables must be instrumented, either at compile time, or by invoking a postprocessor. If the target is not instrumented, and none of the shared objects on its library list is instrumented, a warning is displayed, but the experiment is run. Other Thread Analyzer data do not require instrumentation.

See the **tha**(1) man page for more detail.

−S *interval*

Collect periodic samples at the interval specified (in seconds). Record data samples from the process, and include a timestamp and execution statistics from the kernel, among other things. The allowed values of *interval* are:

Value	**Meaning**
off	Turn off periodic sampling
on	Turn on periodic sampling with the default sampling interval (1 second)
n	Turn on periodic sampling with a sampling interval of *n* in seconds; *n* must be positive.

By default, turn on periodic sampling.

If no data specification arguments are supplied, collect clock-based profiling data, using the default resolution.

If clock-based profiling is explicitly disabled, and neither hardware counter overflow profiling nor any kind of tracing is enabled, display a warning that no function-level data is being collected, then execute the target and record global data.

Experiment Controls

−L *size* Limit the amount of profiling and tracing data recorded to *size* megabytes. The limit applies to the sum of all profiling data and tracing data, but not to sample points. The limit is only approximate, and can be exceeded. When the limit is reached, stop profiling and tracing data, but keep the experiment open and record samples until the target process terminates. The allowed values of *size* are:

Value	Meaning
unlimited or none	Do not impose a size limit on the experiment
n	Impose a limit of *n* MB.; *n* must be positive and greater than zero.

The default limit on the amount of data recorded is 2000 Mbytes.

−F *option*

Control whether or not descendant processes should have their data recorded. The allowed values of *option* are:

Value	Meaning
on	Record experiments on descendant processes from **fork** and **exec**
all	Record experiments on all descendant processes
off	Do not record experiments on descendant processes
=<regex>	Record experiments on all descendant processes whose executable name (a.out name) or lineage match the regular expression.

By default, do not record descendant processes. For more details, read the section "FOLLOWING DESCENDANT PROCESSES", below.

−A *option*

Control whether or not load objects used by the target process should be archived or copied into the recorded experiment. The allowed values of *option* are:

Value	Meaning
on	Archive load objects into the experiment.
off	Do not archive load objects into the experiment.
copy	Copy and archive load objects (the target and any shared objects it uses) into the experiment.

If you copy experiments onto a different machine, or read the experiments from a different machine, specify **−A copy**. Note that doing so does not copy any sources or object files (.o's); it is your responsibility to ensure that those files are accessible from the machine where the experiment is being examined.

The default setting for **−A** is **on**.

−j *option*

Control Java profiling when the target is a JVM machine. The allowed values of *option* are:

Value	Meaning
on	Record profiling data for the JVM machine, and recognize methods compiled by the Java HotSpot[TM] virtual machine, and also record Java callstacks.
off	Do not record Java profiling data.
<path>	Record profiling data for the JVM, and use the JVM as installed in <path>.

See the section "JAVA PROFILING", below.

You must use **−j on** to obtain profiling data if the target is a JVM machine. The **−j on** option is not needed if the target is a class or jar file. if you are on a 64-bit JVM machine, you must specify its path explicitly as the target; do not use the **-d64** option for a 32-bit JVM machine. If the **-j on** option is specified, but the target is not a JVM machine, an invalid argument might be passed to

the target, and no data would be recorded. The **collect** command validates the version of the JVM machine specified for Java profiling.

−J *java_arg*

Specify additional arguments to be passed to the JVM used for profiling. If **−J** is specified, but Java profiling is not specified, an error is generated, and no experiment run. The *java_arg* must be surrounded by quotes if it contains more than one argument. It consists of a set of tokens, separated by either a blank or a tab; each token is passed as a separate argument to the JVM. Note that most arguments to the JVM must begin with a "-" character.

−l *signal*

Record a sample point whenever the given signal is delivered to the process.

−y *signal*[**,r**]

Control recording of data with *signal*. Whenever the given signal is delivered to the process, switch between paused (no data is recorded) and resumed (data is recorded) states. Start in the resumed state if the optional **,r** flag is given, otherwise start in the paused state. This option does not affect the recording of sample points.

Output Controls

−o *experiment_name*

Use *experiment_name* as the name of the experiment to be recorded. The *experiment_name* must end in the string **.er**; if not, print an error message and do not run the experiment.

If **−o** is not specified, give the experiment a name of the form *stem***.***n***.er**, where *stem* is a string, and *n* is a number. If a group name has been specified with **−g**, set *stem* to the group name without the **.erg** suffix. If no group name has been specified, set *stem* to the string "**test**".

If invoked from one of the commands used to run MPI jobs, for example, **mpirun**, but without -*M on*, and **−o** is not specified, take the value of *n* used in the name from the environment variable used to define the MPI rank of that process. Otherwise, set *n* to one greater than the highest integer currently in use. (See MPI PROFILING, below.)

If the name is not specified in the form *stem***.***n***.er**, and the given name is in use, print an error message and do not run the experiment. If the name is of the form *stem***.***n***.er** and the name supplied is in use, record the experiment under a name corresponding to one greater than the highest value of *n* that is currently in use. Print a warning if the name is changed.

−d *directory_name*

Place the experiment in directory *directory_name*. If no directory is given, place the experiment in the current working directory. If a group is specified (see **−g**, below), the group file is also written to the directory named by **−d**.

For the lightest-weight data collection, it is best to record data to a local file, with **−d** used to specify a directory in which to put the data. However, for MPI experiments on a cluster, the founder experiment must be available at the same path to all processes to have all data recorded into the founder experiment.

Experiments written to long-latency file systems are especially problematic, and might progress very slowly, especially if Sample data is collected (**−S on**, the default). If you must record over a long-latency connection, disable Sample data.

−g *group_name*

Add the experiment to the experiment group *group_name*. The *group_name* string must end in the string *.erg*; if not, report an error and do not run the experiment.
The first line of a group file must contain the string

```
#analyzer experiment group
```

and each subsequent line is the name of an experiment.

−O *file* Append all output from **collect** itself to the named file, but do not redirect the output from the spawned target. If *file* is set to */dev/null* suppress all output from **collect**, including any error messages.

−t *duration*
Collect data for the specified duration. *duration* can be a single number, followed by either **m**, specifying minutes, or **s**, specifying seconds (default), or two such numbers separated by a **-** sign. If one number is given, data is collected from the start of the run until the given time; if two numbers are given, data is collected from the first time to the second. If the second time is zero, data is collected until the end of the run. If two non-zero numbers are given, the first must be less than the second.

Other Arguments

−P <*pid*>
Write a script for **dbx** to attach to the process with the given PID, and collect data from it, and then invoke **dbx** with that script. Only profiling data, not tracing data can be specified, and timed runs (**-t**) are not supported.

−C *comment*
Put the comment into the **notes** file for the experiment. Up to ten **-C** arguments can be supplied.

−n Dry run: do not run the target, but print all the details of the experiment that would be run. Turn on **-v**.

−R Display the text version of the performance tools README in the terminal window. If the README is not found, print a warning. Do not examine further arguments and do no further processing.

−V Print the current version. Do not examine further arguments and do no further processing.

−v Print the current version and further detailed information about the experiment being run.

−x Leave the target process stopped on the exit from the *exec* system call, in order to allow a debugger to attach to it. The **collect** command prints a message with the process PID.

To attach a debugger to the target once it is stopped by **collect**, you must follow the procedure below.

- Obtain the PID of the process from the message printed by the **collect -x** command
- Start the debugger
- Configure the debugger to ignore SIGPROF and, if you chose to collect hardware counter data, SIGEMT on Solaris or SIGIO on Linux
- Attach to the process using the PID.

As the process runs under the control of the debugger, the Collector records an experiment.

FOLLOWING DESCENDANT PROCESSES

Data from the initial process spawned by collect, called the founder process, is always collected. Processes can create descendant processes by calling system library functions, including the variants of **fork**, **exec**, **system**, *etc.*. If a **-F** argument is used, the collector can collect data for descendant processes, and it opens a new experiment for each descendant process inside the parent experiment. These new experiments are named with their lineage as follows:

- An underscore is appended to the creator's experiment name.
- A code letter is added: either "f" for a fork, or "x" for an exec, or "c" for other descendants.
- A number is added after the code letter, which is the index of the fork or exec. The assignment of this number is applied whether the process was started successfully or not.

- The experiment suffix, ".er" is appended to the lineage.

For example, if the experiment name for the initial process is "test.1.er", the experiment for the descendant process created by its third fork is "test.1.er/_f3.er". If that descendant process execs a new image, the corresponding experiment name is "test.1.er/_f3_x1.er".

If **-F on** is used, descendant processes initiated by calls to **fork**(2), **fork1**(2), **fork**(3F), **vfork**(2), and **exec**(2) and its variants are followed. The call to **vfork** is replaced internally by a call to **fork1**. Descendants creates by calls to **system**(3C), **system**(3F), **sh**(3F), **popen**(3C), and similar functions, and their associated descendant processes, are not followed.

If the **-F all** argument is used, all descendants are followed, including those from **system**(3C), **system**(3F), **sh**(3F), **popen**(3C), and similar functions.

If the **-F =<regex>** argument is used, all descendants whose name or lineage match the regular expression are followed. When matching lineage, the ".er" should be omitted. When matching names, both the command, and its arguments are part of the expression.

For example, to capture data on the descendant process of the first **exec** from the first **fork** from the first call to **system** in the founder, use:

collect -F '=_c1_f1_x1'

To capture data on all the variants of **exec**, but not **fork**, use:

collect -F '=.*_x[0-9]/*'

To capture data from a call to **system("echo hello")**
but not **system("goodbye")**, use:

collect -F '=echo hello'

The Analyzer and **er_print** automatically read experiments for descendant processes when the founder experiment is read, but the experiments for the descendant processes are not selected for data display.

To select the data for display from the command line, specify the path name explicitly to either **er_print** or Analyzer. The specified path must include the founder experiment name, and the descendant experiment's name inside the founder directory.

For example, to see the data for the third fork of the test.1.er experiment:

er_print test.1.er/_f3.er
analyzer test.1.er/_f3.er

You can prepare an experiment group file with the explicit names of descendant experiments of interest.

To examine descendant processes in the Analyzer, load the founder experiment and chhose View > Filter data. The Analyzer displays a list of experiments with only the founder experiment checked. Uncheck the founder experiment and check the descendant experiment of interest.

JAVA PROFILING

Java profiling consists of collecting a performance experiment on the JVM machine as it runs your .class or .jar files. If possible, callstacks are collected in both the Java model and in the machine model.

Data can be shown with view mode set to User, Expert, or Machine. User mode shows each method by name, with data for interpreted and HotSpot-compiled methods aggregated together; it also suppresses data for non-user-Java threads. Expert mode separates HotSpot-compiled methods from interpreted methods, and does not suppress non-user Java threads. Machine mode shows data for interpreted Java methods against the JVM machine as it does the interpreting, while data for methods compiled with the Java HotSpot virtual machine is reported for named methods. All threads are shown. In all three modes, data is reported in the usual way for any non-OpenMP C, C++, or Fortran code called by a Java target. Such code corresponds to Java native methods. The Analyzer and the **er_print** utility can switch between the view mode User, view mode Expert, and view mode Machine, with User being the default.

Clock-based profiling and hardware counter overflow profiling are supported. Synchronization tracing collects data only on the Java monitor calls, and synchronization calls from native code; it does not collect data

about internal synchronization calls within the JVM.

Heap tracing is not supported for Java, and generates an error if specified.

When **collect** inserts a target name of *java* into the argument list, it examines environment variables for a path to the *java* target, in the order JDK_HOME, and then JAVA_PATH. For the first of these environment variables that is set, the resultant target is verified as an ELF executable. If it is not, **collect** fails with an error indicating which environment variable was used, and the full path name that was tried.

If neither of those environment variables is set, the **collect** command uses the the version set by your PATH. If there is no *java* in your PATH, a system default of */usr/java/bin/java* is tried.

Java Profiling requires Java[TM] 2 SDK (JDK) 6, Update 3 or later; some earlier versions (but no earlier than JDK 1.4.2) might work, but are not supported.

JAVA PROFILING WITH A DLOPEN'd LIBJVM.SO

Some applications are not pure Java, but are C or C++ applications that invoke dlopen to load *libjvm.so*, and then start the JVM by calling into it. To profile such applications, set the environment variable **SP_COLLECTOR_USE_JAVA_OPTIONS**, and add **-j on** to the **collect** command line. Do not set either **LD_LIBRARY_PATH** for this scenario.

SHARED_OBJECT HANDLING

Normally, the collect command causes data to be collected for all shared objects in the address space of the target, whether on the initial library list, or explicitly *dlopen*'d. However, there are some circumstances under which some shared objects are not profiled.

One such scenario is when the target program is invoked with lazy-loading. In such cases, the library is not loaded at startup time, and is not loaded by explicitly calling *dlopen*, so the shared object name is not included in the experiment, and all PCs from it are mapped to the **<Unknown>** function. The workaround is to set LD_BIND_NOW, to force the library to be loaded at startup time.

Another such scenario is when the executable is built with the **-B direct**. In that case the object is dynamically loaded by a call specifically to the dynamic linker entry point of *dlopen*, and the libcollector interposition is bypassed. The shared object name is not included in the experiment, and all PCs from it are mapped to the **<Unknown>** function. The workaround is to not use **-B direct**.

OPENMP PROFILING

Data collection for OpenMP programs collects data that can be displayed in any of the three view modes, just as for Java programs. The presentation is identical for user mode and expert mode. Slave threads are shown as if they were really forked from the master thread, and have call stacks matching the master thread. Frames in the call stack coming from the OpenMP runtime code (*libmtsk.so*) are suppressed. For machine mode, the actual native stacks are shown.

In user mode, various artificial functions are introduced as the leaf function of a call stack whenever the runtime library is in one of several states. These functions are <OMP-overhead>, <OMP-idle>, <OMP-reduction>, <OMP-implicit_barrier>, <OMP-explicit_barrier>, <OMP-lock_wait>, <OMP-critical_section_wait>, and <OMP-ordered_section_wait>.

Two additional clock-profiling metrics are added to the data for clock-profiling experiments:

OpenMP Work
OpenMP Wait

OpenMP Work is counted when the OpenMP runtime thinks the code is doing work. It includes time when the process is consuming User-CPU time, but it also can include time when the process is consuming System-CPU time, waiting for page faults, waiting for the CPU, *etc.*. Hence, OpenMP Work can exceed User-CPU time. OpenMP Wait is accumulated when the OpenMP runtime thinks the process is waiting. It can include User-CPU time for busy-waits (spin-waits), but it also includes Other-Wait time for sleep-waits.

The inclusive metrics are visible by default; the exclusive are not. Together, the sum of those two metrics equals the Total LWP Time metric. These metrics are added for all clock- and hardware counter profiling

experiments.

Collecting information for every fork in the execution of the program can be very expensive. You can suppress that cost by setting the environment variable **SP_COLLECTOR_NO_OMP**. If you do so, the program will have substantially less dilation, but you will not see the data from slave threads propagate up the the caller, and eventually to **main()**, as it normally will without that variable being set.

A new collector for OpenMP 3.0 is enabled by default in this release. It can profile programs that use explicit tasking. Programs built with earlier compilers can be profiled with the new collector only if a patched version of *libmtsk.so* is available. If it is not installed, you can switch data collection to use the old collector by setting the environment variable **SP_COLLECTOR_OLDOMP**.

Note that the OpenMP profiling functionality is only available for applications compiled with the Studio compilers, since it depends on the Studio compiler runtime. GNU-compiled code will only see machine-level callstacks.

DATASPACE PROFILING

A dataspace profile is a data collection in which memory-related events, such as cache misses, are reported against the data object references that cause the events rather than just the instructions where the memory-related events occur. Dataspace profiling is not available on systems running the Linux OS, nor on x86 based systems running the Solaris OS.

To allow dataspace profiling, the target can be written in C, C++ or Fortran, and must be compiled for SPARC architecture, with the **-xhwcprof -xdebugformat=dwarf -g** flags, as described above. Furthermore, the data collected must be hardware counter profiles and the optional **+** must be prepended to the counter name. If the optional **+** is prepended to one memory-related counter, but not all, the counters without the **+** reports dataspace data against the *<Unknown>* data object, with subtype *(Dataspace data not requested during data collection)*.

With the data collected, the **er_print** utility allows three additional commands: **data_objects**, **data_single**, and **data_layout**, as well as various commands relating to Memory Objects. See the **er_print(1)** man page for more information.

In addition, the Analyzer now includes two tabs related to dataspace profiling, labeled DataObjects and DataLayout, as well as a set of tabs relating to Memory Objects. See the **analyzer(1)** man page for more information.

Clock-based dataspace profiling should only be used on machines that do not support hardware counter profiling with memory-based counters. It requires the same compilation flags as for hardware counter profiling. Data should be interpreted with care, as explained above.

MPI PROFILING

The **collect** command can be used for MPI profiling to manage collection of the data from the constituent MPI processed, collect MPI trace data, and organize the data into a single "founder" experiment, with "subexperiments" for each MPI process.

The **collect** command can be used with MPI by simply prefacing the command that starts the MPI job and its arguments with the desired **collect** command and its arguments (assuming you have inserted the **--** argument to indicate the end of the **mpirun** arguments). For example, on an SMP machine,

```
% mpirun -np 16 -- a.out 3 5
```

can be replaced by

```
% collect -M on mpirun -np 16 -- a.out 3 5
```

This command runs an MPI tracing experiment on each of the 16 MPI processes, collecting them all in an MPI experiment, named by the usual conventions for naming experiments.

The initial **collect** process reformats the **mpirun** command to specify running **collect** with appropriate arguments on each of the individual MPI processes.

Note that the **--** argument immediately before the target name is required for MPI profiling (although it is optional for **mpirun** itself), so that **collect** can separate the **mpirun** arguments from the target and its arguments. If it is not supplied, **collect** prints an error message, and no experiment is run.

Furthermore, a **-x PATH** argument is added to the **mpirun** arguments by **collect**, so that the remote collect's can find their targets. If any environment variables in your environment begin with "VT_" or with "SP_COLLECTOR_", they are passed to the remote collect with -x flags for each.

MIMD MPI runs are supported, with the similar proviso that there must be a "--" argument after each ":" (indicating a new target and local **mpirun** arguments for it). If it is not supplied, **collect** prints an error message, and no experiment is run.

Some versions of Sun HPC ClusterTools have functionality for MPI State profiling. When clock-profiling data is collected on an MPI experiment run with such a version of ClusterTools, two additional metrics can be shown:

MPI Work
MPI Wait

MPI Work accumulates when the process is inside the MPI runtime doing work, such as processing requests or messages; MPI Wait accumulates when the process is inside the MPI runtime, but waiting for an event, buffer, or message.

In the Analyzer, when MPI trace data is collected, two additional tabs are shown, MPI Timeline and MPI Chart.

The technique of using **mpirun** to spawn explicit **collect** commands on the MPI processes is no longer supported to collect MPI trace data, and should not be used. It can still be used for all other types of data.

MPI profiling is based on the open source VampirTrace 5.5.3 release. It recognizes several supported VampirTrace environment variables, and a new one, **VT_STACKS**, which controls whether or not callstacks are recorded in the data. For further information on the meaning of these variables, see the VampirTrace 5.5.3 documentation.

The default values of the environment variables **VT_BUFFER_SIZE** and **VT_MAX_FLUSHES** limit the internal buffer of the MPI API trace collector to 64 MB and the number of times that the buffer is flushed to 1, respectively. Events that are to be recorded after the limit has been reached are no longer written into the trace file. The environment variables apply to every process of a parallel application, meaning that applications with n processes will typically create trace files n times the size of a serial application.

To remove the limit and get a complete trace of an application, set **VT_MAX_FLUSHES** to 0. This setting causes the MPI API trace collector to flush the buffer to disk whenever the buffer is full. To change the size of the buffer, use the environment variable **VT_BUFFER_SIZE**. The optimal value for this variable depends on the application which is to be traced. Setting a small value will increase the memory available to the application but will trigger frequent buffer flushes by the MPI API trace collector. These buffer flushes can significantly change the behavior of the application. On the other hand, setting a large value, like 2G, will minimize buffer flushes by the MPI API trace collector, but decrease the memory available to the application. If not enough memory is available to hold the buffer and the application data this might cause parts of the application to be swapped to disk leading also to a significant change in the behavior of the application.

Another important variable is *VT_VERBOSE*, which turns on various error and status messages, and setting it to 2 or higher is recommended if problems arise.

USING COLLECT WITH PPGSZ

The **collect** command can be used with **ppgsz** by running the **collect** command on the **ppgsz** command, and specifying the **-F on** flag. The founder experiment is on the **ppgsz** executable and is uninteresting. If your path finds the 32-bit version of **ppgsz**, and the experiment is being run on a system that supports 64-bit processes, the first thing the collect command does is execute an exec function on its 64-bit version, creating **_x1.er**. That executable forks, creating **_x1_f1.er**. The descendant process attempts to execute an **exec** function on the named target, in the first directory on your path, then in the second, and so forth, until one of the **exec** functions succeeds. If, for example, the third attempt succeeds, the first two descendant experiments are named **_x1_f1_x1.er** and *_x1_f1_x2.er*, and both are completely empty. The experiment on the target is the one from the successful **exec**, the third one in the example, and is named **_x1_f1_x3.er**, stored

under the founder experiment. It can be processed directly by invoking the Analyzer or the **er_print** utility on **test.1.er/_x1_f1_x3.er**.

If the 64-bit **ppgsz** is the initial process run, or if the 32-bit **ppgsz** is invoked on a 32-bit kernel, the **fork** descendant that executes **exec** on the real target has its data in **_f1.er**, and the real target's experiment is in **_f1_x3.er**, assuming the same path properties as in the example above.

See the section "FOLLOWING DESCENDANT PROCESSES", above. For more information on hardware counters, see the "Hardware Counter Overflow Profiling" section below.

USING COLLECT ON SETUID/SETGID TARGETS

The **collect** command operates by inserting a shared library, **libcollector.so**, into the target's address space (LD_PRELOAD), and by using a second shared library, **collaudit.so**, to record shared-object use with the runtime linker's audit interface (LD_AUDIT). Those two shared libraries write the files that constitute the experiment.

Several problems might arise if **collect** is invoked on executables that call setuid or setgid, or that create descendant processes that call setuid or setgid. If the user running the experiment is not root, collection fails because the shared libraries are not installed in a trusted directory. The workaround is to run the experiments as root, or use **crle**(1) to grant permission. Users should, of course, take great care when circumventing security barriers, and do so at their own risk.

In addition, the umask for the user running the **collect** command must be set to allow write permission for that user, and for any users or groups that are set by the setuid/setgid attributes of a program being **exec**'d and for any user or group to which that program sets itself. If the mask is not set properly, some files might not be written to the experiment, and processing of the experiment might not be possible. If the log file can be written, an error is shown when the user attempts to process the experiment.

Other problems can arise if the target itself makes any of the system calls to set UID or GID, or if it changes its umask and then forks or runs exec on some other process, or **crle** was used to configure how the runtime linker searches for shared objects.

If an experiment is started as root on a target that changes its effective GID, the **er_archive** process that is automatically run when the experiment terminates fails, because it needs a shared library that is not marked as trusted. In that case, you can run **er_archive** (or **er_print** or Analyzer) explicitly by hand, on the machine on which the experiment was recorded, immediately following the termination of the experiment.

DATA COLLECTED

Three types of data are collected: profiling data, tracing data and sampling data. The data packets recorded in profiling and tracing include the callstack of each LWP, the LWP, thread, and CPU IDs, and some event-specific data. The data packets recorded in sampling contain global data such as execution statistics, but no program-specific or event-specific data. All data packets include a timestamp.

Clock-based Profiling

The event-specific data recorded in clock-based profiling is an array of counts for each accounting microstate. The microstate array is incremented by the system at a prescribed frequency, and is recorded by the Collector when a profiling signal is processed.

Clock-based profiling can run at a range of frequencies which must be multiples of the clock resolution used for the profiling timer. If you try to do high-resolution profiling on a machine with an operating system that does not support it, the command prints a warning message and uses the highest resolution supported. Similarly, a custom setting that is not a multiple of the resolution supported by the system is rounded down to the nearest non-zero multiple of that resolution, and a warning message is printed.

Clock-based profiling data is converted into the following metrics:

User CPU Time

Wall Time
Total LWP Time
System CPU Time
Wait CPU Time
User Lock Time
Text Page Fault Time
Data Page Fault Time
Other Wait Time

For experiments on multithreaded applications, all of the times, other than Wall Time, are summed across all LWPs in the process; Wall Time is the time spent in all states for LWP 1 only. Total LWP Time adds up to the real elapsed time, multiplied by the average number of LWPs in the process.

If clock-based profiling is performed on an OpenMP program, two additional metrics:

OpenMP Work
OpenMP Wait

are provided. On the Solaris OS, OpenMP Work accumulates when work is being done in parallel. OpenMP Wait accumulates when the OpenMP runtime is waiting for synchronization, and accumulates whether the wait is using CPU time or sleeping, or when work is being done in parallel, but the thread is not scheduled on a CPU.

On Linux, OpenMP Work and OpenMP Wait are accumulated only when the process is active in either user or system mode. Unless you have specified that OpenMP should do a busy wait, OpenMP Wait on Linux will not be useful.

If clock-based profiling is performed on an MPI program, run under Sun HPC ClusterTools release 8.1 or later, two additional metrics:

MPI Work
MPI Wait

is provided. On Solaris, MPI Work accumulates when the MPI runtime is active. MPI Wait accumulates when the MPI runtime is waiting for the send or receive of a message, or when the MPI runtime is active, but the thread is not running on a CPU.

On Linux, MPI Work and MPI Wait are accumulated only when the process is active in either user or system mode. Unless you have specified that MPI should do a busy wait, MPI Wait on Linux will not be useful. If clock-based dataspace profiling is specified, an additional metric:

Max. Mem Stalls

is provided.

Hardware Counter Overflow Profiling

Hardware counter overflow profiling records the number of events counted by the hardware counter at the time the overflow signal was processed. This type of profiling is now available on systems running the Linux OS, provided that they have the Perfctr patch installed.

Hardware counter overflow profiling can be done on systems that support overflow profiling and that include the hardware counter shared library, **libcpc.so**(3). You must use a version of the Solaris OS no earlier that the Solaris 10 OS. On UltraSPARC[R] computers, you must use a version of the hardware no earlier than the UltraSPARC III hardware. On computers that do not support overflow profiling, an attempt to select hardware counter overflow profiling generates an error.

The counters available depend on the specific CPU processor and operating system. Running the *collect* command with no arguments prints out a usage message that contains the names of the counters. The counters that are aliased to common names are displayed first in the list, followed by a list of the raw hardware counters. If neither the performance counter subsystem nor collect know the names for the counters on a specific chip, the tables are empty. In most cases, however, the counters can be specified numerically. The lines of output are formatted similar to the following:

```
Aliased HW counters available for profiling:
  cycles[/{0|1}],9999991 ('CPU Cycles', alias for Cycle_cnt; CPU-cycles)
  insts[/{0|1}],9999991 ('Instructions Executed', alias for Instr_cnt; events)
  dcrm[/1],100003 ('D$ Read Misses', alias for DC_rd_miss; load events)
  ...
Raw HW counters available for profiling:
  Cycle_cnt[/{0|1}],1000003 (CPU-cycles)
  Instr_cnt[/{0|1}],1000003 (events)
  DC_rd[/0],1000003 (load events)
  SI_snoop[/0],1000003 (not-program-related events)
  ...
```

In the first line of aliased counter output, the first field, "cycles", gives the counter name that can be used in the **−h counter...** argument. It is followed by a specification of which registers can be used for that counter. The next field, "9999991", is the default overflow value for that counter. The following field in parentheses, "CPU Cycles", is the metric name, followed by the raw hardware counter name. The last field, "CPU-cycle", specifies the type of units being counted. There can be up to two words for the type of information. The second or only word of the type information can be either "CPU-cycles" or "events". If the counter can be used to provide a time-based metric, the value is CPU-cycles; otherwise it is events.

The second output line of the aliased counter output above has "events" instead of "CPU-cycles" at the end of the line, indicating that it counts events, and cannot be converted to a time.

The third output line above has two words of type information, "load events", at the end of the line. The first word of type information can have the value of "load", "store", "load-store", or "not-program-related". The first three of these type values indicate that the counter is memory-related and the counter name can be preceded by the "+" sign when used in the collect -h command. The "+" sign indicates the request for data collection to attempt to find the precise instruction and virtual address that caused the event on the counter that overflowed.

The "not-program-related" value indicates that the counter captures events initiated by some other program, such as CPU-to-CPU cache snoops. Using the counter for profiling generates a warning and profiling does not record a call stack. It does, however, show the time being spent in an artificial function called "collector_not_program_related". Thread IDs and LWP IDs are recorded, but are meaningless.

Each line in the raw hardware counter list includes the internal counter name as used by cputrack(1), the register number(s) on which that counter can be used, the default overflow value, and the counter units, which is either CPU-cycles or Events.

EXAMPLES:

Example 1: Using the aliased counter information listed in the above sample output, the following command:

```
collect -h cycles/0,hi,+dcrm,9999
```

enables the CPU Cycle profiling on register 0. The "hi" value enables a sample rate that is approximately 10 times faster than the default rate of 9999991. The "dcrm" value enables the D$ Read Miss profiling on register 1 and the preceding "+" enables Dataspace profiling for the dcrm. The "9999" value sets the sampling to be done every 9999 read misses, instead of the default value of every 100003 read misses.

Example 2:

Running the *collect* command with no arguments on an AMD Opteron machine would produce a raw hardware counter output similar to the following :

```
FP_dispatched_fpu_ops[/{0|1|2|3}],1000003 (events)
FP_cycles_no_fpu_ops_retired[/{0|1|2|3}],1000003 (CPU-cycles)
...
```

Using the above raw hardware counter output, the following command:

```
collect -h FP_dispatched_fpu_ops~umask=0x3/2,10007
```

enables the Floating Point Add and Multiply operations to be tracked at the rate of 1 capture every 10007 events. (For more details on valid attribute values, refer to the processor documentation). The "/2" value specifies the data is to be captured using the register 2 of the hardware.

Synchronization Delay Tracing

Synchronization delay tracing records all calls to the various thread synchronization routines where the real-time delay in the call exceeds a specified threshold. The data packet contains timestamps for entry and exit to the synchronization routines, the thread ID, and the LWP ID at the time the request is initiated. (Synchronization requests from a thread can be initiated on one LWP, but complete on another.)

Synchronization delay tracing data is converted into the following metrics:

Synchronization Delay Events
Synchronization Wait Time

Heap Tracing

Heap tracing records all calls to **malloc**, **free**, **realloc**, **memalign**, and **valloc** with the size of the block requested, its address, and for **realloc**, the previous address.

Heap tracing data is converted into the following metrics:

Leaks
Bytes Leaked
Allocations
Bytes Allocated

Leaks are defined as allocations that are not freed. If a zero-length block is allocated, it counts as an allocation with zero bytes allocated. If a zero-length block is not freed, it counts as a leak with zero bytes leaked.

For applications written in the Java[TM] programming language, leaks are defined as allocations that have not been garbage-collected. Heap profiling for such applications is obsolescent and will not be supported in future releases.

Heap tracing experiments can be very large, and might be slow to process.

MPI Tracing

MPI tracing records calls to the MPI library for functions that can take a significant amount of time to complete. MPI tracing is implemented using the Open Source Vampir Trace code.

MPI tracing data is converted into the following metrics:

MPI Time
MPI Sends
MPI Bytes Sent
MPI Receives
MPI Bytes Received
Other MPI Events

MPI Time is the total LWP time spent in the MPI function. IF MPI state times are also collected, MPI Work Time plus MPI Wait Time for all MPI functions other than MPI_Init and MPI_Finalize should approximately equal MPI Work Time. On Linux, MPI Wait and Work are based on user+system CPU time, while MPI Time is based on real tine, so the numbers will not match.

The MPI Bytes Received metric counts the actual number of bytes received in all messages. MPI Bytes Sent counts the actual number of bytes sent in all messages. MPI Sends counts the number of messages sent, and MPI Recieves counts the number of messages received. MPI_Sendrecv counts as both a send and a receive. MPI Other Events counts the events in the trace that are neither sends nor receives.

Count Data

Count data is recorded by instrumenting the executable, and counting the number of times each instruction was executed. It also counts the number of times the first instruction in a function is executed, and calls that the function execution count.

Count data is converted into the following metric:

Bit Func Count
Bit Inst Exec
Bit Inst Annul

Data−race Detection Data

Data-race detection data consists of pairs of race-access events that constitute a race. The events are combined into a race, and races for which the call stacks for the two access are identical are merged into a race group.

Data-race detection data is converted into the following metric:

Race Accesses

Deadlock Detection Data

Deadlock detection data consists of pairs of threads with conflicting locks.

Deadlock detection data is converted into the following metric:

Deadlocks

Sampling and Global Data

Sampling refers to the process of generating markers along the time line of execution. At each sample point, execution statistics are recorded. All of the data recorded at sample points is global to the program, and does not map to function-level metrics.

Samples are always taken at the start of the process, and at its termination. By default or if a non-zero **−S** argument is specified, samples are taken periodically at the specified interval. In addition, samples can be taken by using the **libcollector**(3) API.

The data recorded at each sample point consists of microstate accounting information from the kernel, along with various other statistics maintained within the kernel.

RESTRICTIONS

The Collector can support up to 16K user threads. Data from additional threads is discarded, and a collector error generated. To support more threads, set the environment variable SP_COLLECTOR_NUMTHREADS to a larger number.

By default, the Collector collects stacks that are 256 frames deep. To support deeper stacks, set the environment variable SP_COLLECTOR_STACKBUFSZ to a larger number.

The Collector interposes on some signal-handling routines to ensure that its use of SIGPROF signals for clock-based profiling and SIGEMT (Solaris) or SIGIO (Linux) for hardware counter overflow profiling is not disrupted by the target program. The Collector library re-installs its own signal handler if the target program installs a signal handler. The Collector's signal handler sets a flag that ensures that system calls are not interrupted to deliver signals. This setting could change the behavior of the target program.

The Collector interposes on **setitimer**(2) to ensure that the profiling timer is not available to the target program if clock-based profiling is enabled.

The Collector interposes on functions in the hardware counter library, **libcpc.so**, so that an application cannot use hardware counters while the Collector is collecting performance data. The interposed functions return a value of -1.

Dataspace profiling are not available on systems running the Linux OS.

For this release, the data from collecting periodic samples is not reliable on systems running the Linux OS.

For this release, wide data discrepancies are observed when profiling multithreaded applications on systems running the RedHat Enterprise Linux OS.

Hardware counter overflow profiling cannot be run on a system where **cpustat** is running, because **cpustat** takes control of the counters, and does not let a user process use them.

Java Profiling requires JDK 6 Update 3 or later updates of JDK 6.

Data is not collected on descendant processes that are created to use the **setuid** attribute, nor on any descendant processes created with an **exec** function run on an executable that is not dynamically linked. Furthermore, subsequent descendant processes might produce corrupted or unreadable experiments. The workaround is to ensure that all processes spawned are dynamically-linked and do not have the **setuid** attribute.

Applications that call **vfork**(2) have these calls replaced by a call to **fork1**(2).

SEE ALSO

analyzer(1), **collector**(1), **dbx**(1), **er_archive**(1), **er_cp**(1), **er_export**(1), **er_mv**(1), **er_print**(1), **er_rm**(1), **tha**(1), **libcollector**(3), and the *Performance Analyzer* manual.