# Shade
# User's Manual

V6.1 (beta 2.0.28)

Sun Microsystems, Inc.
Palo Alto, California 94303, U.S.A.

Copyright 2005 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, Sun Microelectronics, the Sun Logo, Solaris, and SunOS are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

U.S. Government approval required when exporting the product.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Govt is subject to restrictions of FAR 52.227-14(g) (2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

**NAME**

shade_intro − introduction to Shade analyzers

**SYNOPSIS**

analyzer [ −**shade** opt,opt,... ] [ analyzer-options ]

**DESCRIPTION**

This section describes common command-line interfaces that are shared by many Shade analyzers. See the other manpages in this section for a description the analyzers that ship in the standard Shade distribution.

The −**shade** switch specifies options that are interpreted by the Shade library itself, so they are supported by all Shade analyzers. The *analyzer-options* are interpreted and supported by analyzers individually. Although all of the analyzers described in this section support these *analyzer-options*, other analyzers may not.

The options available with the −**shade** switch are listed below. Multiple options should be separated by commas. For example:

**-shade -benchmem=0,-timescale=5.5**

−**assume_ioctl_simple**

This option may provide a work-around if Shade prints the error message "Unknown ioctl". This occurs if Shade does not know how to translate an ioctl( ) request issued by the application program. Many ioctl( ) requests require only simple translations, and this option causes Shade to assume any unknown ioctl( ) requests require only this simple translation. An ioctl( ) request may only use simple translations if the only file descriptor it references is the first argument to the ioctl( ) call. Such ioctl( ) requests may not reference file descriptors via their third argument. Specifying this switch for an application that uses unknown ioctl( ) requests that are not simple will cause the application to behave unpredictably under Shade. When this switch is specified and Shade encounters an unknown application ioctl( ) request, Shade prints a warning message with the unknown ioctl number and performs simple translations on that request.

This switch only takes effect if the −**benchmem=0** switch is also specified.

−**benchmem=***num*

In order for the application and analyzer to colocate in the same address space, Shade normally shifts the address range occupied by the application to avoid conflict with the analyzer. By default, Shade determines a suitable value for this address shift. The −**benchmem** option allows the user to specify this address shift. Regardless of the address shift, Shade simulates the application as though it were executing at its normal address.

The value of *num* must be a multiple of the page size on the host system. When using this option, the user is responsible for choosing an address shift that does not cause the application to conflict with addresses used by the analyzer. If Shade detects a conflict, it issues an error message and terminates immediately.

It is sometimes useful to specify a value of zero for *num*. This causes the application to run at its native address location, which can reduce some of Shade's simulation overhead. Specifying a zero address, however, will cause address conflicts with the analyzer unless the analyzer has been linked at a non-standard location. The analyzers described in this section are all linked at non-standard locations to avoid this conflict. See the ''Introduction to Shade'' manual for instructions on linking your own analyzers at non-standard locations.

−**crt32**

−**crt64**

Shade is able to simulate applications that expect either 32-bit or 64-bit start-up parameters. The start-up parameters (the argument strings, and environment strings) are passed to the application on its stack before it starts executing. The −**crt32** switch informs Shade that the application expects the argument string count and argument pointers to be 32-bit values. The −**crt64** switch informs Shade that the application expects these to be 64-bit values. The default is −**crt32**.

−**flushbench**
−**noflushbench**

These options apply to applications that have self-modifying or dynamically-generated code. The −**flushbench** switch informs Shade that the application executes FLUSH instructions after generating and before executing the code (as required by the SPARC architecture). The −**noflushbench** switch informs Shade that the application does not execute FLUSH instructions as required. Shade performance is greatly improved when in −**flushbench** mode, which is the default.

−**sigdfl=***[!]sig,[!]sig,...*

When a Unix signal is sent to the Shade process, Shade normally ignores the signal and causes the traced application to emulate the effect of receiving that signal. This allows Shade to trace applications that receive signals from other processes. The −**sigdfl** switch informs Shade that the specified signals should not be emulated in the application, but should affect Shade itself. Signals may be specified either by name (without the leading SIG) or by number.

By default, Shade does not pass the SIGINT (CTRL-C) signal to the application. Instead, Shade itself will terminate upon receipt of this signal. Thus, typing CTRL-C will terminate Shade. This behavior can be overridden by specifying −**sigdfl=!INT**. Prefixing a signal with **!** overrides the effect of any previously specified signal and causes the signal to be passed to the application.

−**timescale=***scale-factor*

Since Shade simulates the execution of the application and performs tracing, it inevitably executes the application more slowly than it would normally run. This can adversely affect applications that set up and respond to real-time interval timers. For example, an application that sets up a 1-second interval timer would execute more instructions between each timer expiration when it is run natively than it would when running under Shade. Thus, the application's behavior may be skewed when run under Shade.

The **-timescale** switch provides a way to compensate for this skewing. The parameter to this switch is a scale factor that is used to lengthen the intervals for real-time timers set up by the application. For example, if an application requests a 1-second interval timer and the −**timescale=5.5** switch is specified, Shade would change the interval of the timer to expire every 5.5 seconds. Since each analyzer may slow the application's execution by a different amount, the user must specify a scale factor that is appropriate for that particular analyzer.

−**win32**
−**win64**

Shade is able to simulate applications that use either 32-bit or 64-bit register save areas. The register save area is a location on the application's stack that is reserved by the compiler for each procedure. The application may save its registers in this location upon entry to the procedure. The −**win32** switch informs Shade that the application expects only the low 32 bits of each register to be saved. The −**win64** switch informs Shade that the

application expects (and has reserved enough space) to save all 64 bits of each register. The default is −**win32**.

The *analyzer-options* supported by the analyzers in the standard Shade distribution are:

−**U**

Print a usage message and immediately exit.

−**V**

Print a version message and immediately exit.

−**exec**

If the traced application exec's a new program image, the analyzer will not normally continue tracing the new image but will execute it natively instead. Thus, tracing the shell process, for example, will trace just the shell and not any of the commands it spawns. The −**exec** switch causes the analyzer to trace into the new image. For example, specifying −**exec** when tracing the shell would cause the analyzer to trace not only the shell, but every command spawned by that shell. Each analyzer's manpage specifies how that analyzer presents data collected from multiple applications, so look there for more details.

As a security precaution, Shade will not trace into setuid or setgid programs even if −**exec** is specified unless the owner of the application is the same as the user running Shade.

−**o** *file*

Redirect analyzer output from standard output to *file*.

−**t***from,to*
+**t***from,to*

These options (which may be repeated and/or combined) restrict analysis to specific regions of the application's address space. See **shade_argtrange(3sh)** for more details.

−**v**

Cause the analyzer to print verbose time accounting and version information in addition to its normal output.

An application may be specified to any of the standard-distribution Shade analyzers in one of three ways:

−− *command*

If this option is given, all subsequent arguments, *command*, are interpreted as the application's name and its arguments.

−**c** "*command*"

If this option is given the *command* is interpreted as the application's name and arguments, including possible I/O redirection. For example, −**c** "**ls -l** > **out**" specifies that the **ls** command be traced and that its output be redirected to the file **out**.

If neither the −− nor the −**c** switches are specified, the analyzer reads commands from the terminal in a shell-like mode. Shade's shell supports I/O redirection and simple variable usage. The analyzer traces each application as it is specified. However, each analyzer may display the data collected from multiple application's differently, so see the analyzers's manpage for more details. Shade's shell mode is terminated by typing CTRL-D on an empty line.

If the environment variable **SHADE_BENCH_PATH** is set, Shade uses it as a search path for finding application programs, otherwise Shade uses the **PATH** environment variable.

Any Shade analyzer can trace a variety of different types of applications. If an application uses shared libraries, the analyzer traces the application itself, the shared libraries, and the dynamic loader. If an application is a shell script, the analyzer traces the shell as it interprets the script. Shade analyzers cannot, however, trace setuid or setgid programs unless the owner is the same as the user running Shade.

**FILES**

| | |
|---|---|
| $SHADE | Shade installation base directory |
| $SHADE/bin | contains analyzers |

**SEE ALSO**

The ‘‘Introduction to Shade’’ document.

**NAME**

icount − count executed instructions

**SYNOPSIS**

**icount** [ −**annul** ] [ −**perthread** ]

**DESCRIPTION**

The **icount** analyzer counts and prints the number of instructions executed by each of the specified application programs. In addition to the standard Shade analyzer switches, the **icount** analyzer accepts the following options:

−**annul**

Causes **icount** to count annulled instructions as well as executed instructions.

−**perthread**

Causes **icount** to display the instruction count for each LWP (thread) in the application separately, rather than combining the counts from all threads together.

**THREADS**

The **icount** analyzer combines the execution counts for all application LWPs (threads) together unless the −**perthread** switch is specified.

**FORK AND EXEC**

If the traced application forks, the **icount** analyzer forks too, and each analyzer then reports its own execution count. The execution counts reported for the child are exclusive of the counts reported for the parent. The output from the child application is labeled with the process ID of the child.

**SEE ALSO**

shade_intro(1sh), ifreq(1sh), spixcounts(1sh).

**NAME**

ifreq − opcode execution frequency

**SYNOPSIS**

**ifreq**

**DESCRIPTION**

The **ifreq** analyzer counts and prints the number of instructions executed and annulled on a per-opcode basis by each of the specified application programs. The execution counts for all application programs are combined into a single report.

**THREADS**

The **ifreq** analyzer combines the execution counts for all application LWPs (threads) together.

**FORK AND EXEC**

If the traced application forks, the **ifreq** analyzer forks too, and each analyzer then reports its own set of execution counts. The execution counts reported for the child are exclusive of the counts reported for the parent, and the output from the child is labeled with the process ID of the child process.

**SEE ALSO**

shade_intro(1sh), icount(1sh), spixcounts(1sh).

**NAME**

rcount − count executed instructions per region

**SYNOPSIS**

**rcount** [-o outfile] [-sample sample_info] [-skip skip_count] -r regionfile -- command

**DESCRIPTION**

The **rcount** Shade analyzer counts the number of instructions executed in a set of user defined regions. A region is defined by a starting PC and zero or more ending PC's. When the application executes the instruction at a region's starting PC, that region becomes active. The region remains active until the application executes one of the region's ending PC's. Note, the instructions encompassed by a region need not be contiguous, and there may be many simultaneously active regions.

The **rcount** analyzer maintains a separate counter for each region and increments that counter for each executed instruction whenever the region is active. When the application terminates, the **rcount** analyzer prints the total number of instructions executed within each region.

The required −**r** switch specifies an input file defining the extent of each region in the application. The file consists of lines of the following three forms:

*+region_name address*
*−region_name address*
*#comment*

Lines starting with a plus sign ('+') define the starting address of a region. The *region_name* identifies the name of the region, which can be any sequence of alphanumeric characters. The region starts immediately before the application executes the instruction at the given address. It is an error to specify more than one starting address for the same *region_name.*

Lines starting with a minus sign ('−') define an ending address for a region. The *region_name* must specify a region whose starting address is previously defined in the file. The region ends immediately before the application executes the instruction at the given address, thus the addressed instruction is outside of the region. There may be any number of ending addresses for a region. The region terminates when the application executes any one of them.

Empty lines and lines starting with a pound sign ('#') are considered comments and are ignored. Lines of any other form are illegal and cause an error.

Any *address* in the *regionfile* can be specified either as an address constant, the name of an application symbol, or as an application symbol plus an address constant. Address constants can be specified as an octal, decimal, or hexadecimal number. Constants starting with "0x" or "0X" are interpreted in hexadecimal. Constants starting with a zero ("0") are interpreted in octal. Any other constant is interpreted in decimal. Following are some valid *address* specifiers:

0x1234
main
_start + 0460

The optional −**sample** switch causes **rcount** to gather execution data by sampling rather than tracing every instruction. This mode increases performance, but its statistical nature introduces some error in the result. The −**sample** switch takes a parameter of the following form:

*period,length[,warmup_length]*

Where *period* is the number of instructions from the start of one sample period to the next, *length* is the number of instructions in which data is collected in each sample period, and *warmup_length* (if specified) is the number of instructions in which the state of the active regions is "warmed" prior to each sample period. *period* must be greater than the sum of *length* and *warmup_length.*

The optional −**skip** switch specifies an initial number of application instructions to skip before checking for executed regions. This switch can speed the execution of **rcount** because it executes skipped instructions more quickly than fully traced instructions. The **skip** switch can be used either with or

without the −**sample** switch.

The optional −**o** switch specifies a file to which the output is printed. If no −**o** switch is specified, the output is printed to stdout. The output shows the application's name and parameters, the contents of the region file, some statistics about the execution, and a table with a line for each region that has at least one executed instruction. Each line starts with the name of the region and is followed by the number of instructions executed in that region.

**THREADS**

The **rcount** analyzer is not well behaved for applications that have more than one LWP (thread).

**FORK AND EXEC**

The **rcount** analyzer is not well behaved for applications that fork or exec.

**EXIT CODE**

The **rcount** analyzer exits with 0 on success and 1 on any error.

**SEE ALSO**

shade_intro(1sh).

**NAME**

spixcounts − generate spix counts file

**SYNOPSIS**

**spixcounts** [ −**b** *fmt* ] [ −**data** *fmt* ] [ −**merge** ] [ −**s** *signal* ] [ **-shlibs** ]

**DESCRIPTION**

The **spixcounts** Shade analyzer generates one or more *spixcounts*(5sh) format files for each application that is traced. The *spixcounts* files can be used with the SpixTools commands to produce detailed execution information about an application and its shared libraries.

In addition to the standard Shade analyzer switches, the **spixcounts** analyzer analyzer accepts the following options:

−**b** *fmt*

This option specifies the names of the *spixcounts*(5sh) format output files. Since there may be several output files, the *fmt* parameter is a template that is used to construct the names of these files. See below for a definition of a file name template.

−**shlibs**

This option causes **spixcounts** to count instructions executed in the application's shared libraries. By default, only instructions in the application's main executable image are counted. The analyzer creates one *spixcounts*(5sh) format output file for each counted shared library.

−**data** *fmt*

This option causes **spixcounts** to count instructions that are not in the application's main executable image and are not in any shared library. This would include, for example, instruction residing in the application's data space and any dynamically generated instructions. The execution counts for these instructions are written to an output file with the given file name template. The format of this file is human readable text showing each instruction's disassembly, address, and execution count. This file cannot be used as input to any of the SpixTools commands.

Note, if −**shlibs** is not specified but −**data** is, instructions executed from the application's shared libraries will be reported in the −**data** file.

−**merge**

Normally, when **spixcounts** traces more than one application, each application's execution counts are reported in a separate set of files. The −**merge** switch causes **spixcounts** to combine execution counts whenever possible. Execution counts will be combined when the same application executes twice. Also, if two different application's use the same shared library, execution counts for that shared library will be combined. Note, that execution counts collected in the −**data** files are never combined.

−**s** *signal*

The −**s** switch specifies a signal number or name. When **spixcounts** receives this signal, it writes the execution count files representing the application's execution up to that point. This is useful for applications that never terminate. Once the output files are created, **spixcounts** zeros its internal counters. Thus, the signal may be sent multiple times to count the instructions executed during different phases of the application.

Specifying this switch prevents the application from receiving the given signal. Therefore, care should be taken to specify a signal that the application is not expecting. The −**s** switch also overrides any behavior specified with the −**sigdfl** switch (see *shade_intro*(1sh)).

Both the −**b** and −**data** switches require a file name template to be specified. The file name template may contain format specifies which are replaced as follows:

%p Replaced with the basename of the application program or the basename of the shared library. When used with the −**data** switch, this is always replaced with the basename of the application program.

%n Replaced with a per-command sequence number. The sequence number starts out at one and is incremented for each application that is traced. This specifier is not allowed in the −**b** file name template when the −**merge** switch is specified.

%i Replaced with the process ID of the analyzer. This specifier is not allowed in the −**b** file name template when the −**merge** switch is specified.

%% Replaced with '%'.

If no −**b** switch is specified, **spixcounts** uses the template name "%p.%n.bb" (if the −**merge** switch is not specified), or "%p.bb" (if the −**merge** switch is specified).

**THREADS**

The **spixcounts** analyzer combines the execution counts for all application LWPs (threads) together.

**FORK AND EXEC**

If the traced application forks, the **spixcounts** analyzer forks too, and each analyzer then writes its own set of output files. The execution counts reported for the child are exclusive of the counts reported for the parent. The "%i" file name template format can be used to distinguish output files generated by the parent and child.

If the application exec's a new image and the −**exec** switch is specified (see *shade_intro*(1sh)), instruction counting for the previous application ceases and instructions are counted from the new application. The new application causes the filename template's "%n" format to be incremented to the next value.

**SEE ALSO**

shade_intro(1sh), icount(1sh), ifreq(1sh), spixstats(1sh), sdas(1sh), sprint(1sh), sadd(1sh), spixcounts(5sh).

**NAME**

pairs − instruction pairs analyzer

**SYNOPSIS**

**pairs**

**addpairs**

**postpairs** [ −**t**title ] [ −**s**width**,**length ] [ −**m**[**lrtb**]margin ]

**DESCRIPTION**

The **pairs** Shade analyzer observes how frequently one type of instruction follows another, and how frequently a general purpose integer or floating point register written by the first instruction is read by the second.

The **addpairs** utility reads results (concatenated on standard input) from multiple **pairs** runs, ‘‘adds’’ them, and writes the result in the same format to standard output.

The **postpairs** utility reads **pairs** output and generates postscript for a graph of the instruction-instruction frequencies. A prologue file such as **pairs.ps** or **pairs.color.ps** must be prepended to the **postpairs** output before printing.

A title may be specified with the −**t** option. The size of the graph (in inches) may be specified with the −**s** option. Left, right, top, and bottom margins (in inches) may be specified with the −**m** option. The −**m** option effectively reduces the area specified by −**s**.

**THREADS**

The **pairs** analyzer tracks the execution of each LWP (thread) independently, but merges the statistics for all threads together in its output.

**FORK AND EXEC**

If the traced application forks, the **pairs** analyzer forks too, and each analyzer then reports its own set of statistics. The statistics reported for the child are exclusive of the statistics reported for the parent, and the output from the child is labeled with the process ID of the child process.

**FILES**

| | |
|---|---|
| $SHADE/lib/pairs.ps | monochrome *postpairs* prologue |
| $SHADE/lib/pairs.color.ps | color *postpairs* prologue |

**SEE ALSO**

shade_intro(1sh), trips(1sh).

**NAME**

trips − instruction triplets analyzer

**SYNOPSIS**

**trips** [ −**a** ]

**DESCRIPTION**

The **trips** analyzer is like **pairs**(1sh) except it looks at three instructions at a time instead of two.

Normally **trips** truncates its output after printing information for the top 90% of instruction triplets. The −**a** option causes information for all executed instruction triplets to be printed.

Like **pairs**(1sh), **trips** displays statistics by opcode group rather than by opcode.

**THREADS**

The **trips** analyzer tracks the execution of each LWP (thread) independently, but merges the statistics for all threads together in its output.

**FORK AND EXEC**

If the traced application forks, the **trips** analyzer forks too, and each analyzer then reports its own set of statistics. The statistics reported for the child are exclusive of the statistics reported for the parent, and the output from the child is labeled with the process ID of the child process.

**SEE ALSO**

shade_intro(1sh), pairs(1sh).

**NAME**

window − register window analyzer

**SYNOPSIS**

**window**

**DESCRIPTION**

The **window** Shade analyzer tracks the register window usage for one or more applications. The output includes overflow and underflow counts for different numbers of windows, save depth statistics, and save/restore run length statistics.

In the overflow/underflow table, the number of windows is given as ‘‘1+*n*’’, where *n* represents the number of windows simulated and ‘‘1+’’ signifies the extra window reserved for the trap handlers.

**THREADS**

The **window** analyzer tracks the register window usage of each LWP (thread) independently but combines the statistics for all threads in its output.

**FORK AND EXEC**

If the traced application forks, the **window** analyzer forks too, and each analyzer then reports its own set of statistics. The statistics reported for the child are exclusive of the statistics reported for the parent, and the output from the child is labeled with the process ID of the child process.

**CAVEATS**

The simulation does not take into account overflows or underflows which occur while in the kernel.

**SEE ALSO**

shade_intro(1sh).

**NAME**

cachesim5 − cache simulator

**SYNOPSIS**

**cachesim5**
[**-single-cpu**] [**-pcs**] <*cachespec*>+

**DESCRIPTION**

The **cachesim5** analyzer simulates the cache behavior for one or more applications.

**-single-cpu** option forces all threads to be simulated as though they were executed on a single CPU.

**-pcs** option provides per-cpu cache statistics for multithreaded programs.

Each *cachespec* specifies either an instruction cache **(−i ...)**, a data cache **(−d ...)**, or a combined (unified) instruction and data cache **(−u ...)**. For multilevel cache simulations, lower level (closer to CPU) caches are specified before higher level (closer to memory) caches. For each level there must be either a unified cache *cachespec*, or an instruction cache *cachespec* and a data cache *cachespec*.

The remainder of the *cachespec* specifies the cache size, block size, subblock size, set associativity, set replacement algorithm, write policy, and cache inclusion:

−{**i**|**d**|**u**}*sz***b***bsz*[**,***subbsz*][**s***ass*][**r***rep*][**wb**|**wt**][**wa**][**wi**][**cw**][**I***inc*]

The *sz*, *bsz*, and *subbsz* parameters are, respectively, the overall cache size, block size, and subblock size. Each size is specified in bytes. If the size ends with the character ‘K’, ‘M’, or ‘G’, the size is effectively multipled by, 1024, 1048576, or 1073741824. A missing subblock size implies no subblocking. A null cache (a place holder cache which always misses) is indicated by using a *sz* of 0 (no other information is expected for this cache).

The *ass* parameter is the set associativity (1 by default, i.e. direct mapped). The *rep* parameter is the set replace algorithm:

**random**
Default. Uses the built-in rand(3C) function to decide which way within a set should be replaced.

**lru**
Keeps track of accesses to each way of each set and selects the least recently used way for replacement. Minor deviation from true LRU after every 4G accesses.

**plru** Only supports 4-way associativity. Approximates lru by tracking which pair of ways (0 and 1 versus 2 and 3) was most recently used, and, for each pair, which one was most recently used. Selects the least recently used way from the least recently used pair for replacement.

**lfsr** Only supports 4-way associativity. Uses a 5-bit linear feedback shift register with a sequence length of 31 to generate a pseudo-random number that determines which way to replace.

**nvlfsr** Only supports 4-way associativity. Like lfsr except that each set (index) has a "next victim" value that determines which way will be replaced the next time; this value is initialized to 0 and then calculated from the lfsr at the end of each line fill.

**nvctr** Supports any associativity that is a power of two. Like nvlfsr except that one simple counter is used in place of the lfsr random number generator (pseudo-random behavior is achieved because this counter is shared between all sets).

The **wb** option specifies write-back (the default with write-allocate), **wt** specifies write-through (the default with no-write-allocate), and **wa** specifies write-allocate (implied by write-back). The **wi** option specifies write-invalidate (for write-through caches), and **cw** option specifies clean write back.

Higher level caches may include zero or more lower level caches. When data is invalidated (victimized) in the including cache it is back invalidated in the included cache, so that any data in the included cache is also in the including cache. The included (and any intervening) caches must be write-through. The included cache *inc* is specified as **i**, **d**, or **u** followed by the cache level (lowest level is 1).

Caches are virtually addressed. Annulled instructions cause an instruction (or unified) cache reference, but never a data cache reference. Instruction or data references which are larger than the subblock size (or block size if no subblocking) are split into multiple references as necessary.

**EXAMPLES**

Consider the following cache specification:

```
cachesim5 −i20Kb64,32s5rlruwt −d16Kb32s4rlru \
  −u4Mb128,32wbwaIi1Id1
```

This command will simulate a cache system consisting of:

i1 First level instruction cache: 20K bytes, 64 byte blocks, 32 byte subblocks 5-way set associative with LRU set replacement, write-through, no write-allocate.

d1 First level data cache: 16K bytes, 32 byte blocks, no subblocking, 4-way set associative with LRU set replacement, write-through, no write-allocate.

u2 Second level unified cache: 4M bytes, 128 byte blocks, 32 byte subblocks, direct mapped, write-back, write allocate, includes first level instruction and data caches.

Another interesting cache configuration is that of the UltraSPARC-I/II/IIi:

i1 First level instruction cache: 16K bytes, 32 byte blocks, no subblocking, 2-way set associative with random set replacement, physically indexed and physically tagged (PIPT).

d1 First level data cache: 16K bytes, 32 byte blocks, 16 byte subblocks, direct-mapped, write-through, no write-allocate, virtually indexed and physically tagged (VIPT).

u2 Second level unified cache: 64 byte blocks, direct-mapped, write-back, write-allocate, physically indexed and physically tagged (PIPT), includes first level instruction and data caches. Unified cache sizes can be: 512K, 1M, 2M, 4M, 8M, 16M.

Note that because Shade is unable to trace physical addresses, we must use virtual addressing instead of physical addressing. This example simulates an UltraSPARC-I/II/IIi cache with a 512K second level unified cache:

```
cachesim5 -i16Kb32s2 -d16Kb32,16wt -u512Kb64wbwaIi1Id1
```

**THREADS**

By default, the **cachesim5** analyzer simulates LWPs (threads) as though each thread is being executed on its own CPU. Unless the **-pcs** switch is specified, cachesim5 totals the results from all threads and prints a summary for all CPUs. The **-pcs** switch causes cachesim5 to print the statistics for each CPU separately. The **-single-cpu** switch causes the **cachesim5** analyzer to simulate all threads as though they ran on the same CPU. This switch also causes Shade itself to run on a single CPU, even if the host system is a multi-processor.

**FORK AND EXEC**

If the traced application forks, the **cachesim5** analyzer forks too, and each analyzer then reports its own set of statistics. The statistics reported for the child are exclusive of the statistics reported for the parent, and the output from the child is labeled with the process ID of the child process.

**CAVEATS**

The cache effects of flush instructions are not simulated.

**SEE ALSO**

shade_intro(1sh)

Hennessey and Patterson, "Computer Architecture: A Quantitative Approach", Chapter 5, Morgan Kaufman, 2nd edition, 1996.

**NAME**

brpred − branch predictor performance analyzer

**SYNPOSIS**

**brpred** [**-u**] [**-single-cpu**] [**-pcs**] <*brpredspec*>+

**DESCRIPTION**

The **brpred** is a Shade analyzer for quantifying the performance of the global branch history with index sharing (gshare) branch prediction scheme.

The required *brpredspec* specifies the configuration of the gshare branch prediction scheme which will be analyzed. A *brpredspec* has the following form:

−**g**<*g*>,<*n*>

Here, *g* is the number of bits of global branch history used by the gshare branch prediction scheme, and *n* specifies the size of the counters (in bits) making up each entry in the branch prediction table.

If the −**u** command-line argument is specified, then the outcomes of unconditional branches are not included in the global branch history. The default behavior is for unconditional branch outcomes to be included in the global branch history.

**-single-cpu** option forces all threads to be simulated as though they were executed on a single CPU.

**-pcs** option provides per-cpu cache statistics for multithreaded programs.

**-r** option stores youngest branch outcomes in the MSB of the global branch history.

**EXAMPLE**

brpred -g8,2

This command will model a gshare branch prediction scheme with a 256-entry branch prediction table, each entry containing a two-bit counter. An index into this table is computed by combining the eight-bit global branch history with the eight, least-significant bits of the instruction-aligned address of the conditional branch to be predicted.

**THREADS**

By default, the **brpred** analyzer simulates LWPs (threads) as though each thread is being executed on its own CPU. Unless the **-pcs** switch is specified, brpred totals the results from all threads and prints a summary for all CPUs. The **-pcs** switch causes brpred to print the statistics for each CPU separately. The **-single-cpu** switch causes the **brpred** analyzer to simulate all threads as though they ran on the same CPU. This switch also causes Shade itself to run on a single CPU, even if the host system is a multi-processor.

**FORK AND EXEC**

If the traced application forks, the **brpred** analyzer forks too, and each analyzer then reports its own set of statistics. The statistics reported for the child are exclusive of the statistics reported for the parent, and the output from the child is labeled with the process ID of the child process.

**SEE ALSO**

S. McFarling, "Combining Branch Predictors." WRL Technical Note TN-36, DEC Western Research Laboratory, (June 1993).

**BUGS**

**NAME**

hist − shade tool to print an application's most recent instructions

**SYNOPSIS**

**hist** [ −**exit** ] [ −**pc** address[:count]] [ −**ea[rw]** {B|H|W}address[:count]] [ −**signal** signal] [ −**o** filename] [ −**num** number] [ −**log** filename] [ −**stdenv** ] [ −**notrace** ] [ −**traceafter** number] [ −**tracepc** address[:count]] [-N] -- application

**DESCRIPTION**

The **hist** Shade analyzer maintains a trace history of an applications most recently executed instructions. The trace history is printed when the application causes any of several user-definable events to occur. This produces a history of instructions leading up to that event.

The following options allow you to choose when the trace history is printed. More than one of these options may be specified, causing the trace history to be dumped when any of the chosen events occurs.

−**exit**

Causes the trace history to be dumped when the application exits. In this case the trace is a history of the application's final instructions.

−**pc** <address>[:<count>]

Causes the trace history to be dumped immediately after the application executes the instruction at the given address. If 'count' is not specified, the history is dumped each time the application reaches this address. If 'count' is specified, the history is dumped only when the application reaches the address 'count' times.

−**ea[rw]** {B|H|W}<address>[:<count>]

Causes the trace history to be dumped immediately after the application read or writes the memory at the given address(r after -ea means read only, w - write only). Executing an instruction at this address does not count as a read. Use **-pc** for that. One of the characters B, H, or W must precede the address indicating either a byte, half-word (2 byte), or word (4 byte) range of addresses. If 'count' is not specified, the history is dumped each time the application reads or write this address. If 'count' is specified, the history is dumped only when the application reads or writes the address

−**signal** <signal>

Causes the trace history to be dumped whenever the analyzer receives the given signal. The signal may be specified either by its name or its number. Note, the analyzer must fill its internal trace buffer before it can dump the trace history. Therefore, there may be a delay between sending the signal and dumping the trace history.

In addition, the trace history may also be printed from within the debugger if you run this tool under dbx. This is useful should Shade terminate prematurely (because of a seg fault, for example). From within the debugger, you must call the function 'dbx_complete()' with a single parameter. The parameter must be the address of the next available trace record in Shade's internal trace buffer. While Shade is executing translated code, this value is usually located at the 32-bit memory location referenced by %i2.

This tool also accepts the following switches which affect the collection of the trace history.

−**o** <filename>

Normally, the trace is printed to stdout. However, you can direct the output to a file by using the **-o** switch.

−**num** <number>

Specifies the size of the tool's internal history buffer. A larger buffer results in more

instructions in the tool's trace output. The exact number of instructions in the trace depends both on the size of the buffer and on the mix of instructions the application executes. (Some instructions require more buffer space than others.) You may not use the −**log** option if you specify −**num**.

−**log** <filename>
Specifies a file to use as the tool's internal history buffer. The file must exist, and its size determines the size of the internal buffer. As with −**num**, a larger file results in more instructions in the tool's trace output. You may not use the −**num** option if you specify −**log**.

−**stdenv**
Causes the application to be run with a fixed, standard set of environment strings. This avoids minor differences in application execution due to differences in the environment strings. If this switch is not specified, the application is run with the current set of environment strings.

This tool also accepts the following switch to specify when to start collecting trace history. By default history is collected starting from the application's first instruction. Disabling trace history collection for part of the run causes the tool to run faster. Even when trace history collection is disabled, the tool accurately counts the number of executed instructions.

−**notrace**
Initially disable the collection of trace history.

−**traceafter** <number>
Start collecting trace history after the application has executed the given number of instructions. This option implies −**notrace**.

−**tracepc** <address>[:<count>]
Start collecting trace history immediately after executing the instruction at the given address. If <count> is specified, start collecting trace history after executing the instruction at the address <count> times. This option implies −**notrace**.

You may specify at most one −**traceafter** or −**tracepc** switch. It is illegal to combine the two.

-N
Use N-record long trace buffer. 0 < N <= NTRBUF.

In addition, the collection of trace history can be enabled and disabled from within the debugger if you run this tool under dbx. From within the debugger you may call 'dbx_starttrace()' to start the collection of trace history, or call 'dbx_stoptrace()' to stop the collection of trace history. Neither of these functions takes any parameters.

**OUTPUT**

The output of this tool is human-readable text. The output may contain more than one trace history, depending on the number of traceable events that occurred during the application's execution. Each trace history starts with a short banner indicating the total number of instructions executed by the application so far and the number of instructions not displayed since the last trace history.

Following the banner is a history of instructions leading up the traced event. There is one instruction per line, with the oldest instruction printed first and the traced instruction printed last. Each line contains the instruction's PC, a disassembly of the instruction, and a list of resources modified by the instruction.

If the instruction modified a register, the line contains a record of the form $rn=value, indicating that the given value was written to the given register. If the instruction modifies memory, the line contains a record of the form {B|H|W}address=value, indicating that the given value was written to the given memory location (byte, halfword, or word). If the instruction reads memory, the line contains a record of the form ({B|H|W}address). Lines may contains several records if the instruction modifies multiple registers or memory locations.

Trap instructions are treated somewhat specially. Any register modified as a result of the trap is listed on the instruction's line. However, memory locations written or read as a result of a trap are not listed. Lines for trap instructions contain the string "WARN_TRAP" to remind you of this.

**SEE ALSO**

shade_intro(1sh).

**NAME**

shade_anal, shade_fp, shade_ego, shade_usage, shade_error, shadeuser_initialize, shadeuser_analyze, shadeuser_report, shadeuser_terminate, shadeuser_analusage, shadeuser_analversion − Common Shade analyzer interface and functions that must be defined by user to use the interface.

**SYNOPSIS**

**cc** [ *flag* . . . ] *file* . . . **libshade.a** [ *library* . . . ]

#include <shade_anal.h>

**extern FILE** ∗**shade_fp;**

**const char** ∗**shade_ego(void);**

**void shade_usage(void);**

**void shade_error(const char** ∗*format*, /∗ *args* ∗/ **...);**

**ANALYZER DEFINES**

**extern const char shadeuser_analversion[];**

**int shadeuser_initialize(int** *argc*, **char** ∗∗*argv*, **char** ∗∗*envp*);

**int shadeuser_analyze(void);**

**int shadeuser_terminate(int** *ret*);

**void shadeuser_report(int reason, void∗ data);**

**void shadeuser_analusage(void);**

**DESCRIPTION**

The **shade_anal.o** object contains an optional interface that Shade analyzers can use. Analyzers that use this interface must link **shade_anal.o** before **libshade.a** and must not define the **shade_main(3sh)** function. Instead, such analyzers must define the five interfaces listed above. This optional interface simplifies many analyzers because it provides some common command line options and automatically loads any application(s) the user specifies into Shade.

The **shade_anal.o** object parses the command line arguments before calling the analyzer. It interprets any arguments with the following names implementing them as defined on **shade_intro(1sh)**: **−U**, **−V**, **−exec**, **−o**, **−t**, **+t**, **−v**, **−−**, and **−c**. It then passes any remaining parameters to the **shadeuser_initialize( )** function, which the analyzer defines. Typically, the **shadeuser_initialize( )** function parses any additional arguments, and then sets up the appropriate trace control parameters for the analyzer. When **shadeuser_initialize( )** returns, the interface iteratively loads each application specified by the user and calls the analyzer's **shadeuser_analyze( )** function. This function should run and trace the application. To print results after any or all applications exit or when the analyzer receives signal specified for miscellanous report (−r option), user should use **shadeuser_report( )** function. When all applications have been traced, the interface calls the analyzer's **shadeuser_terminate( )** function to clean up.

The analyzer must also define the the **shadeuser_analversion[ ]** character string to identify the analyzer's name and version level. Finally, the analyzer must define **shadeuser_analusage( )**. This function should print a usage statement to stderr for any analyzer specific command line options.

If the analyzer prints any output, it should print it to the **shade_fp** file stream. The interface defines this to reference either stdout or an output file, as directed by the user. If the analyzer must print an error message, it should do so by calling **shade_error( )**, which has an interface similar to **printf(3s)**. The analyzer may also call **shade_ego( )** to retrieve the name of the Shade analyzer and can call **shade_usage( )** to print a usage statement to stderr.

If the **shadeuser_initialize( )** function detects an error (for example, with the remaining command line parameters), it should issue an appropriate error message and then return a non-zero value. This causes the interface to exit using the returned value as an exit code. If **shadeuser_initialize( )** does not detect an error, it should return zero.

The interface calls **shadeuser_analyze( )** once for each application. If the analyzer detects an error, it should issue a message and return a non-zero value. This causes the interface to ignore any remaining applications and immediately call the analyzer's **shadeuser_terminate( )** function. If **shadeuser_analyze( )** does not detect an error, it should return zero.

When **shadeuser_analyze( )** exits, the interface calls **shadeuser_report( )** to report any results. Please see example analyzers to see how this technique works.

When the interface calls **shadeuser_terminate( )**, it passes the return value from the last call to **shadeuser_analyze( )** or the value one (1) if it was unable to load the last application. If all applications were loaded and analyzed without error, it passes zero to **shadeuser_terminate( )**. The value returned by **shadeuser_terminate( )** becomes the analyzer's exit code. Typically, analyzers should return zero to indicate success and non-zero to indicate an error.

**SEE ALSO**

shade_intro(1sh), shade_main(3sh).

**NAME**

shade_appname, shade_interpname, shade_appbase, shade_interpbase − Retrieve information about Shade application

**SYNOPSIS**

#include <shade.h>

**const char *shade_appname(void);**

**const char *shade_interpname(void);**

**spix_addr_t shade_appbase(void);**

**spix_addr_t shade_interpbase(void);**

**DESCRIPTION**

These functions all return information about the application currently loaded under Shade. If there is no application loaded, they return NULL or zero as appropriate.

The **shade_appname( )** function returns the pathname of the application. The pathname may be either absolute or relative to the directory that was current when the analyzer called **shade_load(3sh)** or **shade_loadp(3sh)**. If the application has an interpreter (eg. all dynamically linked applications on Solaris use /usr/lib/ld.so.1 as an interpreter), the **shade_interpname( )** function returns the pathname of the interpreter. If the application has no interpreter, this function returns NULL. Note, the interpreter returned by this function is not related to the shell application used to interpret shell scripts.

The **shade_appbase( )** function returns the base address of the application. This is the application address that corresponds to the first mapped location of the application's executable file. Like all application addresses, the analyzer may add this base address to the value returned from **shade_bench_memory(3sh)** to yield the host address corresponding to the first mapped location in the application's executable file. If the application has an interpreter, the **shade_interpbase( )** function returns the base address of the interpreter. If the application has no interpreter, this function returns zero.

**SEE ALSO**

shade_load(3sh), shade_bench_memory(3sh).

**NAME**

shade_appstatus − Return status of application running under Shade

**SYNOPSIS**

#include <shade.h>

**shade_status_t shade_appstatus(int** ∗*pcode***);**

**DESCRIPTION**

The **shade_appstatus( )** function returns the status of any application currently loaded under Shade. It returns one of the following values.

**SHADE_STATUS_NOAPP**

This value indicates that there no application is currently loaded under Shade.

**SHADE_STATUS_LOADED**

This value indicates that an application has been loaded, but **shade_run(3sh)** has not yet been called. If the option **SHADE_OPT_EXECTRACE** has been specified via **shade_setopt(3sh)**, this status value could also indicate that the last call to **shade_run(3sh)** returned zero because the application has loaded a new image via an **exec(2)** system calls. Once **shade_run(3sh)** is called again, the status value will change back to **SHADE_STATUS_RUNNING**.

**SHADE_STATUS_RUNNING**

This value indicates that the analyzer has loaded an application and has called **shade_run(3sh)**, and the application has not yet terminated.

**SHADE_STATUS_FORKED**

This value can only be returned if the **SHADE_OPT_FORKNOTIFY** option has been specified via **shade_setopt(3sh)**. When returned, this value indicates that the last call to **shade_run(3sh)** returned zero because the application (and the Shade process) forked a new copy of itself. Once the analyzer calls **shade_run(3sh)** again, the status value will change back to **SHADE_STATUS_RUNNING**.

**SHADE_STATUS_EXITED**

This value indicates that the last application the analyzer loaded has terminated normally. The variable ∗*pcode* is set to the application's exit code.

**SHADE_STATUS_SIGNALED**

This value indicates that the last application the analyzer loaded has terminated due to receipt of an unhandled signal. The variable ∗*pcode* is set to the signal number.

**SEE ALSO**

shade_setopt(3sh), shade_run(3sh).

**NAME**

shade_bench_memory − Return Shade application's base memory address

**SYNOPSIS**

#include <shade.h>

**char *shade_bench_memory(void);**

**DESCRIPTION**

The **shade_bench_memory( )** function returns the base memory address of the application loaded under Shade. The analyzer can add this value to any address in the application to yield the corresponding address on the host. The following example shows how this function can be used in a user-defined trace function (enabled via **shade_trfun(3sh)**) to obtain the contents of memory prior to executing an application store instruction.

```
void pre_store(
        shade_trace_t * ptrace,
        shade_regs_t *  pregs)
{
        unsigned *      pval;
        unsigned        val;

        pval = (unsigned *)(ptrace->tr_ea + shade_bench_memory());
        val = *pval;
}
```

The value returned from **shade_bench_memory( )** remains constant for the entire time that an application is loaded under Shade, with one exception. If the analyzer enables the **SHADE_OPT_EXECTRACE** option via **shade_setopt(3sh)** and the application execs a new executable image, Shade may choose a new base memory address for the new executable image.

Ordinarily, Shade chooses a convenient base address for the application when it is loaded. However, the user may specify a base address with the −**benchmem** command line switch. See **shade_intro(1sh)** for details.

**SEE ALSO**

shade_load(3sh), shade_setopt(3sh), shade_intro(1sh).

**NAME**

shade_getopt, shade_getoptv − Functions for parsing command line and Shade options.

**SYNOPSIS**

**cc** [ *flag* . . . ] *file* . . . **libshgetopt.a** [ *library* . . . ]

#include <shgetopt.h>

**int shade_getopt(const char*** *str***, const shade_options_t*** *opts***, shade_option_val_t*** *value***);**

**int shade_getoptv(int** *argc***, char*** *argv[]***, int*** *optind***, int*** *n_undef_item***, const shade_options_t*** *opts***, shade_option_val_t*** *value***);**

**DESCRIPTION**

These functions identify an option's name, parse parameter and thus find the corresponding element in the array *opts* of specific structures. Each structure contains the following data: option's name, help information, which is printed when parsing '-h' key of analyzer, pointer to suboptions structure, parameter's type (if suboptions exist, parameter must have string type).

The function **shade_getopt** parses string *str* which contains an option (if this option must have a parameter, the string must also contain this parameter).

The function **shade_getoptv** parses the command line parameter with index *optind* instead of *str* in **shade_getopt**. The option's value may be disposed in the next command line parameter. In this case *optind* is incremented by 1. All command line parameters which are not options (the first symbol is nor '-' neither '+') are placed to the beginning of command line parameters list in the sameorder. The variable *n_undef_item* means the number of already parsed command line parameters which are not options.

**RETURN VALUE**

On success both functions save the parameter in union *value* and return index of element from the array *opts*. Otherwise, they return -1 and save information about error in *value*.

**ERRORS**

The following errors are possible.

**GETOPT_NO_PARAM_ERR**

The option is identified, while its parameter is not. In this case index of the corresponding structure is contained in error information.

**GETOPT_INVALID_OPTION_ERR**

The option is not identified.

**SEE ALSO**

shade_print_opt_info(3sh).

**NAME**

shade_io, shade_bench_open, shade_bench_close, shade_bench_dup2 − Manipulate application I/O in Shade

**SYNOPSIS**

#include <shade.h>

**int shade_bench_open(const char** **path*, **int** *oflag*, **mode_t** *mode*);

**int shade_bench_close(int** *fd*);

**int shade_bench_dup2(int** *fd1*, **int** *fd2*);

**DESCRIPTION**

These functions allow the analyzer to perform some I/O operations on behalf of the application. Typically, an analyzer would use these functions to redirect an application's I/O immediately after loading it with **shade_load(3sh)** and before running it with **shade_run(3sh)**. However, they may be used at any time after an application is loaded.

The file descriptors returned and accepted by these functions correspond to file descriptors in the application. They do not necessarily refer to valid file descriptors in the analyzer. The analyzer must be careful not to use application file descriptors in analyzer I/O operations and vice versa.

The operation of these functions is identical to that of **open(2)**, **close(2)**, and **dup2(3c)**.

**SEE ALSO**

open(2), close(2), dup2(3c), shade_load(3sh), shade_shell(3sh).

**NAME**

shade_iset, shade_iset_newclass, shade_iset_newtype, shade_iset_newop, shade_iset_newcopy, shade_iset_free, shade_iset_addclass, shade_iset_addtype, shade_iset_addop − Manage sets of instructions for Shade

**SYNOPSIS**

#include <shade.h>

**shade_iset_t** ∗**shade_iset_newclass(shade_iclass_t** *iclass***, . . .);**

**shade_iset_t** ∗**shade_iset_newcopy(shade_iset_t** ∗*piset***);**

**void shade_iset_free(shade_iset_t** ∗*piset***);**

**shade_iset_t** ∗**shade_iset_addclass(shade_iset_t** ∗ **piset , shade_iclass_t** *iclass***, . . .);**

.

#include <shade_ARCH.h>

**shade_iset_t** ∗**shade_iset_newop(spix_ARCH_iop_t** *iop***, . . .);**

**shade_iset_t** ∗**shade_iset_addop(shade_iset_t** ∗*piset***, spix_ARCH_iop_t** *iop***, . . .);**

**shade_iset_t** ∗**shade_iset_newtype(spix_ARCH_itype_t** *itype***, . . .);**

**shade_iset_t** ∗**shade_iset_addtype(shade_iset_t** ∗*piset***, spix_ARCH_itype_t** *itype***, . . .);**

**DESCRIPTION**

These functions allow a Shade analyzer to manage sets of instructions. The <**shade.h**> header defines interfaces for these functions that are portable from one target architecture to another. Analyzers that use these functions may construct sets of instructions in an architecture independent manner. The architecture specific header file (eg. <**shade_sparcv9.h**>) defines additional interfaces that are specific to a particular target architecture. Analyzers that use these functions may construct sets of instructions specific to that processor. Regardless of how the analyzer constructs a set of instructions, it may use the set to enable tracing via the **shade_trctl(3sh)** or **shade_trfun(3sh)** functions.

The **shade_iset_newclass( )** function creates an instruction set that represents the given architecture independent classes of instructions. Its variable lengthed list of instruction class codes is terminated by a shade_iclass_t value of -1, namely (shade_iclass_t)(-1). Following are the possible values for *iclass*. See the descriptions of the associated architecture dependent **ITYPE** values on *spix_ARCH_iop_istype(3sh)* (eg. *spix_sparc_iop_istype(3sh)*) for an exact description of the instructions included in each of these classes.

**SHADE_ICLASS_ANY**
Selects all instructions.

**SHADE_ICLASS_IWSTART**
Selects all instructions that start an instruction word. (Meaningful only for VLIW target architectures.)

**SHADE_ICLASS_FP**
Selects all floating point instructions.

**SHADE_ICLASS_LOAD**
Selects all instructions that load a value from memory.

**SHADE_ICLASS_USTORE**
Selects all instructions that unconditionally store a value to memory.

**SHADE_ICLASS_CSTORE**
Selects all instructions that conditionally store a value to memory. Typically, this includes atomic synchronization instructions such as compare-and-swap and store-conditional.

**SHADE_ICLASS_BRANCH**
Selects all branch instructions.

**SHADE_ICLASS_UBRANCH**
Selects all unconditional branch instructions.

**SHADE_ICLASS_CBRANCH**
Selects all conditional branch instructions.

**SHADE_ICLASS_TRAP**
Selects all instructions that explicitly trap to privileged code. Typically, this includes instructions an application uses to request system services, but does not include instructions that trap due to, say, a page fault.

The **shade_iset_newcopy( )** function creates a new instruction set that is a copy of the given set. The **shade_iset_addclass( )** function adds new classes of instructions to an existing set. Like **shade_iset_newclass( )**, its variable lengthed list of instruction class codes is terminated by a shade_iclass_t value of -1. The **shade_iset_free( )** function destroys an instruction set. However, it is often not needed since the **shade_trctl(3sh)** and **shade_trfun(3sh)** functions implicitly destroy their instruction set parameter.

The remaining iset manipulation functions are available only to analyzers using the target architecture dependent interfaces. The **shade_iset_newtype( )** and **shade_iset_addtype( )** functions work like **shade_iset_newclass( )** and **shade_iset_addclass( )**, but operate using architecture dependent instruction types. See the description of *spix_ARCH_iop_istype(3sh)* (eg. *spix_sparc_iop_istype(3sh)*) for a list of possible *itype* values.

The **shade_iset_newop( )** and **shade_iset_addop( )** functions are also similar to **shade_iset_newclass( )** and **shade_iset_addclass( )**, except they use instruction opcode values. See the **spix** library instruction opcode value header (eg. <**spix_sparc_iop.h**>) for a list of valid opcode values.

**RETURN VALUES**

The **shade_iset_newclass( )**, **shade_iset_newcopy( )**, **shade_iset_newtype( )**, and **shade_iset_newop( )** functions return a pointer to the newly constructed instruction set. The **shade_iset_addclass( )**, **shade_iset_addtype( )**, and **shade_iset_addop( )** functions return a pointer to their input instruction set.

If an invalid value is passed to **shade_iset_newclass( )**, **shade_iset_newtype( )**, **shade_iset_newop( )**, **shade_iset_addclass( )**, **shade_iset_addtype( )**, or **shade_iset_addop( )**; the function issues a diagnostic message and returns NULL.

**SEE ALSO**

shade_trctl(3sh), shade_trfun(3sh), shade_tset(3sh), spix_ARCH_iop_istype(3sh).

**NAME**

shade_load, shade_loadp, shade_unload − Load an application under Shade

**SYNOPSIS**

#include <shade.h>

**int shade_load(const char** ∗*path*, **char** ∗ **const** ∗*argv*, **char** ∗ **const** ∗*envp*);

**int shade_loadp(const char** ∗*file*, **char** ∗ **const** ∗*argv*, **char** ∗ **const** ∗*envp*);

**void shade_unload(void);**

**DESCRIPTION**

The **shade_load( )** and **shade_loadp( )** functions load a new application under Shade. Both provide the new application with a set of command line arguments, *argv*, and environment strings, *envp*. Any application previously loaded by these functions is automatically unloaded. The **shade_unload( )** function unloads any previously loaded application, thus deallocating resources on the host.

The function **shade_load( )** accepts a pathname for the application, while **shade_loadp( )** accepts a file name and searches a path list to find the application. If the environment variable **SHADE_BENCH_PATH** is defined, **shade_loadp( )** uses it as the search path. Otherwise, it uses the **PATH** environment variable.

Nearly any type of application can be loaded using these functions. The application may be statically linked, dynamically linked, or may even be a shell script. If the application is dynamically linked, Shade loads the application and the dynamic loader (eg. /usr/lib/ld.so.1) and prepares to trace the dynamic loader as it loads the application's shared libraries. If the application is a shell script, Shade loads the appropriate shell application (eg. /bin/sh) and automatically initializes it with the name of the script file. For security reasons, applications with setuid or setgid privileges cannot be traced with Shade, unless the user or group ID of the application are that of the user running Shade.

When a new application is loaded under Shade, its initial set of open file descriptors are the same as the descriptors that were initially open when the Shade analyzer first started. Typically, this means that stdin, stdout, and stderr are initially open for the application. The analyzer may adjust this initial set of open file descriptors (for example, to redirect input or output) by using the functions described in **shade_io(3sh)** after loading the application.

The application's initial set of blocked signals is also inherited from the set of signals initially blocked when the analyzer first starts. See the functions described in **shade_signal(3sh)** for more information about signal handling.

Neither of these functions may be called from within a user-defined trace function (enabled via **shade_trfun(3sh)**). Attempting to do so results in a diagnostic message.

**RETURN VALUES**

The **shade_load( )** and **shade_loadp( )** functions return zero if they successfully load the specified application. They both issue a diagnostic and return -1 if they are unable to load the application.

**SEE ALSO**

shade_splitargs(3sh), shade_appname(3sh), shade_io(3sh), shade_signal(3sh).

**NAME**

shade_lock, shade_lock_new, shade_lock_delete, shade_lock_set, shade_lock_clr − Lock critical regions of Shade code

**SYNOPSIS**

#include <shade.h>

**shade_lock_t** ∗**shade_lock_new(void);**

**void shade_lock_delete(shade_lock_t** ∗*plock*);

**void shade_lock_set(shade_lock_t** ∗*plock*);

**void shade_lock_clr(shade_lock_t** ∗*plock*);

**DESCRIPTION**

These functions provide a way for Shade analyzers to lock critical regions of code in user-defined trace functions (enabled via **shade_trfun(3sh)**). Since the only code in a Shade analyzer that executes concurrently is the user-defined trace functions, only this code need make use of these functions. However, it is not harmful to call them from other parts of the analyzer.

The **shade_lock_new( )** function allocates and initializes a new lock variable. Typically, the analyzer calls this function from its initialization code. The **shade_lock_delete( )** function destroys a lock variable, which the analyzer should do once it no longer needs the lock.

Once the analyzer allocates a lock variable, it may call **shade_lock_set( )** to gain exclusive access to the lock. It should later call **shade_lock_clr( )** to release its hold on the lock.

Note, Shade analyzer may not link against the system threads library, so they are not able to use the locking primitives there. If a Shade analyzer requires locking, it should use these functions or those defined in **shade_rwlock(3sh)**.

**RETURN VALUES**

The **shade_lock_new( )** function returns a pointer to the newly allocated lock.

**SEE ALSO**

shade_trfun(3sh), shade_rwlock(3sh), shade_malloc(3sh).

**NAME**

shade_malloc, shade_calloc, shade_realloc, shade_free − Safe memory allocation for Shade

**SYNOPSIS**

#include <shade.h>

**void** ∗**shade_malloc(size_t** *size***);**

**void** ∗**shade_calloc(size_t** *nelem***, size_t** *elsize***);**

**void** ∗**shade_realloc(void** ∗*ptr***, size_t** *size***);**

**void shade_free(void** ∗*ptr***);**

**DESCRIPTION**

These functions provide a thread-safe way for Shade analyzers to allocate memory safely inside of user-defined trace functions (enabled via **shade_trfun(3sh)**). Since the only code in a Shade analyzer that executes concurrently is the user-defined trace functions, only this code need make use of these functions. However, it is not harmful to allocate memory from other parts of the analyzer with these functions.

Shade analyzers should not allocate memory using the standard C library interfaces (eg. **malloc(3c)**) because these routines are not thread safe in Shade. Analyzers should also avoid calling C library routines that allocate memory implicitly, such as **strdup(3s)**.

**RETURN VALUES**

If there is no available memory, **shade_malloc( )**, **shade_calloc( )**, and **shade_realloc( )** return a NULL pointer. Otherwise, they return a pointer to the newly allocated memory.

**SEE ALSO**

shade_trfun(3sh), shade_lock(3sh). shade_rwlock(3sh),

**NAME**

shade_print_opt_info − Function for printing help information.

**SYNOPSIS**

**cc** [ *flag* . . . ] *file* . . . **libshgetopt.a** [ *library* . . . ]

#include <shgetopt.h>

**void shade_print_opt_info(char*** *anal_info***, const shade_options_t*** *opts***, int** *info_indent***, int** *opts_type***);**

**DESCRIPTION**

The function prints help information about all options of given structure *opts*. If the options of this structure have suboptions, their help information is also printed. The variable *anal_info* can be used for the description of an analyzer. If variable *opts_type* has the value **IS_COMMON_OPTS**, value of the variable *anal_info* is printed. If variable *opts_type* has the value **IS_SPECIFIC_OPTS**, fields of **opts** which contain help information are printed, while *anal_info* is not being printed. Variable *info_indent* specifies the initial indent.

**SEE ALSO**

shade_getopt(3sh), shade_getoptv(3sh).

**NAME**

shade_run − Run and trace application under Shade

**SYNOPSIS**

#include <shade.h>

**unsigned shade_run(shade_trace_t** ∗*ptrace*, **unsigned** *ntrace*);

**DESCRIPTION**

The **shade_run( )** function executes the application loaded under Shade and collects any requested trace information. The *ptrace* parameter specifies the start of a buffer of *ntrace* trace records into which Shade stores trace information from the application. The **shade_run( )** call returns when the trace buffer fills up or when the application terminates. When it returns due to a full trace buffer, subsequent calls continue execution of the application from where the last call left off. Thus, an analyzer can repeatedly call **shade_run( )** to execute and trace the entire application.

When the **shade_run( )** function returns due to a full trace buffer, it returns the number of trace records written to the buffer. This number may not be exactly *ntrace* due to the way Shade writes to records in the buffer. The **shade_run( )** function may also return with the value zero to indicate one of several possible special circumstances. The analyzer can distinguish these cases by calling the **shade_appstatus(3sh)** function.

The **shade_run( )** function will return zero after the application has terminated and there are no more trace records to report. The **shade_appstatus(3sh)** function returns either **SHADE_STATUS_EXITED** or **SHADE_STATUS_SIGNALED** to indicate this case.

The **shade_run( )** function will return zero if the application is about to exec a new executable image and the analyzer had previously set the **SHADE_OPT_EXECTRACE** option with **shade_setopt(3sh)**. The **shade_appstatus(3sh)** function returns **SHADE_STATUS_LOADED** to indicate this case. The analyzer's next call to **shade_run( )** will return trace records from the new application image.

The **shade_run( )** function will return zero if the application forks a new process and the analyzer had previously set the **SHADE_OPT_FORKNOTIFY** option with **shade_setopt(3sh)**. When this occurs, Shade first reports all pending trace records to the analyzer and then forks a new process. Both the parent and child processes then return zero from their next calls to **shade_run( )**. The next call to **shade_run( )** in the parent process will return trace records from the application parent process, and the next call to **shade_run( )** in the child process will return trace records from the application child process. The **shade_appstatus(3sh)** function returns **SHADE_STATUS_FORKED** (in both the parent and child) to indicate this case. The analyzer can distinguish parent from child by calling **getpid(2)**.

**RETURN VALUES**

The **shade_run( )** function either returns the number of trace records written to *ptrace* or zero to indicate one of the special circumstances listed above.

**SEE ALSO**

shade_setopt(3sh), shade_appstatus(3sh), shade_trctl(3sh), shade_trfun(3sh).

**NAME**

shade_rwlock, shade_rwlock_new, shade_rwlock_delete, shade_rwlock_rdset, shade_rwlock_wrset, shade_rwlock_clr − Lock critical regions of Shade code

**SYNOPSIS**

#include <shade.h>

**shade_rwlock_t** ∗**shade_rwlock_new(void);**

**void shade_rwlock_delete(shade_rwlock_t** ∗*prwlock*);

**void shade_rwlock_rdset(shade_rwlock_t** ∗*prwlock*);

**void shade_rwlock_wrset(shade_rwlock_t** ∗*prwlock*);

**void shade_rwlock_clr(shade_rwlock_t** ∗*prwlock*);

**DESCRIPTION**

These functions provide a way for Shade analyzers to lock critical regions of code using reader-writer semantics. These functions are like those defined in **shade_lock(3sh)** except these allow either shared or exclusive access to the lock variable. Typically, an analyzer only needs to lock code in user-defined trace functions (enabled via **shade_trfun(3sh)**). However, it is not harmful to lock other sections of code.

The **shade_rwlock_new( )** function allocates and initializes a new reader-writer lock variable. Typically, the analyzer calls this function from its initialization code. The **shade_rwlock_delete( )** function destroys a reader-writer lock variable, which the analyzer should do once it no longer needs the lock.

Once the analyzer allocates a reader-writer lock variable, it may call either **shade_rwlock_rdset( )** or **shade_rwlock_wrset( )** to acquire the lock. The former obtains shared "reader" access to the lock. The later obtains exclusive "writer" access to the lock. A lock may have multiple simultaneous readers. However, once a writer has obtained the lock, no other writers or readers may gain access until the writer releases the lock. Regardless of how a thread obtains a lock, it may release it by calling **shade_rwlock_clr( )**.

**RETURN VALUES**

The **shade_rwlock_new( )** function returns a pointer to the newly allocated lock.

**SEE ALSO**

shade_trfun(3sh), shade_lock(3sh), shade_malloc(3sh).

**NAME**

shade_setopt − Enable Shade options

**SYNOPSIS**

#include <shade.h>

**int shade_setopt(shade_opt_t** *opt***);**

**DESCRIPTION**

The **shade_setopt( )** function allows an analyzer to set options that affect the way Shade operates. The following options are supported.

**SHADE_OPT_EXECTRACE**

This option allows an analyzer to trace an application after it execs a new executable image. When this option is in effect, **shade_run(3sh)** returns zero after the application execs a new image. The next call to **shade_run(3sh)** will return trace records from the application's new image.

**SHADE_OPT_EXECNOTRACE**

This option disables a previous **SHADE_OPT_EXECTRACE** option, returning Shade to its default behavior. When Shade is in this default mode, it does no longer traces an application after it execs a new image. Rather, the application's **exec(2)** call is executed directly and the entire Shade process is replaced with the new image. Note, this means that all data collected by the Shade analyzer is lost when the application exec's a new image.

Since applications typically fork a new process before calling **exec(2)**, this default Shade behavior typically results in tracing the parent application, but not its child.

**SHADE_OPT_FORKNOTIFY**

This option causes the analyzer to be notified when the application forks a new process. Whenever the application calls **fork(2)**, Shade itself forks a new process. When this option is in effect, the analyzer's call to **shade_run(3sh)** (in both the parent and child processes) return zero immediately after the **fork(2)** call. This gives the analyzer a chance to react to the application's fork. Subsequent calls to **shade_run(3sh)** in the parent process return trace records from the parent, and subsequent calls to **shade_run(3sh)** in the child process return trace records from the child.

**SHADE_OPT_NOFORKNOTIFY**

This option disables a previous **SHADE_OPT_FORKNOTIFY** option, returning Shade to its default behavior. When Shade is in this default mode, it does not return zero from **shade_run(3sh)** after the application calls **fork(2)**. However, Shade still forks a new copy of itself whenever the application forks. Note that in this mode it is possible for a single call to **shade_run(3sh)** to return trace records from both the parent and child processes. It is also possible for some trace records prior to the fork to be reported to both the child and parent processes.

**RETURN VALUE**

If *opt* is not a valid option, **shade_setopt( )** issues a diagnostic message and returns -1. Otherwise, it returns zero.

**SEE ALSO**

shade_run(3sh), shade_appstatus(3sh).

**NAME**

shade_shell, shade_fshell, shade_sshell − Run application scripts in Shade

**SYNOPSIS**

#include <shade.h>

**int shade_shell( int (**anal*)(int, char **, char **, char **));**

**int shade_fshell(FILE **fp*, int (**anal*)(int, char **, char **, char **));**

**int shade_sshell(const char **pstr*, int (**anal*)(int, char **, char **, char **));**

**DESCRIPTION**

These functions allow applications to be specified in a simple command stream and traced under Shade. The **shade_shell( )** function reads a command stream from stdin. The **shade_fshell( )** function reads the command stream from the given file pointer. The **shade_sshell( )** function reads the command string from the given string. When each function encounters the name of an application, it attempts to load the application by calling **shade_loadp(3sh)** and then calls *anal*.

These functions pass four parameters to *anal*. The first is the application's argument count. The second is a pointer to the application's argument strings. The third is a pointer to the application's environment strings. The fourth is a list of I/O redirections that Shade has performed for the application.

These shell functions currently support the following features:

- quoting: \, ', and " as for *sh*(1)
- I/O redirection: <, >, 2>, and >&
- comments: from # to end of line

**RETURN VALUES**

If the *anal* function returns a non-zero value, **shade_shell**, **shade_fshell**, and **shade_sshell** return this value immediately. Otherwise, **shade_shell** and **shade_fshell** return zero when they reach the end of the file, and **shade_sshell** returns zero when it reaches the end of the string.

**SEE ALSO**

sh(1), shade_load(3sh), shade_io(3sh), shade_run(3sh).

**NAME**

shade_signal, shade_analsig, shade_sendsig − Manipulate signals in Shade

**SYNOPSIS**

#include <shade.h>

**int shade_analsig(int** *sig*, **void (**∗*handle*)**(int, siginfo_t** ∗**));**

**int shade_sendsig(int** *sig*, **const siginfo_t** ∗*psiginfo*);

**DESCRIPTION**

The **shade_analsig( )** function provides a way for the analyzer to receive instances of the signal *sig* that are sent to the Shade process. By default all signal sent to the Shade process are forwarded to the application loaded under Shade. If the given function pointer is not NULL, instances of that signal cause the analyzer to invoke that function. The signal number and any signal information are passed to it. If the function pointer is NULL, any existing analyzer signal handler is canceled for that signal, and future instances of the signal are sent to the application.

Specifying an analyzer signal handler using this function is similar to specifying a handler with **sigaction(2)** that has an empty *sa_mask* and no *sa_flags*. Thus, the handler is invoked with only the handled signal masked. As with **sigaction(2)**, attempts to handle SIGKILL or SIGSTOP are ignored.

Note, this function does not allow the analyzer to intercept signals generated by the application. These signals are always sent to the application. Only signals generated from outside the application can be intercepted via **shade_analsig( )**.

The **shade_sendsig( )** function allows the analyzer to send a signal to the application loaded under Shade. The analyzer could, for example, forward a signal to the application after intercepting it with **shade_analsig( )**. The application behaves as though the signal *sig* were sent to it from outside its own process. If *psiginfo* is not NULL, it must point to additional information about the signal. (See **siginfo(5)** for more details.) If the analyzer calls **shade_sendsig( )** when there is no application loaded, the signal is simply ignored.

Note that signals sent via **shade_sendsig( )** affect only the application running under Shade, they do not affect Shade itself. For example, sending SIGKILL will terminate the application, but will not affect the Shade analyzer.

The **shade_analsig( )** function interacts with the −**sigdfl** Shade command line switch as follows. If a signal is specified with both −**sigdfl** and **shade_analsig( )**, the **shade_analsig( )** functionality takes precedence. If the analyzer handler is later disabled by calling **shade_analsig( )** with a NULL pointer, the semantics of −**sigdfl** take over. See **shade_intro(1sh)** for details about −**sigdfl**.

**RETURN VALUES**

If an invalid signal number is passed to **shade_analsig( )** or **shade_sendsig( )**, both functions issue a diagnostic message and return -1. Otherwise, they return zero.

**SEE ALSO**

sigaction(2), siginfo(5), shade_intro(1sh).

**NAME**

shade_splitargs − Separate Shade analyzer and application arguments

**SYNOPSIS**

#include <shade.h>

**int shade_splitargs(char** ∗∗*argvin*, **char** ∗∗∗*pargvapp*, **int** ∗*pargcapp*);

**DESCRIPTION**

The **shade_splitargs( )** function provides a mechanism for separating analyzer and application argument lists. It relies on a convention followed by many Shade analyzers of marking the application arguments with the string "--". This function searches for an argument string of "--" in *argvin*. If one is found, **shade_splitargs( )** changes it to a NULL pointer, thus terminating the analyzers argument list. The function returns the remainder of the argument strings in ∗*pargvapp* and the count of the remaining argument strings in ∗*pargcapp*. Finally, **shade_splitargs( )** returns the number of analyzer arguments remaining in *argvin*. If there is no argument string "--" in *argvin*, the argument list is unchanged, zero is stored in ∗*pargcapp*, and **shade_splitargs( )** returns the original argument count.

**RETURN VALUES**

The **shade_splitargs( )** function returns the number of argument strings remaining in *argvin*.

**SEE ALSO**

shade_main(3sh).

**NAME**

shade_trange, shade_addtrange, shade_subtrange, shade_intrange, shade_argtrange, − Restrict Shade tracing by address range

**SYNOPSIS**

#include <shade.h>

**void shade_subtrange(spix_addr_t** *addrlo***, spix_addr_t** *addrhi***);**

**void shade_addtrange(spix_addr_t** *addrlo***, spix_addr_t** *addrhi***);**

**spix_bool_t shade_intrange(spix_addr_t** *addr***);**

**int shade_argtrange(const char** ∗*pstr***);**

**DESCRIPTION**

These functions work in concert with **shade_trctl(3sh)** and **shade_trfun(3sh)** to determine which instructions Shade traces. Shade traces an instruction only if it is selected by **shade_trctl(3sh)** or **shade_trfun(3sh)** and if that instruction resides in an address range selected by the **shade_trange( )** functions. By default, all addresses in the application are selected, so an analyzer need not call the **shade_trange( )** functions unless it wants to restrict the range of traced instructions.

The **shade_subtrange( )** function disables tracing for instructions residing in the given address range. The **shade_addtrange( )** function enables traces for the given range. In both cases, the range starts at *addrlo* and continues up to, but not including, *addrhi*. If *addrhi* is zero, the range continues to the end of memory. Empty ranges (i.e. *addrhi* <= *addrlo***) are silently ignored.**

The **shade_intrange( )** function returns TRUE if the given address resides in a range of instructions where tracing is enabled and returns FALSE if it does not.

The **shade_argtrange( )** function parses an argument string describing a range of addresses and calls **shade_subtrange( )** or **shade_addtrange( )** as appropriate. The string *pstr* may either be of the form +t*addrlo,addrhi* or -t*addrlo,addrhi* where *addrlo* is the low address in the range and *addrhi* is the high address. The first form causes tracing to be enabled for the address range, the second form causes tracing to be disabled. Either address may be omitted, but the comma is required. If the first address is omitted, the beginning of memory is used. If the second address is omitted, the end of memory is used.

Analyzers should note that all address are initially enabled for tracing. Therefore, calling **shade_addtrange( )** or calling **shade_argtrange( )** with a +t argument is ineffectual without first disabling tracing for all addresses.

The address ranges selected by these functions take effect the next time the analyzer calls **shade_run(3sh)**. Analyzers should not attempt to call these functions from within a user-defined trace function (enabled via the **shade_trfun(3sh)** functions). Doing so results in a diagnostic

Note, the address ranges selected by these functions do not affect the **shade_trctl_at(3sh)** or **shade_trfun_at(3sh)** functions.

**RETURN VALUES**

The **shade_intrange( )** function returns TRUE or FALSE as documented above. The **shade_argtrange( )** function returns zero if its argument string describes a valid address range. Otherwise, it returns -1 and issues a diagnostic message.

**SEE ALSO**

shade_trctl(3sh), shade_trfun(3sh), shade_run(3sh).

**NAME**

shade_trclear − Clear all Shade trace parameters

**SYNOPSIS**

#include <shade.h>

**void shade_trclear(void);**

**DESCRIPTION**

The **shade_trclear( )** function clears all trace control parameters enabled by previous calls to the **shade_trctl(3sh)** functions or to the **shade_trfun(3sh)** functions.

Analyzers should not attempt to clear the trace control parameters from within a user-defined trace function (enabled via the **shade_trfun(3sh)** functions). Doing so results in a diagnostic message.

**SEE ALSO**

shade_trctl(3sh), shade_trfun(3sh).

**NAME**

shade_trctl, shade_trctl_at − Enable tracing in Shade

**SYNOPSIS**

#include <shade.h>

**void shade_trctl(shade_iset_t** **piset*, **shade_tr_t** *tri*, **shade_tset_t** **ptset*);

**void shade_trctl_at(spix_addr_t** *addr*, **shade_iset_t** **piset*, **shade_tr_t** *tri*, **shade_tset_t** **ptset*);

**DESCRIPTION**

These functions allow a Shade analyzer to enable instruction tracing. The *piset* parameter specifies a set of instructions created with the **shade_iset(3sh)** functions. If it is not NULL, the *ptset* parameter specifies a set of trace control parameters created with the **shade_tset(3sh)** functions. If *ptset* is NULL, it designates the empty set of parameters. The *tri* parameter specifies whether the instructions should be traced when they are executed (**SHADE_TRI_EXECUTED**), annulled (**SHADE_TRI_ANNULLED**), or both (**SHADE_TRI_ISSUED**).

The **shade_trctl( )** function installs a new set of trace control parameters for the instructions described by *piset*. The parameters replace any previous parameters installed for these instructions. The **shade_trctl_at( )** function is like **shade_trctl( )** except it applies only to the instruction starting at the given target address. If the instruction at that address is specified by *piset*, it is traced according to the trace parameters in *ptset*. If **shade_trctl( )** and **shade_trctl_at( )** specify conflicting parameters, the parameters in the **shade_trctl_at( )** call prevail. Moreover, the instruction at address *addr* is evaluated dynamically, so the address need not be mapped when the analyzer calls **shade_trctl_at( )** and the tracing is automatically updated if the application dynamically changes the instruction at that address.

The behavior of these functions is tied to the trace record size registered with **shade_trsize(3sh)**. Typically, an analyzer should register a trace record size by calling **shade_trsize(3sh)** first, and then call **shade_trctl( )** or **shade_trctl_at( )** to enable tracing. Each trace parameter in *ptset* corresponds to a field in the trace record. If a parameter's field is not fully contained by the registered trace record size at the time the analyzer calls **shade_trctl( )** or **shade_trctl_at( )**, Shade silently disables the parameter.

Shade will also silently disable parameters that do not apply to instructions in *piset*. For example, if SHADE_TRCTL_EA were specified for all instructions, Shade would disable it for instructions that cannot be traced with SHADE_TRCTL_EA. See **shade_tset(3sh)** and the architecture dependent trace control parameter manpage (eg. shade_sparcv9_trctl(5sh)) for a list of which instructions can be traced by each trace parameter.

Finally, the trace parameters enabled via **shade_trctl( )** do not apply to instructions within address ranges that have been excluded with the **shade_trange(3sh)** functions. These address ranges are checked each time the analyzer calls **shade_run(3sh)**. Trace parameters enabled via **shade_trctl_at( )** are not affected by the **shade_trange(3sh)** address ranges.

The trace parameters enabled by these functions take effect the next time the analyzer calls **shade_run(3sh)**. Analyzers should not attempt to change the trace parameters from within a user-defined trace function (enabled via the **shade_trfun(3sh)** functions). Doing so results in a diagnostic message.

Both of these functions implicitly destroy the *piset* and *ptset* sets as though **shade_iset_free(3sh)** and **shade_tset_free(3sh)** had been called. Therefore, the analyzer should not reference these sets after calling the **shade_trctl( )** functions. Should the analyzer require a set to be retained, it can call **shade_iset_newcopy(3sh)** or **shade_tset_newcopy(3sh)** to duplicate the set first.

**SEE ALSO**

shade_tset(3sh), shade_iset(3sh), shade_trsize(3sh), shade_trange(3sh), shade_trfun(3sh), shade_trclear(3sh), shade_run(3sh), shade_ARCH_trctl(5sh).

**NAME**

shade_trfun, shade_trfun_at − Enable user-defined trace function in Shade

**SYNOPSIS**

#include <shade.h>

**void shade_trfun(shade_iset_t** ∗*piset*, **shade_tr_t** *tri*,
**void (**∗*prefun*)**(shade_trace_t** ∗, **const shade_regs_t** ∗),
**void (**∗*postfun*)**(shade_trace_t** ∗, **const shade_regs_t** ∗),
**void (**∗*unprefun*)**(shade_trace_t** ∗),
**void (**∗*fixpref*)**(shade_trace_t** ∗, **const shade_regs_t** ∗))

**void shade_trfun_at(spix_addr_t** *addr*, **shade_iset_t** ∗*piset*, **shade_tr_t** *tri*,
**void (**∗*prefun*)**(shade_trace_t** ∗, **const shade_regs_t** ∗),
**void (**∗*postfun*)**(shade_trace_t** ∗, **const shade_regs_t** ∗),
**void (**∗*unprefun*)**(shade_trace_t** ∗),
**void (**∗*fixpref*)**(shade_trace_t** ∗, **const shade_regs_t** ∗))

**DESCRIPTION**

These functions allow a Shade analyzer to collect customized trace information that cannot be obtained by using the **shade_trctl(3sh)** functions. The first two parameters to **shade_trfun( )** specify a set of instructions to trace. These parameters are interpreted exactly like the first two parameters to **shade_trctl(3sh)**. The next two parameters specify two user-defined trace function, *prefun*, which is called before each of the traced instructions and *postfun*, which is called after each of the traced functions. The final two parameters have a specialized meaning which is defined below.

The user-defined functions registered by **shade_trfun( )** replace any previously specified functions for these instructions. An analyzer may cancel a previously registered function by specifying a NULL function pointer.

The first parameter to each user-defined trace function is a pointer to a trace structure for the traced instruction. The second parameter points to a read-only copy of the application's register state. The function may choose to copy information from this register state into the trace structure. The register state passed to *prefun* does not reflect the execution of the traced instruction, while the state passed to *postfun* **does.**

An analyzer may choose to trace a single instruction with both **shade_trctl(3sh)** and with **shade_trfun( )**. In this event, the trace structure passed to *postfun* reflects all of the tracing done by **shade_trctl(3sh)**, however the trace structure passed to *prefun* function does not reflect the tracing that is done after the instruction executes. See **shade_tset(3sh)** and the architecture dependent trace control parameter manpage (eg. **shade_sparcv9_trctl(5sh)**) for details.

Often, the analyzer's user-defined trace function will want to store additional information in the trace record. This information can then be read when the record is returned from **shade_run(3sh)**. This technique usually requires that the analyzer define additional fields in the trace record. Any such fields must be defined after the standard fields. For example, this trace record definition adds the field tr_foo to the end of the record.

```
#include <shade.h>

struct shade_trace_s {
        SHADE_TRACE
        unsigned        tr_foo;
        unsigned        pad;
};
```

Note that the trace record may need additional padding, as in this example, in order to maintain the alignment restrictions imposed by **shade_trsize(3sh)**. An analyzer's user-defined trace function may also store custom values in any of the standard trace record field if it can determine that the field is

unused.

The **shade_trfun_at( )** function is like **shade_trfun( )** except it only applies to the instruction starting at the given target address. If the instruction at that address is specified by *piset*, the user-defined trace functions are called for that instruction as defined above. If **shade_trfun( )** and **shade_trfun_at( )** specify different functions, the functions in the **shade_trfun_at( )** call prevail. Moreover, the instruction at address *addr* is evaluated dynamically, so the address need not be mapped when the analyzer calls **shade_trfun_at( )** and the tracing is automatically updated if the application dynamically changes the instruction at that address.

The behavior of these functions with respect to the address ranges excluded via **shade_trange(3sh)** is that same as the behavior of the **shade_trctl( )** functions. Excluded addresses affect functions installed via **shade_trfun( )**, but not those installed via **shade_trfun_at( )**.

The user-defined trace functions installed by these calls take effect the next time the analyzer calls **shade_run(3sh)**. Analyzers should not attempt to install new trace functions from within an existing trace function. Doing so results in a diagnostic message.

Both of these functions implicitly destroy their *piset* parameter. Therefore, the analyzer should not reference this set after calling the **shade_trfun( )** functions. Should the analyzer require the set to be retained, it can call **shade_iset_newcopy(3sh)** to duplicate the set first.

The final two parameters to the **shade_trfun( )** calls, *unprefun* and *fixpref*, only become meaningful when tracing instructions that block for long periods of time (for example, system call traps). Tracing such instructions in multi-threaded programs becomes tricky because the trace buffer may be returned from **shade_run(3sh)** while the instruction is blocked and the analyzer may change the user-defined trace functions at this time. Analyzers that trace multi-threaded applications and change the trace control parameters via **shade_trctl(3sh)** or **shade_trfun( )** as the application runs must be aware of some subtle issues.

Consider a blocked application instruction that is traced with both *prefun* and *postfun* functions. Further consider that the analyzer's call to **shade_run(3sh)** may return after the instruction's *prefun* function is called and before its *postfun* function is called. Finally, consider that the analyzer may change the instruction's tracing at this time by calling **shade_trfun( )**. Shade ensures that **shade_run(3sh)** never returns a partially filled trace record, so the analyzer need not worry about that. However, the pending instruction's trace record has been passed to the old *prefun* function and may not be passed to the old *postfun* function because the analyzer may have unregistered that function. This could lead to problems if the *prefun* function allocates resources that it expects *postfun* to deallocate. Furthermore, the blocked instruction's trace record cannot be passed to the new *prefun* function because the instruction has already started executing. This could lead to problems when the trace record is later passed to the new *postfun* function.

The *unprefun* and *fixpref* call-back functions solve these problems. If the first set of trace parameters includes an *unprefun* function for the blocked instruction, Shade will call this function after the instruction completes. This gives the analyzer a chance to clean up any resources allocated by the *prefun* functions. Also, if the new trace parameters include a *fixpref* function for this instruction, Shade calls this function *after* the instruction completes. However, it passes a register state that does not reflect the execution of the blocked instruction. This allows the analyzer to mimic the effect of the new *prefun* function for the blocked instruction. Note, that even though the register state passed to the *fixpref* function does not reflect the execution of the blocked instruction, other resources (such as the state of memory) do reflect the instruction. Finally, Shade calls the new trace parameter's *postfun* function for the blocked instruction.

Note that, under some circumstances, Shade may copy the contents of a trace record to a new memory location. This may even happen after Shade calls *prefun* and before it calls *postfun*. Therefore, the analyzer should not store the address of a trace record while executing a trace function and expect that address to remain valid after the trace function completes.

The analyzer must define all four of these call-back functions to be safe in a multi-threaded environment. Unlike other parts of the analyzer, these functions may be called simultaneously by multiple threads of execution if the traced application is multi-threaded. If the analyzer needs to lock critical sections of code, it should use the locking primitives defined in **shade_lock(3sh)** or **shade_rwlock(3sh)**. It should not use the locking primitives in the standard threads library. The call-back functions must also be careful when calling standard C library routines. Due to the way Shade operates, the internal locking in these routines is not enabled. If a call-back function must use a standard C library routine, it should externally lock calls to it in order to ensure that only one thread at a time calls into the C library. Since dynamic memory allocation is a common operation, Shade provides safe dynamic memory allocation routines (see **shade_malloc(3sh)**) that can be safely used in call-back functions without external locking.

**SEE ALSO**

shade_tset(3sh), shade_iset(3sh), shade_trsize(3sh), shade_trange(3sh), shade_trctl(3sh), shade_trclear(3sh), shade_run(3sh), shade_lock(3sh), shade_rwlock(3sh), shade_malloc(3sh), shade_ARCH_trctl(5sh).

**NAME**

shade_trsize − Specify size of Shade trace record

**SYNOPSIS**

#include <shade.h>

**int shade_trsize(size_t** *size***);**

**DESCRIPTION**

This function specifies the size (in bytes) of the Shade analyzer's trace record. Analyzers should call **shade_trsize( )** before calling the **shade_trctl(3sh)** functions in order to tell Shade the size of a trace record. Shade uses this information to determine which trace record fields are valid.

An analyzer may specify any size that is a multiple of eight bytes. Usually, though, an analyzer defines its trace record and then uses **sizeof( )**:

```
#include <shade.h>

struct shade_trace_s {SHADE_TRACE};


int shade_main(int argc, char **argv, char **envp)
{
        shade_trsize(sizeof(shade_trace_t));
        ...
}
```

If a Shade analyzer reduces the size of its trace record after some trace control parameters have been enabled via **shade_trctl(3sh)**, some of those parameters may be disabled if the new size is too small to include the trace record fields associated with those parameters.

Analyzers should not attempt to call **shade_trsize( )** from within a user-defined trace function (enabled via the **shade_trfun(3sh)** functions). Doing so results in a diagnostic message.

**RETURN VALUES**

If the specified trace record size is a multiple of eight bytes, **shade_trsize( )** returns zero. Otherwise, it issues a diagnostic message and returns -1.

**SEE ALSO**

shade_trctl(3sh), shade_trfun(3sh).

**NAME**

shade_tset, shade_tset_new, shade_tset_newcopy, shade_tset_free, shade_tset_add − Manage sets of trace control parameters for Shade

**SYNOPSIS**

#include <shade.h>

**shade_tset_t** ∗**shade_tset_new(shade_trctl_t,** *trctl***, . . .);**

**shade_tset_t** ∗**shade_tset_newcopy(shade_tset_t** ∗*ptset***);**

**void shade_tset_free(shade_tset_t** ∗*ptset***);**

**shade_tset_t** ∗**shade_tset_add(shade_tset_t** ∗*ptset***, shade_trctl_t** *trctl***, . . .);**

**DESCRIPTION**

These functions allow a Shade analyzer to manage sets of trace control parameters. The **shade_tset_new( )** function creates a parameter set that contains the given variable lengthed list of trace control parameters. The list is terminated by a shade_trctl_t value of -1, namely (shade_trctl_t)(-1). The table below lists the possible target architecture independent trace control parameter values. See the target architecture dependent trace control parameter manpage (eg. **shade_sparcv9_trctl(5sh)**) for a list of the architecture dependent values.

The **shade_tset_add( )** function adds trace parameters to an existing set. Again, this variable lengthed parameter list is terminated by a shade_trctl_t value of -1. The **shade_tset_newcopy( )** function creates a new trace parameter set that is a copy of the given set. The **shade_tset_free( )** function destroys a trace parameter set. However, it is often not needed since the **shade_trctl(3sh)** functions implicitly destroy the trace parameter sets they are passed.

Each trace parameter that an analyzer enables causes Shade to write to a field in the trace record. It is the analyzer's responsibility to define the trace record such that it contains the appropriate fields. An architecture independent analyzer typically does this by defining a trace record that starts with **SHADE_TRACE**, for example:

```
#include <shade.h>

struct shade_trace_s {
        SHADE_TRACE
};
```

Other fields may follow **SHADE_TRACE**, but they may not precede it. An architecture dependent analyzer may define its trace record differently, see the architecture dependent trace control parameter manpage (eg. **shade_sparcv9_trctl(5sh)**) for details.

The following table lists the possible target architecture independent trace control parameter values that can be used with **shade_tset_new( )** and **shade_tset_add( )**.

**SHADE_TRCTL_PC**

Record the target PC of the traced instruction in the **tr_pc** field of the trace record.

**SHADE_TRCTL_EA**

If the traced instruction is in the **SHADE_ICLASS_LOAD**, **SHADE_ICLASS_USTORE**, or **SHADE_ICLASS_CSTORE** classes, record the effective virtual address of the memory location referenced by the instruction in the **tr_ea** field of the trace record. If the traced instruction is in the **SHADE_ICLASS_UBRANCH** or **SHADE_ICLASS_CBRANCH** classes, record the branch target address in the **tr_ea** field. If the traced instruction is in the **SHADE_ICLASS_TRAP** class, record the instruction's trap number in the **tr_ea** field.

**SHADE_TRCTL_IH**

Record the traced instruction's architecture dependent opcode value (eg. one of the spix_sparc_iop_t values) in the **tr_ih** field of the trace record.

**SHADE_TRCTL_TID**

Record the ID of the thread executing the traced instruction in the **tr_tid field of the trace record. This ID is assigned by Shade, and may not correspond to any thread ID maintained by the target operating system. Thread IDs are small consecutive integers starting with zero for the first thread, thus analyzers may use them as array indexes. Thread IDs are never reused even if the application's thread exits.**

**SHADE_TRCTL_TAKEN**

If the traced instruction is in the **SHADE_ICLASS_CBRANCH** class, record a one in the **tr_taken** field if the branch is taken or a zero if it is not taken. If the traced instruction is in the **SHADE_ICLASS_CSTORE** class, record a one in the **tr_taken** field if the store happens or a zero if it does not. If the traced instruction is in the **SHADE_ICLASS_TRAP** class, record a one in the **tr_taken** field if the trap is taken or a zero if it is not.

**SHADE_TRCTL_ANNULLED**

If the target architecture allows instructions to be annulled and this instruction was annulled, record a one in the **tr_annulled** field of the trace record. Otherwise, record a zero in this field.

**SHADE_TRCTL_IWSTART**

If the target architecture is a VLIW, record a one in the **tr_iwstart** field if this instruction is the start of an instruction word, or record a zero if the instruction does not start an instruction word. If the target architecture is not a VLIW, always record a one in this field.

**SHADE_TRCTL_STOPB**

This trace parameter does not cause any data to be recorded in the trace record. Rather, it causes the trace buffer (if it is not empty) to be reported to the analyzer before execution of the traced instruction. Execution resumes at the traced instruction when the analyzer resumes tracing the application.

**SHADE_TRCTL_STOPA**

This trace parameter does not cause any data to be recorded in the trace record. Rather, it causes the trace buffer (if it is not empty) to be reported to the analyzer after execution of the traced instruction. Execution resumes at the next instruction when the analyzer resumes tracing the application.

In a multi-threaded application, the calling thread will not trace any further instructions until the trace buffer has been reported to the analyzer. However, other threads may still trace a few instructions before the buffer is reported. Therefore, analyzers may not assume that an instruction marked with **SHADE_TRCTL_STOPA** is necessarily the last traced instruction in the trace buffer.

All of the trace collection described in the table above occurs before the traced instruction executes. Thus, if an instruction is trace with **shade_trfun(3sh)**, all the traced values described here are visible to both the *prefun* user-defined trace function and to the *postfun* user-defined trace function. See **shade_trfun(3sh)** for details.

The table above describes the architecture independent effect of the trace control parameters. A particular target architecture may enable additional tracing when these trace parameters are specified. Therefore, an analyzer may not assume that a trace record field is unused because it is not listed above. See the target architecture dependent trace control parameter manpage (eg. **shade_sparcv9_trctl(5sh)**) for details.

**RETURN VALUES**

The **shade_tset_new( )** and **shade_tset_newcopy( )** functions return a pointer to the newly constructed trace parameter set. The **shade_tset_add( )** function returns a pointer to its input trace parameter set.

If an invalid value is passed to **shade_tset_new( )** or **shade_tset_add( )**, the function issues a diagnostic message and returns NULL.

**SEE ALSO**

shade_trctl(3sh), shade_iset(3sh), shade_ARCH_trctl(5sh).

**NAME**

shade_version − Shade library version string

**SYNOPSIS**

**#include** <**shade.h**>

**const char shade_version[];**

**DESCRIPTION**

The **shade_version** character array contains a read-only string representation of the Shade library's version level.

**NAME**

shadeuser_main − Shade analyzer entry point

**SYNOPSIS**

#include <shade.h>

**int shadeuser_main(int** *argc***, char** ***argv***, char** ***envp***);**

**DESCRIPTION**

The **shadeuser_main( )** function is the user entry point for all Shade analyzers. Each analyzer must supply **shadeuser_main( )**, which is called by the library. Like **main( )** in a normal C program, the parameters to **shadeuser_main( )** specify the number of command line arguments, pointers to the command line argument strings, and pointers to the environment strings.

The Shade library automatically recognizes each command line argument named −*shade* and removes it and the following parameter from the argument list before passing the list to **shadeuser_main( )**. The −*shade* switch specifies options that are recognized directly by the Shade library. See **shade_intro(3sh)** for a list of these options.

**RETURN VALUES**

The value returned by **shadeuser_main( )** becomes the return code (exit status) for the analyzer.

**SEE ALSO**

shade_intro(1sh), shade_splitargs(3sh), shade_anal(3sh).

**NAME**

shade_sparcv9_trctl − SPARC V9 trace parameter codes

**SYNOPSIS**

**#include <shade_sparcv9.h>**

**DESCRIPTION**

The SPARC V9 dependent Shade header contains values for the **shade_trctl_t** enumerated type that enable various SPARC V9 specific trace control parameters. These values can be used to construct sets of trace control parameters using the **shade_tset(3sh)** functions. Those sets, in turn, can be used with the **shade_trctl(3sh)** functions to enable instruction tracing in Shade. This manpage documents the architecture dependent aspects of these trace control parameters. The **shade_tset(3sh)** manpage documents the architecture independent aspects.

The **shade_tset(3sh)** manpage documents a number of architecture independent trace control parameters. However, some of those parameters have additional semantics for SPARC V9 instructions. This table defines those additional semantics.

**SHADE_TRCTL_EA**

If the traced instruction is a CALL, JMPL, or RETURN; this parameter records the target address of these control transfer instructions in the **tr_ea** field of the trace record. If the traced instruction is a FLUSH, PREFETCH, or PREFETCHA; it records the target address in the **tr_ea** field.

**SHADE_TRCTL_TAKEN**

If the traced instruction is one of the conditional move instructions, this parameter records a one in the **tr_taken field if the move occurs or a zero if it does not. This tracing is performed before the instruction executes, so it is visible to any** *prefun* **user-defined trace function that may be set up via shade_trfun(3sh).**

**If the traced instruction is either CASA or CASXA (the instructions in the SHADE_ICLASS_CSTORE class), this parameter records a one in the tr_taken field if the conditional store occurs, or a zero if it does not. This tracing is performed after the instruction executes, so it is not visible to any** *postfun* **user-defined trace function that may be set up via shade_trfun(3sh).**

**SHADE_TRCTL_IWSTART**

Since SPARC is not a VLIW architecture, this parameters always causes a one to be written to the **tr_iwstart** field.

The <*shade_sparcv9.h*> header also defines a number of trace parameters that are specific to the SPARC V9 architecture. However, an analyzer must define its trace record to start with one of the following macros in order to use these parameters: **SHADE_SPARCV9_TRACE**, **SHADE_SPARCV9_REGTRACE**, or **SHADE_SPARCV9_FREGTRACE**. An analyzer should not specify more than one of these macros in the trace record definition, nor should it use any of these macros with the **SHADE_TRACE** macro. The analyzer chooses the macro it uses based on the architecture dependent information it wants to trace. An analyzer may trace any architecture independent information using any of these macros.

Analyzers whose trace records are defined to start with **SHADE_SPARCV9_TRACE** can specify the following parameter.

**SHADE_SPARCV9_TRCTL_I**

Writes the text of the traced instruction to the **tr_i** field of the trace record.

Analyzers whose trace records are defined to start with **SHADE_SPARCV9_REGTRACE** can additionally specify the following trace parameters.

**SHADE_SPARCV9_TRCTL_RS1**

If the traced instruction references an integer register in its RS1 field, write the 64-bit

contents of that register to the **tr_rs1** field of the trace record.

**SHADE_SPARCV9_TRCTL_RS2**
If the traced instruction references an integer register in its RS2 field, write the 64-bit contents of that register to the **tr_rs2** field of the trace record.

**SHADE_SPARCV9_TRCTL_RD**
If the traced instruction references an integer register in its RD field, write the 64-bit contents of that register to the **tr_rd** field of the trace record. This tracing is performed after the instruction executes, so the traced register value reflects the execution of the instruction. Furthermore, the traced value is not visible to any *postfun* user-defined trace function that may be set up via **shade_trfun(3sh)**.

Finally, analyzers whose trace records are defined to start with **SHADE_SPARCV9_FREGTRACE** can specify these trace parameters in addition to the ones listed above.

**SHADE_SPARCV9_TRCTL_FRS1**
If the traced instruction references a single, double, or quad precision floating point register in its RS1 field, write the 32-, 64-, or 128-bit contents of that register to the **tr_frs1** field of the trace record.

**SHADE_SPARCV9_TRCTL_FRS2**
If the traced instruction references a single, double, or quad precision floating point register in its RS2 field, write the 32-, 64-, or 128-bit contents of that register to the **tr_frs2** field of the trace record.

**SHADE_SPARCV9_TRCTL_FRD**
If the traced instruction references a single, double, or quad precision floating point register in its RD field, write the 32-, 64-, or 128-bit contents of that register to the **tr_frd** field of the trace record. This tracing is performed after the instruction executes, so the traced register value reflects the execution of the instruction. Furthermore, the traced value is not visible to any *postfun* user-defined trace function that may be set up via **shade_trfun(3sh)**.

In all of the tables above, the traced value is written before the traced instruction executes unless otherwise specified. Trace values that are written before the instruction executes are visible to both the *prefun* user-defined trace function and to the *postfun* user-defined trace function. See **shade_trfun(3sh)** for details.

**EXAMPLE**

The following code fragment defines a trace record and sets up suitable trace control parameters to trace the effective address of every load as well as the data that was loaded.

```
#include <shade_sparcv9.h>

struct shade_trace_s {SHADE_SPARCV9_FREGTRACE};

int shade_main(int argc, char **argv, char **envp)
{
        shade_iset_t *  piset;
        shade_tset_t *  ptset;

        shade_trsize(sizeof(shade_trace_t));

        piset = shade_iset_newclass(SHADE_ICLASS_LOAD, -1);
        ptset = shade_tset_new(SHADE_TRCTL_EA, SHADE_SPARCV9_TRCTL_RD,
            SHADE_SPARCV9_TRCTL_FRD, -1);
        shade_trctl(piset, SHADE_TRI_EXECUTED, ptset);

        ...
}
```

**SEE ALSO**

shade_tset(3sh).