# Shade
# User's Manual

V5.33A

Sun Microsystems Laboratories, Inc.
Mountain View, CA 94043

## TABLE OF CONTENTS

**1. Commands and Application Programs**



**3. Subroutines**

## PERMUTED INDEX

**NAME**

intro – introduction to Shade analyzers

**SYNOPSIS**

analyzer [ shade-options ] [ analyzer-options ]

**DESCRIPTION**

This section describes some existing Shade analyzers. *shade-options* (which must come first) are interpreted and supported by Shade generally. *analyzer-options* are interpreted and supported by analyzers individually; though all of the analyzers described in this section support these *analyzer-options*, other analyzers may not.

The *shade-options* are:

**–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. The **–noflushbench** option is the default on the SunOS versions of Shade. The **–flushbench** option is the default on the Solaris versions.

**–asignal**
**–ssignal**

These options reduce the delay (as measured in number of application instructions executed) between the occurrence of a signal and delivery of that signal to the application. By default, the delay may be arbitrarily long. The **–asignal** switch reduces the delay to roughly a basic block of application code. The **–ssignal** switch forces delivery at the application instruction where the signal occurred. Use of these options degrade Shade's performance.

**–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**.

**−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. Note, this switch only applies to applications that use the SPARC v9 instruction set. The default is **−win32**.

**−assume_ioctl_simple**

(*Solaris only*) This option may provide a work-around if Shade prints the error message "unimplemented 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.

The *analyzer-options* are:

**−U**

Print a usage message and immediately exit.

**−V**

Print a version message and immediately exit.

**−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 the description of *shade_argtrange* in *trange*(3s) for more details.

**−−** *command*

If this option is given, all subsequent arguments, *command*, are interpreted as the name of and arguments for the application program (benchmark) which is to be analyzed.

If the environment variable **SHADE_BENCH_PATH** is set, it is used as a search path for finding the application program; otherwise the environment variable **PATH** is used. If no −− option is given, the analyzers described here read commands from standard input in *shell*(3s) format.

**CAVEATS**

Older versions of the dynamic linker on SunOS do not execute FLUSH instructions, so the **−flush-bench** switch may not be appropriate on those systems.

**FILES**

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

**SEE ALSO**

memory(3s).
The ‘‘Introduction to Shade’’ document.

**NAME**

cachesim5 – cache simulator

**SYNOPSIS**

**cachesim5** cachespec...

**DESCRIPTION**

**cachesim5** is a Shade analyzer for cache simulation.

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**][**I***inc*]

*sz*, *bsz*, and *subbsz* 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).

*ass* is the set associativity (1 by default, i.e. direct mapped). *rep* is the set replace algorithm: **random** (the default) or **lru** (least recently used).

**wb** specifies write-back (the default with write-allocate), **wt** write-through (the default with no-write-allocate), and **wa** write-allocate (implied by 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.

**EXAMPLE**

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.

**SEE ALSO**

intro(1s).

**BUGS**

The cache effects of flush instructions are not simulated.

**NAME**

icount – count executed instructions

**SYNOPSIS**

**icount**

**DESCRIPTION**

**icount** counts and prints the number of instructions executed by the given application program(s).

**SEE ALSO**

intro(1s), ifreq(1s), spixcounts(1s).

**NAME**

ifreq – opcode execution frequency

**SYNOPSIS**

**ifreq**

**DESCRIPTION**

**ifreq** counts and prints the number of instructions executed and/or annulled on a per-opcode basis by the given application program(s).

**SEE ALSO**

intro(1s), icount(1s), spixcounts(1s).

**NAME**

pairs – instruction pairs analyzer

**SYNOPSIS**

**pairs**

**addpairs**

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

**DESCRIPTION**

**pairs** is a Shade analyzer which 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.

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

**postpairs** 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) (effectively reducing the area specified by **–s**) may be specified with the **–m** option.

**FILES**

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

**SEE ALSO**

intro(1s), trips(1s).

**NAME**

spixcounts – generate spix counts file

**SYNOPSIS**

**spixcounts** [ **–b** *bbfmt* ] [ **–l** *logfmt* ] [ **–s** *signal* ]

**DESCRIPTION**

The **spixcounts** Shade analyzer generates a *spixcounts*(5s) format file for each command run. The *spixcounts* file can be used with the SpixTools commands to produce detailed execution information about an application.

The *bbfmt* specifier is a file name template which **spixcounts** uses to determine the name of the application's *spixcounts* file. The *logfmt* is a file name template which **spixcounts** uses to determine the name of a log file to which diagnostic messages may be printed. Both templates may contain format specifiers which are replaced as follows:

%p Replaced with the basename of the application program. This specifier is only valid for the *bbfmt* template.

%n Replaced with a per-command sequence number.

%i Replaced with the process ID of the analyzer.

%% Replaced with '%'.

If no *bbfmt* is specified, **spixcounts** uses the specifier "%p.%n.bb". If no *logfmt* is specified, **spixcounts** prints its diagnostic messages to stdout.

The **–s** switch specifies a signal number or name. When **spixcounts** receives this signal it creates the *bbfmt* file representing the application's execution up to that point. This is useful for application that never terminate.

**CAVEATS**

**spixcounts** may be used on dynamically linked programs, however only the statically linked portion of the program is counted.

**SEE ALSO**

icount(1s), ifreq(1s),
spix(1s), spixstats(1s), sdas(1s), sprint(1s), sadd(1s), spixcounts(5s).

**NAME**

trips – instruction triplets analyzer

**SYNOPSIS**

**trips** [ **–v** ] [ **–g** groupfile ]

**DESCRIPTION**

**trips** is like **pairs**(1s), 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 **–v** option causes information for all executed instruction triplets to be printed.

Like **pairs**(1s), **trips** displays statistics by opcode group rather than by opcode. The **–g** option allows the user to specify the opcode groups. Here for example is the default opcode groups file:

```
# trips – default opcode groups

alu!cc  add[x] sub[x] {s,u}{div,mul} \
        and[n] or[n] x[n]or \
        sll sra srl

alucc   add[x]cc sub[x]cc {s,u}{div,mul}cc mulscc t{add,sub}cc[tv] \
        and[n]cc or[n]cc x[n]orcc

ld      ld{{s,u}{b,h},,d}[a]
ldc     ld[d]c
ldf     ld[d]f

st      st{b,h,,d}[a]
stc     st[d]c
stf     st[d]f

bicc    bicc
cbccc   cbccc
fbfcc   fbfcc
call    call
jmpl    jmpl
nop     nop
sethi   sethi
ticc    ticc
window  save restore

fpop1   f{abs,neg,mov}s f{add,sub,mul,div,sqrt}{s,d,q} fdmulq fsmuld \
        fito{s,d,q} fsto{i,d,q} fdto{i,s,q} fqto{i,s,d}

fcmp    fcmp[e]{s,d,q}

misc    # everything else
```

**SEE ALSO**

intro(1s), pairs(1s).

**NAME**

window – register window analyzer

**SYNOPSIS**

**window**

**DESCRIPTION**

**window** is a Shade analyzer for register windows.

The output produced includes overflow/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*’’. *n* represents the number of windows simulated; ‘‘1+’’ signifies the extra window reserved for the trap handlers.

**SEE ALSO**

intro(1s).

**BUGS**

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

**NAME**

intro – introduction to Shade library

**SYNOPSIS**

**extern char *shade_version;**

**DESCRIPTION**

This section describes the Shade run-time user interface upon which Shade analyzers are built.

Shade is built on top of SpixTools. Shade analyzers typically require SpixTools C header files and the SpixTools library.

Shade library version information is available from the string **shade_version**.

**FILES**

| | |
|---|---|
| $SHADE | Shade installation base directory |
| $SPIX | SpixTools installation base directory |
| $SHADE/src/include | Shade C header files |
| $SPIX/src/include | SpixTools C header files |
| $SHADE/lib/libshade.a | Shade library functions |
| $SHADE/lib/libshade_p.a | Shade profiling library |
| $SPIX/lib/libspix.a | SpixTools library |
| $SPIX/lib/libspix_p.a | SpixTools profiling library |

**SEE ALSO**

cc(1), ld(1), prof(1),
intro(1s).
"Introduction to Shade".
"Introduction to SpixTools".
"SpixTools User's Manual".

**NAME**

shade_appstatus – get application execution state

**SYNOPSIS**

**shade_status_t shade_appstatus(**
**int * pexitval)**

**DESCRIPTION**

This function returns the execution state of the application most recently loaded with **shade_load** or **shade_loadp**. **shade_appstatus** returns one of the following values depending on the application's status:

**SHADE_EXITED**

Indicates the application has exited normally. If *pexitval* is not not, ∗*pexitval* is set to the application's exit code.

**SHADE_ERRORED**

Indicates that the application has exited abnormally. This can occur if the application receives an unhandled signal or executes an illegal instruction.

**SHADE_RUNNING**

Indicates that the application is still running.

**SHADE_NOAPP**

Returned when there is no application loaded.

**SEE ALSO**

load(3s), run(3s).

**NAME**

shade_bench_open, shade_bench_close, shade_bench_creat, shade_bench_dup, shade_bench_dup2, shade_bench_fcntl, shade_bench_ioctl − application I/O

**SYNOPSIS**

```
int
shade_bench_open(path, mode, flags)
char *path;
int mode, flags;

int
shade_bench_close(vfd)
int vfd;

int
shade_bench_creat(path, mode)
char *path;
int mode;

int
shade_bench_dup(vfd)
int vfd;

int
shade_bench_dup2(vfd, vfd2)
int vfd, fd2;

int
shade_bench_fcntl(vfd, op, arg)
int vfd, op, arg;

int
shade_bench_ioctl(vfd, op, arg)
int vfd, op;
char *arg;
```

**DESCRIPTION**

These functions allow the analyzer to perform I/O on behalf of the application. They are provided to allow analyzers to redirect the input and output of an application.

Since the analyzer and the application occupy the same process, Shade must ensure that they do not inadvertently use each others file descriptors. To this end, Shade treats all file descriptors in application I/O requests as virtual numbers. Shade translates these virtual file descriptors to real descriptors on the host system prior to performing the I/O. The functions described here behave like their normal Unix counterparts, except they accept and return virtual file descriptors, not real descriptors.

See **mapfd**(3s) for more details on how Shade maps virtual descriptors to real descriptors.

**SEE ALSO**

mapfd(3s).

**NAME**

shade_load, shade_loadp – load application program

**SYNOPSIS**

```
int
shade_load(path, argv, envp)
char *path;
char **argv;
char **envp;

int
shade_loadp(name, argv, envp)
char *name;
char **argv;
char **envp;

extern char *shade_bench_path;
```

**DESCRIPTION**

**shade_load** and **shade_loadp** prepare the Shade run-time environment to run a new application program with command line arguments *argv* and environment *envp*. Any previously loaded application is lost.

**shade_load** loads the application from the file *path*.

**shade_loadp** searches for an analyzer *name*, and if found, invokes **shade_load**. If the environment variable **SHADE_BENCH_PATH** is set, its value is used for the search path; otherwise the value of the environment variable **PATH** is used.

The file name of the loaded application (for both **shade_load** and **shade_loadp**) is saved in **shade_bench_path**.

Any file descriptors open when the analyzer first starts executing are duplicated for the application. Typically, this duplicates stdin, stdout, and stderr. The analyzer may later redirect the application's I/O with the functions described in **io**(3s), or **mapfd**(3s).

Signal handling is initialized from the initial signal handling state of the analyzer. The analyzer may later alter this with functions described in **signal**(3s).

**RETURN VALUES**

If successful, **shade_load** and **shade_loadp** return 0. Otherwise a diagnostic is printed and −1 is returned.

**SEE ALSO**

io(3s), mapfd(3s), run(3s), shell(3s), signal(3s).

**NAME**

shade_main – Shade analyzer entry point

**SYNOPSIS**

```
int
shade_main(argc, argv, envp)
int argc;
char **argv;
char **envp;

extern char *environ;
```

**DESCRIPTION**

**shade_main** is the user entry point for a Shade analyzer. **shade_main** is supplied by the user and is called by the **main** Shade library function.

*argc* is the number of analyzer command line arguments; *argv* is the analyzer command line argument list. **main** first removes arguments recognized by the Shade library. (See **intro**(1s).)

The environment variable list is available as either *envp* or *environ*. **main** just forwards the environment it started with to the analyzer.

**RETURN VALUES**

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

**CAVEATS**

Shade does not automatically split the argument list in two at ‘‘--’’ as *shadow* did. See **splitargs**(3s) on how to do this.

**SEE ALSO**

intro(1s), intro(3s), splitargs(3s).

**NAME**

shade_mapfd, shade_mappedfd, shade_unmapfd, shade_unmappedfd − file descriptor renumbering

**SYNOPSIS**

**int**
**shade_mapfd(rfd, vfd)**
**int rfd, vfd;**

**int**
**shade_mappedfd(vfd)**
**int vfd;**

**int**
**shade_unmapfd(vfd)**
**int vfd;**

**int**
**shade_unmappedfd(vfd0)**
**int vfd0;**

**DESCRIPTION**

Shade renumbers application file descriptor values when they are supplied to or returned from system calls which are executed on behalf of the application. This allows both the application and analyzer to use, say, file descriptors 0, 1, and 2 without interference. These functions give the analyzer access to the mapping mechanism so it may arrange I/O redirection for the application (but see **io**(3s))

**shade_mapfd** arranges for the real file descriptor *rfd* to be used in place of the virtual (application) file descriptor *vfd*. Any previous mapping for *vfd* is lost.

If *vfd* is −1, the lowest numbered unmapped virtual file descriptor is mapped. If none are available, **shade_mapfd** returns −1. Otherwise, the value of the mapped virtual file descriptor is returned.

**shade_mappedfd** returns the real file descriptor to which the virtual file descriptor *vfd* is mapped (−1 if unmapped).

**shade_unmapfd** removes the mapping for virtual file descriptor *vfd* and returns the real file descriptor to which *vfd* was mapped (−1 if unmapped).

**shade_unmappedfd** returns the lowest numbered unmapped virtual file descriptor greater than or equal to *vfd0* (−1 if there are none).

**CAVEATS**

These functions perform no system calls.

**SEE ALSO**

io(3s), load(3s).

**NAME**

shade_bench_memory – application memory base address

**SYNOPSIS**

**char ***
**shade_bench_memory()**

**DESCRIPTION**

**shade_bench_memory** returns the application memory base address. This value should be added to an application memory address to obtain the corresponding memory address for use by the analyzer.

**NAME**

shade_bench_getrlimit, shade_bench_setrlimit – application resource limits

**SYNOPSIS**

```
#include <sys/time.h>
#include <sys/resource.h>

int
shade_bench_getrlimit(resource, rlim)
int resource;
struct rlimit *rlim;

int
shade_bench_setrlimit(resource, rlim)
int resource;
struct rlimit *rlim;
```

**DESCRIPTION**

To prevent the application from modifying the analyzer's resource limits, Shade intercepts application **getrlimit** and **setrlimit** system calls and redirects them to **shade_bench_getrlimit** and **shade_bench_setrlimit .**

Application resource limits are initialized from the analyzer's initial resource limits. Application requests to increase the limits are honored, but requests to decrease them are silently ignored. This prevents an application from reducing the limits to a value that is too small for the analyzer. All application soft limit changes are recorded, though and are returned back to the application when requested.

Prior to an application **exec** system call, the application's resource limits are instated as the real resource limits for the benefit of the *exec*'d program. Shade does not permit the application to reduce a hard resource limit. If the **exec** fails, the analyzer's resource limits are reinstated.

**CAVEATS**

The RLIM_CORE hard and soft application resource limits always read as 0, reflecting Shade's inability to generate application core dumps.

**SEE ALSO**

exec(2), getrlimit(2), setrlimit(2).

**NAME**

shade_run, shade_step – run and trace application program

**SYNOPSIS**

```
#include <trace.h>

int
shade_run(trbuf, ntrbuf)
Trace *trbuf;
int ntrbuf;

#include <trace.h>
#include <stdtr.h>

Trace *
shade_step()
```

**DESCRIPTION**

Both of these functions run the currently loaded application under Shade and collect trace information. **shade_run** runs the application until *ntrbuf* trace records have been collected in *trbuf*. The buffer *trbuf* must be doubleword aligned.

**shade_step** is a macro that uses **shade_run** to fill an internal buffer of trace records. Each call to **shade_step** returns a new record from the buffer, calling **shade_run** as necessary to refill the buffer.

The **Trace** type is defined in **trace.h** and always contains the following members:

**u_long tr_pc**
: The instruction's virtual address.

**Instr tr_i**
: The instruction word. The type **Instr** is defined in the SpixTools header **instr.h** and is a union of bitfields representing the various components of SPARC instructions.

**char tr_annulled**
: This is 1 if the instruction was annulled, and 0 otherwise.

**char tr_taken**
: This is 1 if the instruction is a branch or trap and the branch or trap was taken. It is also 1 if the instruction is a conditional move and the move happens. Otherwise, it is 0.

**short tr_ih**
: A small integer representing the instruction's opcode. These values are defined in the Spix-Tools header file **IHASH.h**. The **ihash()** function returns one of these opcode values when given an instruction word.

**u_long tr_ea**
: For load and store instructions, this is the virtual effective address of the loaded or stored data. For branch, call, and indirect jump instructions, this is virtual address of the target. For trap instructions, this is the software trap number. Note, on SPARC v9 only the bottom 32 bits of the effective address are stored in this field.

If you define the macro **TR_REGS** prior to including the **trace.h** header, the **Trace** type also contains these members on the SPARC v8 version of Shade:

**int tr_rs1, tr_rs2**
: The contents of the integer registers named in the instruction's *rs1* and (for register+register addressing mode) *rs2* fields prior to executing the instruction.

**int tr_rd**
: The contents of the integer register named in the instruction's *rd* field after executing the

instruction.

**int tr_rd2**
For load and store doubleword instructions, this is the contents of the odd numbered integer register in the register pair after executing the instruction.

If you define the macro **TR_REGS** prior to including the **trace.h** header, the **Trace** type contains these members on the SPARC v9 version of Shade:

```
union ix {
    int         ii[2];
    long long   x;
};
```

**union ix tr_rs1, tr_rs2**
The contents of the integer registers named in the instruction's *rs1* and (for register+register addressing mode) *rs2* fields prior to executing the instruction. The first element of the *ii* array contains the upper 32 bits of the register's value. The second element contains the lower 32 bits.

**union ix tr_rd**
The contents of the integer register named in the instruction's *rd* field after executing the instruction. For load and store doubleword instructions, the first element of the *ii* array contains the value of the even register in the register pair and the second element contains the value of the odd register.

If you define the macro **TR_FREGS** prior to including the **trace.h** header, the **Trace** type contains these members on the SPARC v8 version of Shade:

```
union isdq {
    int         i, ii[2], iiii[4];
    float       s, ss[2], ssss[4];
    double      d, dd[2];
    long double q;
};
```

**union isdq tr_frs1, tr_frs2, tr_frd**
The contents of the floating point registers named in the instruction's *rs1* and *rs2* fields prior to executing the instruction, and in the *rd* field after executing the instruction. Single precision values should be accessed with the *i* or *s* fields. Double precision values should be accessed with the *ii*, *ss*, or *d* fields. Quad precision values should be accessed with the *iiii*, *ssss*, *dd*, or *q* fields. For double precision values, the *ii*[0] and *ss*[0] fields contain the even numbered register's value. The *ii*[1] and *ss*[1] fields contains the odd numbered register's field. For quad precision values, the *iiii*[0] and *ssss*[0] fields contain the lowest numbered register's value and the *iiii*[3] and *ssss*[3] fields contains the highest numbered register's value.

If you define the macro **TR_FREGS** prior to including the **trace.h** header, the **Trace** type contains these members on the SPARC v9 version of Shade:

```
union ixsdq {
    int         i, ii[2], iiii[4];
    long long   x, xx[2];
    float       s, ss[2], ssss[4];
    double      d, dd[2];
    long double q;
};
```

**union ixsdq tr_frs1, tr_frs2, tr_frd**
The contents of the floating point registers named in the instruction's *rs1* and *rs2* fields prior

to executing the instruction, and in the *rd* field after executing the instruction. The interpretation of these fields are the same as the SPARC v8 description above except that double precision value may also be accessed as 64-bit integers with the *x* field. Quad precision values may also be accessed as a pair of 64-bit integers with the *xx* field.

You may also add new fields to the **Trace** type by defining the **TR_MORE** macro prior to including **trace.h**. Fields declared with **TR_MORE** can be filled in with a user-defined trace function set up with **shade_trfun_ih()** or **shade_trfun_it()**. Note, the size of the **Trace** type must be a multiple of eight bytes.

**RETURN VALUES**

**shade_run** returns the number of trace records written to the *trbuf* array. When the application program terminates, **shade_run** returns 0.

**shade_step** returns 0 when the application program terminates.

**CAVEATS**

Note that *ntrbuf* is a limit on the number of traced instructions to run, not the total number of instructions. If not all instructions are traced, more than *ntrbuf* instructions may execute before **shade_run** returns.

**SEE ALSO**

load(3s), shell(3s), trange(3s), trctl(3s).

**NAME**

shade_shell, shade_fshell, shade_sshell – run application scripts

**SYNOPSIS**

```
int
shade_shell(anal)
int (*anal)();

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

int
shade_sshell(str, anal)
char *str;
int (*anal)();
```

**DESCRIPTION**

**shade_shell** reads very simple commands from standard input, and for each command calls **shade_loadp**, and then the function pointed to by *anal*. **shade_fshell** is like **shade_shell**, but reads commands from the stream *fp*. **shade_sshell** is like **shade_shell**, but reads commands from the string *str*.

*anal* is called as:

```
(*anal)(argc, argv, envp)
```

*argc*, *argv*, and *envp* are the number of args, arg list, and environment variable list for the current command.

The shell functions currently support:

- 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),
io(3s), load(3s), mapfd(3s), run(3s).

**NAME**

shade_bench_kill, shade_kill_bench – send signal to application

**SYNOPSIS (Solaris)**

```
#include <signal.h>
#include <ucontext.h>

int
shade_bench_kill(pid, sig)
int pid;
int sig;

int
shade_kill_bench(sig, si, uc)
int sig;
siginfo_t *si;
ucontext_t *uc;
```

**SYNOPSIS (SunOS)**

```
#include <signal.h>

int
shade_bench_kill(pid, sig)
int pid;
int sig;

int
shade_kill_bench(sig, code, sc, addr)
int sig;
int code;
struct sigcontext *sc;
char *addr;
```

**DESCRIPTION**

Shade analyzers take precedence over the application programs they run when it comes to signal handling. (Typical uses for analyzer signal handling are checkpointing or printing intermediate results.) If an analyzer calls **sigaction**(2), **signal**(2), or **sigvec**(2) for a particular signal, the analyzer ''owns'' that signal for the remainder of the Shade job (with all the rights and responsibilities), and application programs are prevented (as transparently as possible) from interfering with the analyzer's handling of that signal.

The routines described here allow an analyzer to send signals to the application even if the analyzer owns the signal. The **shade_bench_kill** routine behaves exactly like the normal **kill** function except when the *pid* is the process ID of the current process. In this case, the signal is emulated in the application and not sent to the analyzer.

The **shade_kill_bench** routine allows an analyzer to forward a caught signal, along with its signal handling parameters, to the application. The application's handler (if any) will be invoked the next time the analyzer calls **shade_run**.

**SEE ALSO**

kill(2).

**NAME**

shade_splitargs – separate analyzer and application argument lists

**SYNOPSIS**

**int**
**shade_splitargs(argv1, pargv2, pargc2)**
**char **argv1;**
**char ***pargv2;**
**int *pargc2;**

**DESCRIPTION**

This 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 "−−" **shade_splitargs** searches for an argument string of "−−" *argv1*. If one is found, it is changed to a NULL pointer, thus terminating the analyzers argument list. The remainder of the argument list and the number of remaining arguments are returned in **pargv2* and **pargc2*. **shade_splitargs** then returns the number of analyzer arguments remaining in *argv1*. If there is no argument string "−−" in *argv1*, the argument list is unchanged, zero is stored in **pargc2*, and the original argument count is returned.

**RETURN VALUES**

The number of argument strings remaining in *argv1* is returned.

**SEE ALSO**

main(3s).

**NAME**

shade_addtrange, shade_subtrange, shade_intrange, shade_argtrange – restrict tracing by instruction address

**SYNOPSIS**

```
void
shade_addtrange(from, to)
unsigned long from, to;

void
shade_subtrange(from, to)
unsigned long from, to;

int
shade_intrange(a)
unsigned long a;

char *
shade_argtrange(arg)
char *arg;
```

**DESCRIPTION**

Instruction tracing can be enabled or disabled according to the address of the application instruction. Per-opcode tracing (see **trctl**(3s)) for the given instruction must also be enabled for the instruction to be traced.

**shade_addtrange** is used to turn tracing on for text in the (asymmetric) address range [*from*,*to*); **shade_subtrange** turns tracing off for a given address range. (The *to* value 0 represents the end of memory.) The low order two bits of *from* and *to* are silently cleared to insure that they are aligned instruction addresses.

**shade_intrange** returns 1 if tracing is enabled for the instruction at address *a*; otherwise it returns 0.

Initially tracing is enabled for all instructions. Aside from this initialization, Shade (in particular **shade_load**) does not make any trace range changes (calls to **shade_addtrange** or **shade_subtrange**). Trace range changes do not take effect until the next call to **shade_run**.

**shade_argtrange** interprets a string (e.g. command line argument) and calls **shade_addtrange** (for strings of the form **+t***from*,*to*) or **shade_subtrange** (for strings of the form **–t***from*,*to*). *from* and *to* here are assumed to be hexadecimal constants. If *from* is missing, the beginning of memory is used; if *to* is missing, the end of memory is used. The comma is required.

If called, **shade_addtrange**, **shade_subtrange**, and **shade_argtrange** must be called before the first call to **shade_run** for an application.

**RETURN VALUES**

**shade_argtrange** returns 0 if successful; otherwise it returns a diagnostic message string.

**SEE ALSO**

run(3s), trctl(3s).

**NAME**

shade_trctl – instruction trace control

**SYNOPSIS**

```
#include <trctl.h>
#include <IHASH.h>

int
shade_trctl_trsize(trsz)
int trsz;

unsigned long
shade_trctl_ih(ih, on, onannulled, trace)
int ih, on, onannulled;
unsigned long trace;

void
shade_trfun_ih(ih, prefun, postfun)
int ih;
void (*prefun)(), (*postfun)();

#include <trctl.h>
#include <ITYPES.h>

unsigned long
shade_trctl_it(it, on, onannulled, trace)
unsigned long it, trace;
int on, onannulled;

void
shade_trfun_it(it, prefun, postfun)
unsigned long it;
void (*prefun)(), (*postfun)();
```

**DESCRIPTION**

These functions determine the information that Shade traces from each instruction in an application. The analyzer must set these tracing parameters before calling **shade_run** and may change its parameters at any point during the analysis. However, new tracing prameters do not take effect until the next call to **shade_run**.

The analyzer should call **shade_trctl_trsize** before any other function in this section to specify the size of the trace structure. The *trsz* parameter must either be zero (which prevents Shade from saving any trace information) or a positive multiple of eight bytes. Often, this parameter is specified as **sizeof(Trace)**. (See **run**(3s).)

The routine **shade_trctl_ih** establishes the tracing parameters for instructions with opcode *ih*. (Opcode values are defined in the header **IHASH.h**.) If the parameter *on* is non-zero, Shade enables tracing for these instructions. If *onannulled* is also non-zero, Shade enables tracing even when these instructions are annulled. The *trace* parameter is a bit mask specifying the members of the **Trace** structure that Shade fills in for these instructions. Possible values include:

**TC_I** Set the *tr_i* field of the trace buffer.

**TC_IH** Set the *tr_ih* field.

**TC_ANNULLED**
Set the *tr_annulled* field.

**TC_TAKEN**
Set the *tr_taken* field if this is a branch, trap, or conditional move instruction. This field is set after the instruction executes.

**TC_PC** Set the *tr_pc* field.

**TC_EA** Set the *tr_ea* field if this instruction is a load, store, branch, call, jump, or trap.

**TC_RS1**
Set the *tr_rs1* field (before executing the instruction) if this instruction has an integer *rs1* operand.

**TC_RS2**
Set the *tr_rs2* field (before executing the instruction) if this instruction has an integer *rs2* operand.

**TC_RD**
Set the *tr_rd* field (and potentially the *tr_rd2* field) after executing this instruction, if this instruction has an integer *rd* operand.

**TC_FRS1**
Set the *tr_frs1* field if this instruction has a floating point *rs1* operand.

**TC_FRS2**
Set the *tr_frs2* field if this instruction has a floating point *rs2* operand.

**TC_FRD**
Set the *tr_frd* field after executing this instruction, if it has a floating point *rd* operand.

The **shade_trfun_ih** routine allows the analyzer to specify two functions which will be called before (*prefun*) and after (*postfun*) executing an instruction with a given opcode. To specify only one function or to cancel a previously registered function, specify a NULL parameter. Functions are only called for instructions with tracing enabled and are never called for annulled instructions.

Each function is called with two arguments. The first is a pointer to the **Trace** structure corresponding to the current instruction. For *prefun*, all requested **Trace** *members except tr_taken*, *tr_rd*, and *tr_frd* will be filled in. For *postfun*, all requested Trace members will be filled in.

The second tracing function argument is a pointer to the **Shade** structure (see **shade.h**), which is a save area for emulated register values. The analyzer may access the current values of the application's registers through this structure. In addition, the analyzer may directly access the application's memory inside user tracing functions. (However, see **memory**(3s) for details.)

**shade_trctl_it** calls **shade_trctl_ih** for each opcode in the groups (as defined in **ITYPES.h**) specified by the bit mask *it*. The most useful of these groups is **IT_ANY**, which includes all opcodes.

Similarly, **shade_trfun_it** calls **shade_trfun_ih**.

**shade_trctl_ih** and **shade_trfun_ih** (and consequently **shade_trctl_it** and **shade_trfun_it**) may be called repeatedly for the same opcode (or the same or overlapping opcode groups); the last call sticks.

**RETURN VALUES**

**shade_trctl_size** returns *trsz* if successful, −1 if unsuccessful.

**shade_trctl_ih** returns a bit mask which indicates which bits of *trace* were accepted. Unacceptable requests include those which don't make sense (e.g. **TC_EA** for NOP instructions) or those which would write beyond the end of the **Trace** structure as set by **shade_trctl_size**.

**shade_trctl_it** returns the bit-wise and of the **shade_trctl_ih** values returned for members of the instruction group(s), *it*.

**SEE ALSO**

memory(3s), run(3s), trange(3s).