sun®
microsystems

Introduction to Shade

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

V5.33A last modified 25/Jun/97

Copyright 1998 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.

Contents

Chapter 0 Introduction **1**

Chapter 1 Example **3**

Chapter 2 Getting Started **7**

Chapter 3 Running and Tracing **11**

Chapter 4 Conflicts of Interest **23**

0

Introduction

What is Shade?

Shade links instruction set simulation and trace generation with custom trace analysis. It finds uses in such areas as computer architecture, microarchitecture, or compiler evaluation where detailed, dynamic, instruction-level information is needed.

Shade tends to run fast because:

1. Shade (the tracer), the application (the tracee), and the analyzer (the trace user) all reside within the same process, which reduces the I/O and context switch overhead associated with file or pipe based trace delivery.
2. Shade dynamically translates the application code into host machine code (adding tracing code) which is directly executed to simulate (emulate) and trace the application code. This new code is cached to minimize translation overhead.
3. The analyzer can control how much trace information to collect and when to collect it.

The result is that for reasonably interesting analyzers, application simulation and tracing is nearly free.

Other features include:

1. Multiple, distinct applications may be run sequentially within a single Shade job. This eliminates the need to combine (by hand, `awk`, etc.) results for each command of multiple-command benchmarks.
2. Tracing is extensible. The analyzer can arrange for its own trace collection functions to be called before and/or after an application instruction is run. These functions have complete access to the application’s state including memory and registers, and can collect information that Shade hasn’t been preprogrammed to collect.
3. Many of the conflicts of interest that arise because the application and analyzer reside within the same process (e.g. I/O, signal handling, storage allocation) are dealt with in some manner to reduce interference.

Some History

Shade grew out of work done in the late 1980's by Peter Hsu, late of Sun Microsystems. A system called Shadow then exhibited the distinguishing features of what later became Shade: application/analyzer coresidence, and dynamic code translation. Before Hsu's departure, work on code translation caching had also begun.

Shade is a new and improved Shadow. It takes the best of Hsu's Shadow, and adds an improved user interface, and increased robustness and efficiency.

Versions

Shade currently comes in four varieties. There are versions to analyze SPARC v8 code and to analyze SPARC v9 code. There are also versions to analyze applications compiled on the old SunOS (4.x) systems and on the newer Solaris (5.x) systems. The SunOS versions run only on SunOS hosts and the Solaris versions run only on Solaris hosts. However, the SPARC v8 and v9 versions run on either SPARC v8 or v9 hosts.

The interfaces to all four Shade versions are very similar, and this document applies to all of them. Most of the interface differences exist to support the 64-bit registers on SPARC v9. These differences are described below where appropriate.

Examples

All of the example programs in this document are distributed on-line with the Shade kit. The `eg` directory of the kit contains the sources for the examples and a makefile to build and run them.

Shortcomings

Roughly in order of increasing likelihood of being fixed ever:

1. Shade cannot run the kernel.
2. Shade cannot run multiprocessor applications.

Upcoming

Subsequent chapters contain:

Example. Contains a short Shade analyzer for the reader to try out, complete with source, and compiling and running instructions.

Getting Started. Describes how Shade analyzers get started, and get command line arguments and environment. Describes how Shade analyzers start application programs and give them command line arguments and environment.

Running and Tracing. Describes how a Shade analyzer runs an application program while collecting and utilizing instruction trace information.

Conflicts of Interest. Describes how to share some per-process resources between analyzer and application programs: memory allocation, I/O, and signal handling.

References

‘‘Shade User’s Manual.’’ UNIX manual pages for Shade analyzers and library functions.

‘‘SpixTools User’s Manual.’’ UNIX manual pages for SpixTools, upon which Shade is based.

‘‘Introduction to SpixTools.’’ SpixTools Tutorial.

The SPARC Architecture Manual, Version 8. SPARC International, Inc.

The SPARC Architecture Manual, Version 9. SPARC International, Inc.

Acknowledgements

Thanks to Peter Hsu of course for Shadow. Thanks to David Keppel for implementation ideas. Thanks to Steve Richardson and Malcolm Wing for user interface, documentation, and debugging ideas.

1

Example

This chapter shows how to construct and run a simple Shade analyzer.

Analyzer Source Code

Suppose you wish to know how often one of the operands is zero when executing an integer add instruction. Shade can do this by examining each add instruction as an application executes. Figure 1.1 shows such a Shade analyzer for SPARC v8. Figure 1.2 shows a SPARC v9 version.

`shade_main`

Execution of the analyzer begins at `shade_main`. Shade makes any analyzer command line arguments and environment variables available to the analyzer via arguments to `shade_main`. (Here, for simplicity, no command line arguments are expected, nor checked.)

`shade_trctl_trsize`

`shade_main` begins by specifying what trace information is desired. The `Trace` structure defined in `trace.h` defines the layout of instruction trace information for a single executed instruction. `Trace` may be customized to a degree by the user. Here `TR_REGS` was defined prior to including `trace.h` to provide storage space for integer register values. The size of the resulting `Trace` structure is supplied to Shade in the call to `shade_trctl_trsize`.

`shade_trctl_ih`

By default, no instruction trace information is collected. The user must explicitly specify what information is to be collected for each opcode (or opcode group). Here, `shade_trctl_ih` is used to specify collection of the same information for each of four opcodes (ADD, ADDX, ADDcc, and ADDXcc on SPARC v8; ADD, ADDC, ADDcc, and ADDCcc on SPARC v9). The `IH_` values (defined in `ihash.h`) are small integers, each representing a particular opcode.

The second and third `shade_trctl_ih` arguments specify that tracing should be enabled for the opcode, except if the instruction is annulled. The fourth argument is a bit mask specifying that the instruction text, and rs1 and rs2 register contents (not register numbers) should be saved for each add opcode.

`shade_shell`

`shade_shell` reads commands one at a time from standard input, and for each command, loads the specified program (setting up I/O redirection, signal handling, etc.), and then calls a user specified function (here `analyze`) to run the program and utilize the trace information. `shade_shell` provides `analyze`

```
#include <stdio.h>
#include <IHASH.h>
#define TR_REGS 1
#include <trace.h>
#include <stdtr.h>
#include <trctl.h>

static double  nadd,   /* # adds executed */
               nadd0;  /* # adds with a 0 operand */

static int     analyze();

int
shade_main (argc, argv, envp)
    int     argc;
    char    **argv;
    char    **envp;
{
    shade_trctl_trsize (sizeof (Trace));

    shade_trctl_ih (IH_ADD, 1, 0, TC_I | TC_RS1 | TC_RS2);
    shade_trctl_ih (IH_ADDX, 1, 0, TC_I | TC_RS1 | TC_RS2);
    shade_trctl_ih (IH_ADDCC, 1, 0, TC_I | TC_RS1 | TC_RS2);
    shade_trctl_ih (IH_ADDXCC, 1, 0, TC_I | TC_RS1 | TC_RS2);

    (void) shade_shell (analyze);

    printf ("%.0f adds, %.0f add0s\n", nadd, nadd0);

    return (0);
}

static int
analyze (argc, argv, envp)
    int     argc;
    char    **argv;
    char    **envp;
{
    Trace   *tr;

    for (; tr = shade_step(); nadd++)
        if (tr->tr_rs1 == 0 ||
            tr->tr_i.i_i && tr->tr_i.i_simm13 == 0 ||
           !tr->tr_i.i_i && tr->tr_rs2 == 0)
                nadd0++;
    return (0);
}
```

Figure 1.1. add0.c (for SPARC v8)

```
#include <stdio.h>
#include <IHASH.h>
#define TR_REGS 1
#include <trace.h>
#include <stdtr.h>
#include <trctl.h>

static double  nadd,   /* # adds executed */
               nadd0;  /* # adds with a 0 operand */

static int     analyze();

int
shade_main (argc, argv, envp)
    int     argc;
    char    **argv;
    char    **envp;
{
    shade_trctl_trsize (sizeof (Trace));

    shade_trctl_ih (IH_ADD, 1, 0, TC_I | TC_RS1 | TC_RS2);
    shade_trctl_ih (IH_ADDC, 1, 0, TC_I | TC_RS1 | TC_RS2);
    shade_trctl_ih (IH_ADDCC, 1, 0, TC_I | TC_RS1 | TC_RS2);
    shade_trctl_ih (IH_ADDCCC, 1, 0, TC_I | TC_RS1 | TC_RS2);

    (void) shade_shell (analyze);

    printf ("%.0f adds, %.0f add0s\n", nadd, nadd0);

    return (0);
}

static int
analyze (argc, argv, envp)
    int     argc;
    char    **argv;
    char    **envp;
{
    Trace   *tr;

    for (; tr = shade_step(); nadd++)
        if (tr->tr_rs1.ii[0] == 0 && tr->tr_rs1.ii[1] == 0 ||
            tr->tr_rs2.ii[0] == 0 && tr->tr_rs2.ii[1] == 0 &&
            tr->tr_i.i_i == 0 ||
            tr->tr_i.i_i == 1 &&
            tr->tr_i.i_simm13 == 0)
                nadd0++;
    return (0);
}
```

Figure 1.2. add0.c (for SPARC v9)

with the command line arguments and environment variables for the command being run. (Here they aren't used.)

shade_step

analyze runs the command with shade_step. Each invocation of shade_step delivers information for one traced instruction (here just integer add instructions). Untraced instructions, though run, aren't delivered by shade_step. After the command has been completely executed in this fashion, shade_step returns 0.

Per previous shade_trctl_ih requests, Shade records the instruction text in tr_i, and the rs1 and rs2 register contents (both recorded before the instruction is executed) in tr_rs1 and tr_rs2. Each iteration of the for loop increments the add counter nadd, and increments the add-zero counter nadd0 if either operand is zero. Note that in the SPARC v9 version, the tr_rs1 and tr_rs2 fields are arrays. The first element in the array corresponds to the high 32 bits of the register's value. The second element corresponds to the low 32 bits. Since registers are only 32 bits wide on SPARC v8, these fields are not arrays in the v8 version.

Once the commands have been run and shade_shell returns, shade_main prints the final counter values and returns. The value returned by shade_shell becomes the exit status for the Shade process. Equivalently, the analyzer may call exit to terminate the Shade process.

Compiling Analyzer

Since Shade is built atop SpixTools, include directories and libraries for both are typically required to compile a Shade analyzer:

```
$ cc -O -I$SHADE/src/include -I$SPIX/src/include add0.c \
    -o add0 $SHADE/lib/libshade.a $SPIX/lib/libspix.a
```

$SHADE and $SPIX here represent the directories where the Shade and SpixTools software has been installed. It is not required that these variables be present in the environment to compile or run Shade analyzers.

Running Analyzer

Now to run this analyzer on, for example, the /bin/date command:[1]

```
$ add0
/bin/date
Wed Jun 25 15:08:46 EDT 1997
<CTRL-D>
75426 adds, 5616 add0s
```

The shade_shell function in the analyzer reads the /bin/date command

[1] User input is shown in **bold**. Press CTRL-D to terminate the analyzer.

from standard input, loads the `/bin/date` command into memory, and then lets the `analyze` function run and analyze the command.

Getting Started

This chapter describes how Shade analyzers and the application programs they run get started.

Starting a Shade Analyzer

To illustrate this, the source for a simple analyzer `analecho` is shown in Figure 2.1.

```
#include <stdio.h>

int
shade_main (argc, argv, envp)
    int    argc;
    char   **argv;
    char   **envp;
{
    int    i;

    printf ("argc=%d\n", argc);

    for (i = 0; i < argc; i++)
        printf ("argv[%d]=%s\n", i, argv[i]);

    for (i = 0; envp[i]; i++)
        printf ("envp[%d]=%s\n", i, envp[i]);

    return (0);
}
```

Figure 2.1. analecho.c

Here is a sample run of `analecho`.

```
$ analecho hello world
argc=3
argv[0]=analecho
argv[1]=hello
argv[2]=world
envp[0]=HOME=/home/sobchak7/rfc
[...]
```

The `main` function is supplied by the Shade run-time library. `main` interprets and deletes Shade-specific command line options, calls some Shade initialization functions, and then calls `shade_main`.

Shade provides `shade_main` with the number of command line arguments `argc`, command line arguments `argv`, and environment variable list `envp` (inherited unmodified from `shade`). The variable `environ`, which is used by, e.g., the C library functions `getenv` and `putenv`, is initialized to the same value as `envp`.

Starting an Application

Shade permits an analyzer to run and trace one application at a time. The function `shade_load` starts a new application program.

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

`path` is the name of the file containing the application program. `argv` and `envp` are the command line arguments and environment variable list to be supplied to the application. Note that the environment variables that the application sees need not be the same as those provided to the analyzer.

If `shade_load` is successful, it returns 0. Otherwise it prints a diagnostic and returns –1.

A variant of `shade_load` which does a path search for the application is `shade_loadp`.

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

If `name` is unqualified, `shade_loadp` uses the analyzer environment variable `SHADE_BENCH_PATH` (or if this is not present, `PATH`) to search for the application program. If it is found, `shade_load` is supplied with file name of the application, `argv`, and `envp`. `shade_loadp` returns 0 if successful, or prints a diagnostic and returns –1.

Once an application has been loaded, it may be run and traced with `shade_run` as described in a subsequent chapter.

The functions `shade_shell` and `shade_fshell` read (very simple) commands from a standard I/O stream, invoke `shade_loadp`, set up I/O redirection for the application, and call a user function to run and trace each application.

Under Shadow, the convention was to specify both analyzer and application command line arguments on the `shadow` command line.

```
$ shadow analyzer args -- application args
```

The function `shade_splitargs` may be used to support this convention under Shade.

```
int
shade_splitargs (argv, pbargv, pbargc)
    char **argv, ***pbargv;
    int  *pbargc;
```

Given an argument list `argv`, `shade_splitargs` searches for the ‘‘`--`’’ argument. If found, it is changed to 0 (thus null terminating the analyzer’s argument list at that point). The remainder of the argument list and number of remaining arguments are returned (by reference) in `*pbargv` and `*pbargc`. `shade_splitargs` then returns the new number of analyzer arguments. If ‘‘`--`’’ isn’t found, the argument list is unchanged, 0 is returned in `*pbargc` and the original argument count is returned by `shade_splitargs`.

Running and Tracing

As an application is run, instruction trace records for executed or annulled instructions may be saved for later use by the analyzer. Shade is preprogrammed to record most of the information about an instruction that an analyzer might need. An escape mechanism is provided to record additional information. Instructions may be selectively traced by opcode or address.

Trace Records

Shade trace records are composed of two variable length parts. The first part is used by Shade to record trace information that Shade knows how to collect such as instruction addresses, load/store data addresses, register values, etc. The second part may be used by the analyzer to collect any other trace information. Either or both of these parts may be empty (zero length).

Shade currently imposes two perhaps strange restrictions on the trace record format. First, both parts of the trace record must be doubleword (8 bytes) aligned. Second, the offsets within the trace record for information recorded by Shade are fixed. These restrictions simplify trace code generation and improve the efficiency of the resultant tracing code, but may introduce unused ‘‘holes’’ in trace records. Offsetting this however is the placement of ‘‘more useful’’ trace information nearer the beginning of the trace record.

Figure 3.1 shows the SPARC v8 version of the `trace.h` header file, which defines the Shade trace record. Figure 3.2 shows the SPARC v9 version of this header. The default trace record should be sufficient for most purposes, though limited customization is provided by a few preprocessor symbols. Space for integer and/or floating point registers may be reserved by defining `TR_REGS` or `TR_FREGS` prior to including `trace.h`. Space for analyzer specific trace information may be reserved by defining `TR_MORE`.

The `Trace` structure members are:

`tr_pc`
    Instruction address.

`tr_i`
    Instruction text (word). The type `Instr` (defined in the SpixTools header

```
#ifndef _trace_h_
#define _trace_h_

#include <instr.h>

typedef struct {
    u_long  tr_pc;        /* instruction address */
    Instr   tr_i;         /* instruction text */
    char    tr_annulled;  /* instruction annulled? */
    char    tr_taken;     /* branch or trap taken? */
    short   tr_ih;        /* ihash() value (opcode) */
    u_long  tr_ea;        /* target address for dcti’s.
                           * (NOT fall thru address for untaken branches).
                           * rs1+rs2|simm13 for loads, stores, traps.
                           */

#if defined(TR_REGS) || defined(TR_FREGS)
    int     tr_rs1;  /* rs1 contents before execution */
    int     tr_rs2;  /* rs2 contents before execution */
    int     tr_rd;   /* rd contents after execution */
    int     tr_rd2;  /* rd contents 2nd word (ldd, std) */
#endif

#if defined(TR_FREGS)
    union isdq {
        int             i, ii[2], iiii[4];
        float           s, ss[2], ssss[4];
        double          d, dd[2];
#ifdef REAL128
        long double     q;
#endif
    }
        tr_frs1,  /* frs1 contents before execution */
        tr_frs2,  /* frs2 contents before execution */
        tr_frd;   /* frd contents after execution */
#endif

#if defined(TR_MORE)
    TR_MORE
#endif
} Trace;

#endif  /* _trace_h_ */
```

Figure 3.1. trace.h (for SPARC v8)

file `instr.h`) is a union of bit fields representing the various components of a SPARC instruction.

`tr_annulled`

This is 1 if the traced instruction was annulled (squashed), or 0 otherwise.

```
#ifndef _trace_h_
#define _trace_h_

#include <instr.h>

typedef struct {
    u_long  tr_pc;        /* instruction address */
    Instr   tr_i;         /* instruction text */
    char    tr_annulled;  /* instruction annulled? */
    char    tr_taken;     /* branch or trap taken? */
    short   tr_ih;        /* ihash() value (opcode) */
    u_long  tr_ea;        /* target address for dcti’s.
                           * (NOT fall thru address for untaken branches).
                           * rs1+rs2|simm13 for loads, stores, traps.
                           */

#if defined(TR_REGS) || defined(TR_FREGS)
    union ix {
        int             ii[2];
#ifdef INT64
        long long       x;
#endif
    }
        tr_rs1;  /* rs1 contents before execution */
        tr_rs2;  /* rs2 contents before execution */
        tr_rd;   /* rd contents after execution */
#endif

#if defined(TR_FREGS)
    int         tr_pad;

    union ixsdq {
        int             i, ii[2], iiii[4];
#ifdef INT64
        long long       x, xx[2];
#endif
        float           s, ss[2], ssss[4];
        double          d, dd[2];
#ifdef REAL128
        long double     q;
#endif
    }
        tr_frs1,  /* frs1 contents before execution */
        tr_frs2,  /* frs2 contents before execution */
        tr_frd;   /* frd contents after execution */
#endif

#if defined(TR_MORE)
    TR_MORE
#endif
} Trace;

#endif  /* _trace_h_ */
```

Figure 3.2. trace.h (for SPARC v9)

The analyzer can control whether or not annulled instructions are traced.

`tr_taken`

For branch or trap instructions, this is 1 if the branch or trap was taken, or 0 otherwise. For conditional moves (on SPARC v9), this is 1 if the move happened, or 0 otherwise.

`tr_ih`

A small integer representing the opcode. These values are defined in the SpixTools header file `IHASH.h`, and are returned (given the instruction word) by the SpixTools function `ihash`.

`tr_ea`

Effective address. For load and store instructions, this is the address of the loaded or stored data. For branch, call, or indirect jump instructions, this is the target (destination) address. For trap instructions, this is the software trap number. Note, on SPARC v9 only the bottom 32 bits of the address are stored in this field.

`tr_rs1`, **`tr_rs2`**

Contents of the integer registers named in the instruction's rs1 and (for register+register addressing mode) rs2 fields before executing instruction. Note, on SPARC v9 these fields are arrays. The first element of the array is the upper 32 bits of the register's value. The second element is the lower 32 bits.

`tr_rd`, **`tr_rd2`**

Contents of the integer register(s) named in the instruction's rd field after executing instruction. On SPARC v8, the `tr_rd2` field is used to hold the value of the odd numbered register for load and store doubleword instructions. On SPARC v9, the first element of the `tr_rd` field holds the value of the even numbered register and the second element holds the value of the odd numbered register for load and store doubleword instructions.

`tr_frs1`, **`tr_frs2`**, **`tr_frd`**

Contents of the floating point registers named in the instruction's rs1 and rs2 fields prior to executing instruction, or rd field after executing instruction. For single precision operations, the value should be accessed with the `i` (for integer) or `s` (for single precision floating point) `isdq` union member. For double precision operations, the value should be accessed with the `d` (for double precision floating point) or `ii` (for integer register pair) or `ss` (for single precision floating point register pair) `isdq` member. `ii[0]` and `ss[0]` contain the value of the pair's even numbered register. For quad precision operations, the value should be accessed with the `q, iiii, ssss,` or `dd` members. On SPARC v9, double precision values can also be accessed as 64-bit integers with the `x` or `xx` fields.

However the trace record is defined, Shade needs to be informed how big it is. Typically this is as simple as:

```
shade_trctl_trsize (sizeof (Trace));
```

Trace Control

By default, Shade collects none of the trace information just described. For each opcode, the user must turn tracing on or off (including, or not, annulled instructions), and turn filling on or off for each of the trace record fields.

```
unsigned long
shade_trctl_ih (ih, on, onannulled, mask)
    int ih, on, onannulled;
    unsigned long mask;

unsigned long
shade_trctl_it (it, on, onannulled, mask)
    unsigned long it;
    int on, onannulled;
    unsigned long mask;
```

`shade_trctl_ih` is used to control tracing for a single opcode identified by `ih` (values are defined in the SpixTools header file `IHASH.h`). `shade_trctl_it` is used to control tracing for a group of instructions specified as a bit mask `it` (component values are defined in the SpixTools header file `ITYPES.h`).

The remaining arguments have the same meaning for both functions. `on` enables tracing for the indicated opcode(s). If this is not done, no trace records will be generated for these opcodes. (The instruction must furthermore be in a traced instruction range (see below) to be traced.) `onannulled` additionally enables tracing of annulled instructions. The effective address and register value trace record fields are not filled for annulled instructions.

`mask` is a bit mask indicating which trace record fields should be filled. It is composed from values defined in the Shade header file `trctl.h`.

```
#define TC_I         1
#define TC_IH        2
#define TC_ANNULLED  4
#define TC_TAKEN     8
#define TC_PC        16
#define TC_EA        32
#define TC_RS1       64
#define TC_RS2       128
#define TC_RD        256
#define TC_FRS1      512
#define TC_FRS2      1024
#define TC_FRD       2048
```

These functions return `mask` after clearing bits representing trace record fields which are meaningless or unsupported for the given opcode. (`shade_trctl_it` just repeatedly calls `shade_trctl_ih`, and then returns the bitwise conjunction of the `shade_trctl_ih` return values.)

`shade_trctl_ih` and `shade_trctl_it` calls may both be used. The last call which applies to a given opcode sticks (overrides previous calls). The following sequence (from a Shade cache simulator) turns instruction address, annulled flag, and opcode tracing on for all instructions, annulled included, and furthermore turns effective address tracing on just for load and store instructions.

```
shade_trctl_it (IT_ANY, 1, 1, TC_ANNULLED|TC_IH|TC_PC);
shade_trctl_it (IT_LOAD|IT_STORE, 1, 1, TC_ANNULLED|TC_IH|TC_PC|TC_EA);
```

Trace Address Ranges

Instruction tracing may be enabled or disabled according to the instruction's address. Initially, tracing is enabled for instructions anywhere in memory. The user may restrict tracing to specific regions of memory with the following functions.

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

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

`shade_addtrange` enables tracing of instructions with addresses from `from` to (but excluding) `to`. Similarly, `shade_subtrange` disables tracing of instructions in a given address range. Any changes will take effect the next time `shade_run` is called.

For simplicity, the low order two bits of `from` and `to` are silently cleared before use; instruction addresses should be word aligned. A `to` value of 0 represents the end of memory.

After initialization, Shade does not call these functions, even when an application is loaded with `shade_load`. If instruction address tracing restrictions have been established, and a different application is then loaded, the previous trace address ranges will likely be meaningless. It is then the analyzer's responsibility to cope with the situation, say by terminating with a diagnostic.

Given an address `shade_intrange` returns 1 if the that address lies within an address range for which tracing is enabled, or 0 otherwise.

```
int
shade_intrange (a)
    unsigned long a;
```

The function `shade_argtrange` is provided to simplify processing of analyzer command line arguments which specify trace address ranges.

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

`arg` is a string of the form `+t`[*from*]`,`[*to*] or `-t`[*from*]`,`[*to*]. `shade_argtrange` interprets *from* and *to* as hex constants, and calls `shade_addtrange` (for `+t`) or `shade_subtrange` (for `-t`). If *from* is missing the start of memory is used; if *to* is missing the end of memory is used. The comma is always required.

If successful `shade_argtrange` returns 0. Otherwise it returns a diagnostic message string. Here is an example of how `shade_argtrange` might be used.

```
int
shade_main (argc, argv, envp)
    int   argc;
    char  **argv, **envp;
{
    char  *tmsg;
    int   anyt, i;

    for (anyt = 0, i = 1; i < argc; i++)
        if ((argv[i][0] == '-' ||
             argv[i][0] == '+') &&
             argv[i][1] == 't') {
            if (!anyt++ && argv[i][0] == '+')
                (void) shade_argtrange ("-t,");
            if (tmsg = shade_argtrange (argv[i]))
                shade_fatal ("%s: %s", argv[i], tmsg);
        }

    /* etc */
}
```

Note that if the user gives a `+t` option first, tracing is first turned off for all of memory. If the analyzer did not provide this convenience, the user would have to use one or more `-t` options since initially tracing is enabled for all of memory.

User Trace Functions

To collect additional trace information the user may specify functions to be called before or after the traced instruction is executed.

```
unsigned long
shade_trfun_ih (ih, prefun, postfun)
    int ih;
    void (*prefun)(), (*postfun)();

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

User trace functions may be specified for a single opcode `ih` or opcode group `it` as with `shade_trctl_ih` and `shade_trctl_it`. Tracing must be enabled (even if no preprogrammed trace record filling is enabled) to enable calling of user trace functions. User trace functions are not called for annulled instructions.

The function pointed to by `prefun` is called before the traced instruction is executed, and the function pointed to by `postfun` is called after.

User functions are called with two arguments. The first is a pointer to the trace record for the instruction. When the pre-execution user trace function is called, the taken flag and destination register values in the trace record will be unfilled (these fields are filled after instruction execution). Otherwise all requested fields will be filled when the user trace functions are called.

The second user trace function argument is a pointer to a `Shade` structure as defined in the Shade header file `shade.h`. Figure 3.3 shows the SPARC v8 definition of this header and figure 3.4 shows the SPARC v9 definition.

```
#ifndef _shade_h_
#define _shade_h_

typedef struct {
    int         sh_r[32];   /* int register file */
    int         sh_y;       /* y register */
    char        sh_icc;     /* integer cond codes (see below) */

    union {
        int     i[32];
        float   s[32];
        double  d[16];
    } sh_fr;                /* fp register file */

    unsigned    sh_fsr;     /* fp state register */
} Shade;

#define sh_g0     sh_r[0]
#define sh_g1     sh_r[1]
[...]
#define sh_i6     sh_r[30]
#define sh_i7     sh_r[31]

#define sh_fp     sh_i6
#define sh_sp     sh_o6

/*
 * sh_icc component values:
 */
#define SH_ICC_N  64   /* negative */
#define SH_ICC_Z  32   /* zero */
#define SH_ICC_V  16   /* overflow */
#define SH_ICC_C   8   /* carry */

#endif    /* _shade_h_ */
```

Figure 3.3. shade.h (for SPARC v8)

The trace function may extract application state from this structure, as well as read directly from the application's memory space. The trace function may not modify the `Shade` structure. Doing so will cause unpredictable behavior.

```
#ifndef _shade_h_
#define _shade_h_

typedef union {
    int         ii[2];
    unsigned    uu[2];
#ifdef INT64
    long long   x;
#endif
} xreg_t;

typedef struct {
    xreg_t          sh_r[32];   /* int register file */
    int             sh_y;       /* y register */
    char            sh_icc;     /* integer cond codes (see below) */
    char            sh_xcc;     /* extended integer cond codes */
    unsigned char   sh_asi;     /* address space identifier */
    unsigned char   sh_gsr;     /* graphic status register */

    int             sh_fr[128]; /* floating point registers */
    unsigned        sh_fsr;     /* floating point state register, lsw */
    unsigned        sh_fcc[3];  /* fp condition codes 1-3 (in %fsr format) */
} Shade;

#define sh_g0     sh_r[0]
#define sh_g1     sh_r[1]
[...]
#define sh_i6     sh_r[30]
#define sh_i7     sh_r[31]

#define sh_fp     sh_i6
#define sh_sp     sh_o6

/*
 * sh_icc component values:
 */
#define SH_ICC_N  64   /* negative */
#define SH_ICC_Z  32   /* zero */
#define SH_ICC_V  16   /* overflow */
#define SH_ICC_C   8   /* carry */

#endif    /* _shade_h_ */
```

Figure 3.4. shade.h (for SPARC v9)

Running Applications

Once tracing parameters have been established, the analyzer may begin running the application.

```
int
shade_run (tr, ntr)
    Trace *tr;
    int ntr;
```

`shade_run` runs the application and fills in successive entries in the array `tr` (up to a limit of `ntr` entries) for each executed or annulled instruction for which tracing is enabled. Note that `ntr` limits the amount of tracing done, not the number of instructions run.

`shade_run` returns the number of `tr` entries that it filled. This may be less than `ntr` if the application terminates or if there is insufficient room near the end of `tr` for the next ‘‘block’’ of instructions to be run. After the application has terminated and previous calls have returned the final trace information, `shade_run` returns 0.

`shade_step` is a variant of `shade_run` which goes a single traced instruction at a time.

```
Trace *
shade_step()
```

`shade_step` runs the application through the next traced instruction, and returns the trace information for that instruction. It returns 0 when the application terminates.

Actually, `shade_step` is just a macro defined in the Shade header file `stdtr.h`. It uses `shade_run` as necessary to fill a statically allocated trace buffer and then marches through the buffer, one traced instruction at a time. `shade_step` is to `shade_run` as `getchar` is to `read`.

Example

Figures 3.5a and 3.5b show a simple analyzer called `syscall` which traces application system calls. It should be compiled with `-Dsolaris` if you are using the Solaris version of Shade or with `-Dsunos` if you are using the SunOS version of Shade.

This Shade analyzer relies on a particular implementation of the UNIX system call interface for SPARC. A system call is performed by executing a software trap instruction with trap number ST_SYSCALL. The system call is specified in register g1 (see `/usr/include/sys/syscall.h`). System call arguments are passed in registers o0-o5. Upon return, the carry bit of the integer condition codes indicates whether the call was successful (clear) or not (set). If successful, the return value is in registers o0 and sometimes additionally o1. Otherwise the error number is in register o0 (see `/usr/include/sys/errno.h`).

```
#include <stdio.h>
#include <ITYPES.h>
#include <shade.h>

#define TR_MORE        int tr_syscall, tr_errno;
#include <trace.h>
#include <stdtr.h>
#include <trctl.h>

#ifdef solaris
#   include <sys/trap.h>
#else
#   include <sparc/trap.h>
#endif

static void  pre_ticc();
static void  post_ticc();

int
shade_main (aargc, aargv, envp)
    int    aargc;
    char   **aargv;
    char   **envp;
{
    Trace  *tr;
    char   **bargv;
    int    bargc;

    aargc = shade_splitargs (aargv, &bargv, &bargc);

    if (bargc <= 0 ||
      shade_loadp (*bargv, bargv, envp) < 0)
        return (1);

    shade_trctl_trsize (sizeof (Trace));
    shade_trctl_it (IT_TICC, 1, 0, TC_EA | TC_TAKEN);
    shade_trfun_it (IT_TICC, pre_ticc, post_ticc);

    while (tr = shade_step())
        if (tr->tr_syscall != -1)
            printf ("syscall %3d  errno %3d\n",
              tr->tr_syscall, tr->tr_errno);
    return (0);
}
```

Figure 3.5a. syscall.c (Part 1 of 2)

For simplicity this analyzer, just traces system call numbers and error numbers. The ambitious reader may wish extend it to generate such output as the *trace*(1) or *truss*(1) commands generate.

```
static void
pre_ticc (tr, sh)
    Trace  *tr;
    Shade  *sh;
{
    if (tr->tr_ea != ST_SYSCALL)
        tr->tr_syscall = -1;
    else {
        tr->tr_syscall = sh->sh_g1;
        if (tr->tr_syscall == 0)
            tr->tr_syscall = sh->sh_o0;
    }
}

static void
post_ticc (tr, sh)
    Trace  *tr;
    Shade  *sh;
{
    if (tr->tr_syscall != -1)
        if (!tr->tr_taken)
            tr->tr_syscall = -1;
        else
            if (sh->sh_icc & SH_ICC_C)
                tr->tr_errno = sh->sh_o0;
            else
                tr->tr_errno = 0;
}
```

Figure 3.5b. syscall.c (Part 2 of 2)

With `TR_MORE` we extend the `Trace` structure to add space for a system call number `tr_syscall` and a system call error number `tr_errno`. The size of the resulting `Trace` structure is supplied to Shade with `shade_trctl_trsize`.

The `shade_trctl_it` call causes Shade to only trace non-annulled trap instructions, and only record the software trap number (in `tr_ea`) and a flag indicating whether the trap was taken (in `tr_taken`).

The `shade_trfun_it` call causes Shade to call the function `pre_ticc` before executing a trap instruction, and call the function `post_ticc` afterwards.

The function `pre_ticc` records the system call number in `tr_syscall`. For non-system-call traps, –1 is stored instead. For indirect system calls (g1==0), the real system call number (o0) is recorded.

The function `post_ticc` records (if the trap was taken) the error number for failed system calls, or 0 for successful calls.

Each invocation of `shade_step` here returns information for one nonannulled trap instruction, since that is all that tracing has been enabled for. Note that it could be a long time between executing the application system call and processing the corresponding trace record in `shade_main`.

This example is written for the SPARC v8 version of Shade. It can be ported to SPARC v9 by changing the references to `sh_g1` and `sh_o0` to `sh_g1.ii[1]` and `sh_o0.ii[1]` respectively.

Here is a sample run of the `syscall` analyzer.

```
$ syscall -- /bin/date
Wed Jun 25 15:08:46 EDT 1997
syscall   5  errno   0
syscall 115  errno   0
syscall   5  errno   2
syscall   5  errno   0
syscall  28  errno   0
syscall 115  errno   0
syscall 115  errno   0
[...]
```

4

Conflicts of Interest

This chapter describes how Shade copes with some of the contention that comes from running analyzer and applications within the same UNIX process. The information in this chapter is not generally needed to write Shade analyzers, and may be skipped on a first reading.

Memory

Shade simulates the application's address space. The application text, data, etc. are placed in an out of the way place in memory, and application memory addresses are translated to/from their corresponding actual memory addresses. For example, when the application executes a load instruction, the application memory address used in the load instruction is translated to obtain the actual memory address that Shade uses to perform the load operation.

All application memory addresses are at fixed offset from their corresponding actual memory addresses. This offset, or application base address, is returned by the `shade_bench_memory` function. If the analyzer wishes to examine the application's memory (e.g. from inside a user trace function), it should add this value to the application memory address to obtain a pointer to dereference.

By default, Shade determines a good location for the application's memory addresses. Since the application may dynamically grow its address space, though, it is possible that the application's addresses will collide with the analyzer's. If this occurs, Shade issues an error message and terminates the application. User's can then avoid the problem by specifying the Shade switch **–benchmem=***num* (see *intro*(1s) in the ‘‘Shade User's Manual’’). This switch allows users to override the default location for the application's addresses.

It is sometimes useful to specify **–benchmem=0**. This tells Shade to place the application's addresses at their native locations. This only works, though, if the analyzer is linked at an out of the way spot. All the analyzers described in section 1 of the ‘‘Shade User's Manual’’ are linked like this to support **–benchmem=0**.

The method for linking an analyzer like this differs depending on the version of your operating system. On Solaris systems, simply use the linker mapfile provided with the Shade kit. For example:

```
$ cc -o add0 -dn -Wl,-M,$SHADE/lib/mapfile add0.o \
    $SHADE/lib/libshade.a $SPIX/lib/libspix.a
```

Here, the `–Wl,–M,$SHADE/lib/mapfile` switch specifies the linker mapfile that places the analyzer at an out of the way location. The `–dn` switch links the analyzer statically, without shared libraries. It is better to avoid linking the analyzer with shared libraries because shared libraries occupy more address space and increase the likelyhood of memory conflicts with the application.

The method for linking an analyzer at a nonstandard location is more complex on SunOS systems. On these systems you must link the analyzer as an overlay and then run it with a special driver. A typical linker command looks like this:

```
$ ld -o add0.anal -Bstatic -A $SHADE/lib.anal/dummy -T 10000020 \
    $SHADE/lib.anal/crt0.o add0.o $SHADE/lib/libshade.a \
    $SPIX/lib/libspix.a -lc
```

Here, the `–Bstatic` switch links the analyzer without shared libraries. The `–A $SHADE/lib.anal/dummy` switch specifies that this is an overlay. The `–T 10000020` switch specifies an out of the way hexadecimal address for the analyzer. You can change this address, but be sure to specify a value that is 32 (20 hex) bytes larger than a page boundary. Note, the first object module specified on the command line must be the special Shade start-up code, `$SHADE/lib.anal/crt0.o.` You must also link against the standard C library `–lc.`

Once linked, you must use a special driver program to run the analyzer:

```
$ $SHADE/bin.anal/shade add0.anal -benchmem=0
```

Note, analyzers linked this way on SunOS do not support profiling or shared libraries. (Although, the application running under the analyzer may use shared libraries.)

I/O

In order to reduce I/O conflict, Shade renumbers file descriptors as used by the application. So for example, when the application performs an operation on standard output (file descriptor 1), it is actually using some other file descriptor (say 27) without knowing it. This leaves the analyzer free and clear to use file descriptor 1.

To do this, Shade intercepts all application system calls that use or generate a file descriptor and translates the value. This renumbering may be controlled by the analyzer at two levels. Firstly (likely most usefully), the analyzer may directly call several functions which Shade uses to handle application I/O system calls.

For example, Figure 4.1 shows some code used by `shade_shell` to handle I/O redirection for an application command.

```
static void
shade_shell_io (op, file)
    char  *op, *file;
{
    int   fd;

    if (!strcmp (op, "<")) {
        if (0 > (fd = shade_bench_open (file, 0)))
            shade_fatal ("%s: can't open", file);
        (void) shade_bench_dup2 (fd, 0);
        (void) shade_bench_close (fd);
    }
    else if (!strcmp (op, ">")) {
        if (0 > (fd = shade_bench_creat (file, 0666)))
            shade_fatal ("%s: can't creat", file);
        (void) shade_bench_dup2 (fd, 1);
        (void) shade_bench_close (fd);
    }
    else if (!strcmp (op, ">&")) {
        if (0 > (fd = shade_bench_creat (file, 0666)))
            shade_fatal ("%s: can't creat", file);
        (void) shade_bench_dup2 (fd, 1);
        (void) shade_bench_dup2 (fd, 2);
        (void) shade_bench_close (fd);
    }
    else [...]
    else
        shade_fatal ("%s: bad i/o redirect", op);
}
```

Figure 4.1. shade_shell_io

At a deeper level, the analyzer may use the functions `shade_mapfd`, `shade_mappedfd`, `shade_unmapfd`, and `shade_unmappedfd` to get and set the file descriptor mappings. For example, Figure 4.2 shows how `shade_bench_open` is written.

For more information, see *io*(3s) and *mapfd*(3s) in ‘‘The Shade User’s Manual.’’

Signals

No, signals aren’t renumbered. Instead an ownership protocol is introduced: if the analyzer calls `sigaction`, `signal`, or `sigvec` for a given signal, the analyzer *owns* that signal from then on, and Shade will try to keep the application program from interfering with the analyzer’s use of that signal. So for example if the analyzer wants interrupts ignored, and the application wants interrupts caught, then interrupts will be ignored.

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

    if (0 > (vfd = shade_unmappedfd (0))) {
        errno = EMFILE;
        return (-1);
    }
    if (0 > (pfd = open (path, mode, flags)))
        return (-1);

    return (shade_mapfd (pfd, vfd));
}
```

Figure 4.2. shade_bench_open

For more information, see *signal*(3s) in ‘‘The Shade User’s Manual.’’