

```

new/usr/src/cmd/dtrace/test/tst/common/privilges/tst.procpriv.ksh
*****
4157 Fri Jun 22 23:09:00 2012
new/usr/src/cmd/dtrace/test/tst/common/privilges/tst.procpriv.ksh
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
*****
```

```

1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #

22 #
23 # Copyright (c) 2012, Joyent, Inc. All rights reserved.
24 #

26 ppriv -s A=basic,dtrace_proc,dtrace_user $$

28 #
29 # When we have dtrace_proc (but lack dtrace_kernel), we expect to be able to
30 # read certain curpsinfo/curlwpsinfo/curcpu fields even though they require
31 # reading in-kernel state. However, there are other fields in these translated
32 # structures that we know we shouldn't be able to read, as they require reading
33 # in-kernel state that we cannot read with only dtrace_proc. Finally, there
34 # are a few fields that we may or may not be able to read depending on the
35 # specifics of context. This test therefore asserts that we can read what we
36 # think we should be able to, that we can't read what we think we shouldn't be
37 # able to, and (for purposes of completeness) that we are indifferent about
38 # what we cannot assert one way or the other.
39 #
40 /usr/sbin/dtrace -q -Cs /dev/stdin <<EOF

42 #define CANREAD(what, field) \
43     BEGIN { errmsg = "can't read field from what"; printf("field: "); \
44             trace(what->field); printf("\n"); }

46 #define CANTREAD(what, field) \
47     BEGIN { errmsg = ""; trace(what->field); \
48             printf("\nable to successfully read field from what!"); exit(1); }

50 #define MIGHTREAD(what, field) \
51     BEGIN { errmsg = ""; printf("field: "); trace(what->field); printf("\n"); }

53 #define CANREADVAR(vname) \
54     BEGIN { errmsg = "can't read vname"; printf("vname: "); \
55             trace(vname); printf("\n"); }

57 #define CANTREADVAR(vname) \
58     BEGIN { errmsg = ""; trace(vname); \
59             printf("\nable to successfully read vname!"); exit(1); }

61 #define MIGHTREADVAR(vname) \

```

```

1 new/usr/src/cmd/dtrace/test/tst/common/privilges/tst.procpriv.ksh
2
62     BEGIN { errmsg = ""; printf("vname: "); trace(vname); printf("\n"); }

64 CANREAD(curpsinfo, pr_pid)
65 CANREAD(curpsinfo, pr_nlwp)
66 CANREAD(curpsinfo, pr_ppid)
67 CANREAD(curpsinfo, pr_uid)
68 CANREAD(curpsinfo, pr_euid)
69 CANREAD(curpsinfo, pr_gid)
70 CANREAD(curpsinfo, pr_egid)
71 CANREAD(curpsinfo, pr_addr)
72 CANREAD(curpsinfo, pr_start)
73 CANREAD(curpsinfo, pr_fname)
74 CANREAD(curpsinfo, pr_psargs)
75 CANREAD(curpsinfo, pr_argc)
76 CANREAD(curpsinfo, pr_argv)
77 CANREAD(curpsinfo, pr_envp)
78 CANREAD(curpsinfo, pr_dmodel)

80 /*
81  * If our p_pgdp points to the same pid structure as our p_pidp, we will
82  * be able to read pr_pgdp -- but we won't if not.
83  */
84 MIGHTREAD(curpsinfo, pr_pgdp)

86 CANTREAD(curpsinfo, pr_sid)
87 CANTREAD(curpsinfo, pr_ttydev)
88 CANTREAD(curpsinfo, pr_projid)
89 CANTREAD(curpsinfo, pr_zoneid)
90 CANTREAD(curpsinfo, pr_contract)

92 CANREAD(curlwpsinfo, pr_flag)
93 CANREAD(curlwpsinfo, pr_lwpid)
94 CANREAD(curlwpsinfo, pr_addr)
95 CANREAD(curlwpsinfo, pr_wchan)
96 CANREAD(curlwpsinfo, pr_stype)
97 CANREAD(curlwpsinfo, pr_state)
98 CANREAD(curlwpsinfo, pr_sname)
99 CANREAD(curlwpsinfo, pr_syscall)
100 CANREAD(curlwpsinfo, pr_pri)
101 CANREAD(curlwpsinfo, pr_onpro)
102 CANREAD(curlwpsinfo, pr_bindpro)
103 CANREAD(curlwpsinfo, pr_bindpset)

105 CANTREAD(curlwpsinfo, pr_clname)
106 CANTREAD(curlwpsinfo, pr_lgrp)

108 CANREAD(curcpu, cpu_id)

110 CANTREAD(curcpu, cpu_pset)
111 CANTREAD(curcpu, cpu_chip)
112 CANTREAD(curcpu, cpu_lgrp)
113 CANTREAD(curcpu, cpu_info)

115 /*
116  * We cannot assert one thing or another about the variable "root": for those
117  * with only dtrace_proc, it will be readable in the global but not readable in
118  * the non-global.
119  */
120 MIGHTREADVAR(root)

122 CANREADVAR(cpu)
123 CANTREADVAR(pset)
124 CANTREADVAR(cwd)
125 CANTREADVAR(chip)
126 CANTREADVAR(lgrp)
```

```
128 BEGIN
129 {
130     exit(0);
131 }

133 ERROR
134 /errormsg != ""/
135 {
136     printf("fatal error: %s", errormsg);
137     exit(1);
138 }
```

```
*****
414109 Fri Jun 22 23:09:00 2012
new/usr/src/uts/common/dtrace/dtrace.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
*****
```

```

1 /*
2  * CDDL HEADER START
3 *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7 *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23 * Copyright (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
24 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
25 */
26 */

27 /*
28 * DTrace - Dynamic Tracing for Solaris
29 *
30 * This is the implementation of the Solaris Dynamic Tracing framework
31 * (DTrace). The user-visible interface to DTrace is described at length in
32 * the "Solaris Dynamic Tracing Guide". The interfaces between the libdtrace
33 * library, the in-kernel DTrace framework, and the DTrace providers are
34 * described in the block comments in the <sys/dtrace.h> header file. The
35 * internal architecture of DTrace is described in the block comments in the
36 * <sys/dtrace_impl.h> header file. The comments contained within the DTrace
37 * implementation very much assume mastery of all of these sources; if one has
38 * an unanswered question about the implementation, one should consult them
39 * first.
40 *
41 * The functions here are ordered roughly as follows:
42 *
43 * - Probe context functions
44 * - Probe hashing functions
45 * - Non-probe context utility functions
46 * - Matching functions
47 * - Provider-to-Framework API functions
48 * - Probe management functions
49 * - DIF object functions
50 * - Format functions
51 * - Predicate functions
52 * - ECB functions
53 * - Buffer functions
54 * - Enabling functions
55 * - DOF functions
56 * - Anonymous enabling functions
57 * - Consumer state functions
58 * - Helper functions
59 * - Hook functions
60 * - Driver cookbook functions

```

```

61 /*
62 * Each group of functions begins with a block comment labelled the "DTrace
63 * [Group] Functions", allowing one to find each block by searching forward
64 * on capital-f functions.
65 */
66 #include <sys/errno.h>
67 #include <sys/stat.h>
68 #include <sys/modctl.h>
69 #include <sys/conf.h>
70 #include <sys/system.h>
71 #include <sys/ddi.h>
72 #include <sys/sunddi.h>
73 #include <sys/cpufreq.h>
74 #include <sys/kmem.h>
75 #include <sys/strsubr.h>
76 #include <sys/sysmacros.h>
77 #include <sys/dtrace_impl.h>
78 #include <sys/atomic.h>
79 #include <sys/cmn_err.h>
80 #include <sys/mutex_impl.h>
81 #include <sys/rwlock_impl.h>
82 #include <sys/ctf_api.h>
83 #include <sys/panic.h>
84 #include <sys/priv_impl.h>
85 #include <sys/policy.h>
86 #include <sys/cred_impl.h>
87 #include <sys/procfs_isa.h>
88 #include <sys/taskq.h>
89 #include <sys/mkdev.h>
90 #include <sys/kdi.h>
91 #include <sys/zone.h>
92 #include <sys/socket.h>
93 #include <netinet/in.h>

95 /*
96 * DTrace Tunable Variables
97 *
98 * The following variables may be tuned by adding a line to /etc/system that
99 * includes both the name of the DTrace module ("dtrace") and the name of the
100 * variable. For example:
101 *
102 *   set dtrace:dtrace_destructive_disallow = 1
103 *
104 * In general, the only variables that one should be tuning this way are those
105 * that affect system-wide DTrace behavior, and for which the default behavior
106 * is undesirable. Most of these variables are tunable on a per-consumer
107 * basis using DTrace options, and need not be tuned on a system-wide basis.
108 * When tuning these variables, avoid pathological values; while some attempt
109 * is made to verify the integrity of these variables, they are not considered
110 * part of the supported interface to DTrace, and they are therefore not
111 * checked comprehensively. Further, these variables should not be tuned
112 * dynamically via "mdb -kw" or other means; they should only be tuned via
113 * /etc/system.
114 */
115 int dtrace_destructive_disallow = 0;
116 dtrace_optval_t dtrace_nonroot_maxsize = (16 * 1024 * 1024);
117 size_t dtrace_difo_maxsize = (256 * 1024);
118 dtrace_optval_t dtrace_dof_maxsize = (256 * 1024);
119 size_t dtrace_global_maxsize = (16 * 1024);
120 size_t dtrace_actions_max = (16 * 1024);
121 size_t dtrace_retain_max = 1024;
122 dtrace_optval_t dtrace_helper_actions_max = 1024;
123 dtrace_optval_t dtrace_helper_providers_max = 32;
124 dtrace_optval_t dtrace_dstate_defsize = (1 * 1024 * 1024);
125 size_t dtrace_strsize_default = 256;
126 dtrace_optval_t dtrace_cleanrate_default = 9900990;
/* 101 hz */

```

```

127 dtrace_optval_t dtrace_cleanrate_min = 200000;          /* 5000 hz */
128 dtrace_optval_t dtrace_cleanrate_max = (uint64_t)60 * NANOSEC; /* 1/minute */
129 dtrace_optval_t dtrace_aggrate_default = NANOSEC;        /* 1 hz */
130 dtrace_optval_t dtrace_statusrate_default = NANOSEC;      /* 1 hz */
131 dtrace_optval_t dtrace_statusrate_max = (hrttime_t)10 * NANOSEC; /* 6/minute */
132 dtrace_optval_t dtrace_switchrate_default = NANOSEC;      /* 1 hz */
133 dtrace_optval_t dtrace_nspec_default = 1;
134 dtrace_optval_t dtrace_specszie_default = 32 * 1024;
135 dtrace_optval_t dtrace_stackframes_default = 20;
136 dtrace_optval_t dtrace_ustackframes_default = 20;
137 dtrace_optval_t dtrace_jstackframes_default = 50;
138 dtrace_optval_t dtrace_jstackstrsize_default = 512;
139 int             dtrace_msgrdszie_max = 128;
140 hrttime_t       dtrace_chill_max = 500 * (NANOSEC / MILLISEC); /* 500 ms */
141 hrttime_t       dtrace_chill_interval = NANOSEC;           /* 1000 ms */
142 int             dtrace_devdepth_max = 32;
143 int             dtrace_err_verbose;
144 hrttime_t       dtrace_deadman_interval = NANOSEC;
145 hrttime_t       dtrace_deadman_timeout = (hrttime_t)10 * NANOSEC;
146 hrttime_t       dtrace_deadman_user = (hrttime_t)30 * NANOSEC;
147 hrttime_t       dtrace_unregister_defunct_reap = (hrttime_t)60 * NANOSEC;

149 /*
150  * DTrace External Variables
151  */
152 /* As dtrace(7D) is a kernel module, any DTrace variables are obviously
153  * available to DTrace consumers via the backtick (`) syntax. One of these,
154  * dtrace_zero, is made deliberately so: it is provided as a source of
155  * well-known, zero-filled memory. While this variable is not documented,
156  * it is used by some translators as an implementation detail.
157  */
158 const char      dtrace_zero[256] = { 0 }; /* zero-filled memory */

160 /*
161  * DTrace Internal Variables
162  */
163 static dev_info_t    *dtrace_devi;          /* device info */
164 static vmem_t        *dtrace_arena;         /* probe ID arena */
165 static vmem_t        *dtrace_minor;         /* minor number arena */
166 static taskq_t       *dtrace_taskq;         /* task queue */
167 static dtrace_probe_t **dtrace_probes;       /* array of all probes */
168 static int            dtrace_nprobes;        /* number of probes */
169 static dtrace_provider_t *dtrace_provider;   /* provider list */
170 static dtrace_meta_t  *dtrace_meta_pid;      /* user-land meta provider */
171 static int            dtraceOpens;          /* number of opens */
172 static int            dtrace_helpers;        /* number of helpers */
173 static void           *dtrace_softstate;     /* softstate pointer */
174 static dtrace_hash_t  *dtrace_bymod;         /* probes hashed by module */
175 static dtrace_hash_t  *dtrace_byfunc;        /* probes hashed by function */
176 static dtrace_hash_t  *dtrace_byname;        /* probes hashed by name */
177 static dtrace_toxrange_t *dtrace_toxrange;    /* toxic range array */
178 static int            dtrace_toxranges;       /* number of toxic ranges */
179 static int            dtrace_toxranges_max;    /* size of toxic range array */
180 static dtrace_anon_t  dtrace_anon;          /* anonymous enabling */
181 static kmem_cache_t  *dtrace_state_cache;   /* cache for dynamic state */
182 static uint64_t       dtrace_vtime_references; /* number of vtimestamp refs */
183 static kthread_t     *dtrace_panicked;      /* panicking thread */
184 static dtrace_ecb_t   *dtrace_ecb_create_cache; /* cached created ECB */
185 static dtrace_genid_t dtrace_probeugen;      /* current probe generation */
186 static dtrace_helpers_t *dtrace_deferred_pid; /* deferred helper list */
187 static dtrace_enabling_t *dtrace_retained;    /* list of retained enablings */
188 static dtrace_genid_t  dtrace_retained_gen;   /* current retained enable gen */
189 static dtrace_dynvar_t dtrace_dynhash_sink;   /* end of dynamic hash chains */
190 static int            dtrace_dynvar_failclean; /* dynvars failed to clean */
192 */

```

```

193  * DTrace Locking
194  * DTrace is protected by three (relatively coarse-grained) locks:
195  *
196  * (1) dtrace_lock is required to manipulate essentially any DTrace state,
197  * including enabling state, probes, ECBS, consumer state, helper state,
198  * etc. Importantly, dtrace_lock is not required when in probe context;
199  * probe context is lock-free -- synchronization is handled via the
200  * dtrace_sync() cross call mechanism.
201  *
202  * (2) dtrace_provider_lock is required when manipulating provider state, or
203  * when provider state must be held constant.
204  *
205  * (3) dtrace_meta_lock is required when manipulating meta provider state, or
206  * when meta provider state must be held constant.
207  *
208  * The lock ordering between these three locks is dtrace_meta_lock before
209  * dtrace_provider_lock before dtrace_lock. (In particular, there are
210  * several places where dtrace_provider_lock is held by the framework as it
211  * calls into the providers -- which then call back into the framework,
212  * grabbing dtrace_lock.)
213  *
214  * There are two other locks in the mix: mod_lock and cpu_lock. With respect
215  * to dtrace_provider_lock and dtrace_lock, cpu_lock continues its historical
216  * role as a coarse-grained lock; it is acquired before both of these locks.
217  * With respect to dtrace_meta_lock, its behavior is stranger: cpu_lock must
218  * be acquired between dtrace_meta_lock and any other DTrace locks.
219  * mod_lock is similar with respect to dtrace_provider_lock in that it must be
220  * acquired between dtrace_provider_lock and dtrace_lock.
221  */
222 static kmutex_t        dtrace_lock;          /* probe state lock */
223 static kmutex_t        dtrace_provider_lock; /* provider state lock */
224 static kmutex_t        dtrace_meta_lock;     /* meta-provider state lock */

226 /*
227  * DTrace Provider Variables
228  */
229 /* These are the variables relating to DTrace as a provider (that is, the
230  * provider of the BEGIN, END, and ERROR probes).
231 */
232 static dtrace_pattr_t  dtrace_provider_attr = {
233 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
234 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
235 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
236 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
237 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
238 };
unchanged portion omitted
347 #define DT_BSWAP_8(x) ((x) & 0xff)
348 #define DT_BSWAP_16(x) ((DT_BSWAP_8(x) << 8) | DT_BSWAP_8((x) >> 8))
349 #define DT_BSWAP_32(x) ((DT_BSWAP_16(x) << 16) | DT_BSWAP_16((x) >> 16))
350 #define DT_BSWAP_64(x) ((DT_BSWAP_32(x) << 32) | DT_BSWAP_32((x) >> 32))
352 #define DT_MASK_LO 0x00000000FFFFFFFF
354 #define DTRACE_STORE(type, tomax, offset, what) \
355     *((type *)((uintptr_t)(tomax) + (uintptr_t)(offset))) = (type)(what);
357 #ifndef __i386
358 #define DTRACE_ALIGNCHECK(addr, size, flags) \
359     if (addr & (size - 1)) { \
360         *flags |= CPU_DTRACE_BADALIGN; \
361         cpu_core[CPU->cpu_id].cpuc_dtrace_illval = addr; \
362         return (0); \
363     } \
364 #else

```

```

365 #define DTRACE_ALIGNCHECK(addr, size, flags)
366 #endif
368 /*
369  * Test whether a range of memory starting at testaddr of size testsz falls
370  * within the range of memory described by addr, sz. We take care to avoid
371  * problems with overflow and underflow of the unsigned quantities, and
372  * disallow all negative sizes. Ranges of size 0 are allowed.
373 */
374 #define DTRACE_INRANGE(testaddr, testsz, baseaddr, basesz) \
375     (((testaddr) - (uintptr_t)(baseaddr)) < (basesz) && \
376      ((testaddr) + (testsz) - (uintptr_t)(baseaddr)) <= (basesz) && \
375      ((testaddr) - (baseaddr) < (basesz) && \
376      ((testaddr) + (testsz) - (baseaddr)) <= (basesz) && \
377      (testaddr) + (testsz) >= (testaddr))
379 /*
380  * Test whether alloc_sz bytes will fit in the scratch region. We isolate
381  * alloc_sz on the righthand side of the comparison in order to avoid overflow
382  * or underflow in the comparison with it. This is simpler than the INRANGE
383  * check above, because we know that the dtms_scratch_ptr is valid in the
384  * range. Allocations of size zero are allowed.
385 */
386 #define DTRACE_INSCRATCH(mstate, alloc_sz) \
387     (((mstate)->dtms_scratch_base + (mstate)->dtms_scratch_size - \
388      (mstate)->dtms_scratch_ptr) >= (alloc_sz))
389
390 #define DTRACE_LOADFUNC(bits)
391 /*CSTYLED*/
392 uint##bits##_t
393 dtrace_load##bits##(uintptr_t addr)
394 {
395     size_t size = bits / NBBY;
396 /*CSTYLED*/
397     uint##bits##_t rval;
398     int i;
399     volatile uint16_t *flags = (volatile uint16_t *)
400         &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
401
402     DTRACE_ALIGNCHECK(addr, size, flags);
403
404     for (i = 0; i < dtrace_toxranges; i++) {
405         if (addr >= dtrace_toxrange[i].dtt_limit)
406             continue;
407
408         if (addr + size <= dtrace_toxrange[i].dtt_base)
409             continue;
410
411         /*
412          * This address falls within a toxic region; return 0.
413          */
414         *flags |= CPU_DTRACE_BADADDR;
415         cpu_core[CPU->cpu_id].cpuc_dtrace_illval = addr;
416         return (0);
417     }
418
419     *flags |= CPU_DTRACE_NOFAULT;
420 /*CSTYLED*/
421     rval = *((volatile uint##bits##_t *)addr);
422     *flags &= ~CPU_DTRACE_NOFAULT;
423
424     return (!(*flags & CPU_DTRACE_FAULT) ? rval : 0);
425 }
426
427 #ifdef _LP64
428 #define dtrace_loadptr dtrace_load64

```

```

429 #else
430 #define dtrace_loadptr dtrace_load32
431 #endif
432
433 #define DTRACE_DYNHASH_FREE 0
434 #define DTRACE_DYNHASH_SINK 1
435 #define DTRACE_DYNHASH_VALID 2
436
437 #define DTRACE_MATCH_FAIL -1
438 #define DTRACE_MATCH_NEXT 0
439 #define DTRACE_MATCH_DONE 1
440 #define DTRACE_ANCHORED(probe) ((probe)->dptr_func[0] != '\0')
441 #define DTRACE_STATE_ALIGN 64
442
443 #define DTRACE_FLAGS2FLT(flags) \
444     (((flags) & CPU_DTRACE_BADADDR) ? DTRACEFLT_BADADDR : \
445      (((flags) & CPU_DTRACE_ILOP) ? DTRACEFLT_ILOP : \
446      (((flags) & CPU_DTRACE_DIVZERO) ? DTRACEFLT_DIVZERO : \
447      (((flags) & CPU_DTRACE_KPRIV) ? DTRACEFLT_KPRIV : \
448      (((flags) & CPU_DTRACE_UPRIV) ? DTRACEFLT_UPRIV : \
449      (((flags) & CPU_DTRACE_TUOPOFLOW) ? DTRACEFLT_TUOPOFLOW : \
450      (((flags) & CPU_DTRACE_BADALIGN) ? DTRACEFLT_BADALIGN : \
451      (((flags) & CPU_DTRACE_NOSCRATCH) ? DTRACEFLT_NOSCRATCH : \
452      (((flags) & CPU_DTRACE_BADSTACK) ? DTRACEFLT_BADSTACK : \
453      DTRACEFLT_UNKNOWN))
454
455 #define DTRACEACT_ISSTRING(act) \
456     (((act)->dta_kind == DTRACEACT_DIFEXPR && \
457      (act)->dta_difo->dtodo_rtype.dtdt_kind == DIF_TYPE_STRING))
458
459 static size_t dtrace_strlen(const char *, size_t);
460 static dtrace_probe_t *dtrace_probe_lookup_id(dtrace_id_t id);
461 static void dtrace_enabling_provide(dtrace_provider_t *);
462 static int dtrace_enabling_match(dtrace_enabling_t *, int *);
463 static void dtrace_enabling_matchall(void);
464 static void dtrace_enabling_reap(void);
465 static dtrace_state_t *dtrace_anon_grab(void);
466 static uint64_t dtrace_helper(int, dtrace_mstate_t *,
467     dtrace_state_t *, uint64_t, uint64_t);
468 static dtrace_helpers_t *dtrace_helpers_create(proc_t *);
469 static void dtrace_buffer_drop(dtrace_buffer_t *);
470 static int dtrace_buffer_consumed(dtrace_buffer_t *, hrtime_t when);
471 static intptr_t dtrace_buffer_reserve(dtrace_buffer_t *, size_t, size_t,
472     dtrace_state_t *, dtrace_mstate_t *);
473 static int dtrace_state_option(dtrace_state_t *, dtrace_optid_t,
474     dtrace_optval_t);
475 static int dtrace_ecb_create_enable(dtrace_probe_t *, void *);
476 static void dtrace_helper_provider_destroy(dtrace_helper_provider_t *);
477 static int dtrace_priv_proc(dtrace_state_t *, dtrace_mstate_t *);
478
479 /*
480  * DTrace Probe Context Functions
481  *
482  * These functions are called from probe context. Because probe context is
483  * any context in which C may be called, arbitrarily locks may be held,
484  * interrupts may be disabled, we may be in arbitrary dispatched state, etc.
485  * As a result, functions called from probe context may only call other DTrace
486  * support functions -- they may not interact at all with the system at large.
487  * (Note that the ASSERT macro is made probe-context safe by redefining it in
488  * terms of dtrace_assfail(), a probe-context safe function.) If arbitrary
489  * loads are to be performed from probe context, they must be in terms of
490  * the safe dtrace_load*() variants.
491  *
492  * Some functions in this block are not actually called from probe context;
493  * for these functions, there will be a comment above the function reading
494  * "Note: not called from probe context."

```

```

495 */
496 void
497 dtrace_panic(const char *format, ...)
498 {
499     va_list alist;
501     va_start(alist, format);
502     dtrace_vpanic(format, alist);
503     va_end(alist);
504 }
unchanged portion omitted
600 /*
601 * Check to see if the address is within a memory region to which a store may
602 * be issued. This includes the DTrace scratch areas, and any DTrace variable
603 * region. The caller of dtrace_canstore() is responsible for performing any
604 * alignment checks that are needed before stores are actually executed.
605 */
606 static int
607 dtrace_canstore(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
608                  dtrace_vstate_t *vstate)
609 {
610     /*
611      * First, check to see if the address is in scratch space...
612      */
613     if (DTRACE_INRANGE(addr, sz, mstate->dtms_scratch_base,
614                         mstate->dtms_scratch_size))
615         return (1);
616     /*
617      * Now check to see if it's a dynamic variable. This check will pick
618      * up both thread-local variables and any global dynamically-allocated
619      * variables.
620      */
621     if (DTRACE_INRANGE(addr, sz, vstate->dtvs_dynvars.dtds_base,
622                        if (DTRACE_INRANGE(addr, sz, (uintptr_t)vstate->dtvs_dynvars.dtds_base,
623                               vstate->dtvs_dynvars.dtds_size)) {
624         dtrace_dstate_t *dstate = &vstate->dtvs_dynvars;
625         uintptr_t base = (uintptr_t)dstate->dtds_base +
626                           (dstate->dtds_hashsize * sizeof (dtrace_dynhash_t));
627         uintptr_t chunkoffs;
628
629         /*
630          * Before we assume that we can store here, we need to make
631          * sure that it isn't in our metadata -- storing to our
632          * dynamic variable metadata would corrupt our state. For
633          * the range to not include any dynamic variable metadata,
634          * it must:
635          *
636          *   (1) Start above the hash table that is at the base of
637          *       the dynamic variable space
638          *
639          *   (2) Have a starting chunk offset that is beyond the
640          *       dtrace_dynvar_t that is at the base of every chunk
641          *
642          *   (3) Not span a chunk boundary
643          *
644          */
645     if (addr < base)
646         return (0);
647
648     chunkoffs = (addr - base) % dstate->dtds_chunksize;
649
650     if (chunkoffs < sizeof (dtrace_dynvar_t))
651         return (0);

```

```

653             if (chunkoffs + sz > dstate->dtds_chunksize)
654                 return (0);
655
656             return (1);
657         }
658
659         /*
660          * Finally, check the static local and global variables. These checks
661          * take the longest, so we perform them last.
662          */
663         if (dtrace_canstore_statvar(addr, sz,
664                                     vstate->dtvs_locals, vstate->dtvs_nlocals))
665             return (1);
666
667         if (dtrace_canstore_statvar(addr, sz,
668                                     vstate->dtvs_globals, vstate->dtvs_nglobals))
669             return (1);
670
671         return (0);
672     }
673
674     /*
675      * Convenience routine to check to see if the address is within a memory
676      * region in which a load may be issued given the user's privilege level;
677      * if not, it sets the appropriate error flags and loads 'addr' into the
678      * illegal value slot.
679      */
680
681     /* DTrace subroutines (DIF_SUBR_*) should use this helper to implement
682      * appropriate memory access protection.
683      */
684     static int
685     dtrace_canload(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
686                    dtrace_vstate_t *vstate)
687     {
688         volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
689
690         /*
691          * If we hold the privilege to read from kernel memory, then
692          * everything is readable.
693          */
694         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
695             return (1);
696
697         /*
698          * You can obviously read that which you can store.
699          */
700         if (dtrace_canstore(addr, sz, mstate, vstate))
701             return (1);
702
703         /*
704          * We're allowed to read from our own string table.
705          */
706         if (DTRACE_INRANGE(addr, sz, mstate->dtms_difo->dtdo_strtab,
707                            if (DTRACE_INRANGE(addr, sz, (uintptr_t)mstate->dtms_difo->dtdo_strlen))
708                                return (1);
709
710         if (vstate->dtvs_state != NULL &&
711             dtrace_priv_proc(vstate->dtvs_state, mstate)) {
712             proc_t *p;
713
714             /*
715              * When we have privileges to the current process, there are
716              * several context-related kernel structures that are safe to
717              * read, even absent the privilege to read from kernel memory.

```

```

718     * These reads are safe because these structures contain only
719     * state that (1) we're permitted to read, (2) is harmless or
720     * (3) contains pointers to additional kernel state that we're
721     * not permitted to read (and as such, do not present an
722     * opportunity for privilege escalation). Finally (and
723     * critically), because of the nature of their relation with
724     * the current thread context, the memory associated with these
725     * structures cannot change over the duration of probe context,
726     * and it is therefore impossible for this memory to be
727     * deallocated and reallocated as something else while it's
728     * being operated upon.
729 */
730 if (DTRACE_INRANGE(addr, sz, curthread, sizeof (kthread_t)))
731     return (1);

733 if ((p = curthread->t_proc) != NULL && DTRACE_INRANGE(addr,
734     sz, curthread->t_proc, sizeof (proc_t))) {
735     return (1);
736 }

738 if (curthread->t_cred != NULL && DTRACE_INRANGE(addr, sz,
739     curthread->t_cred, sizeof (cred_t))) {
740     return (1);
741 }

743 if (p != NULL && p->p_pidp != NULL && DTRACE_INRANGE(addr, sz,
744     &(p->p_pidp->pid_id), sizeof (pid_t))) {
745     return (1);
746 }

748 if (curthread->t_cpu != NULL && DTRACE_INRANGE(addr, sz,
749     curthread->t_cpu, offsetof(cpu_t, cpu_pause_thread))) {
750     return (1);
751 }
752 }

754 DTRACE_CPUFLAG_SET(CPU_DTRACE_KPRIV);
755 *illval = addr;
756 return (0);
757 }



---


unchanged_portion_omitted

2851 /*
2852  * This function implements the DIF emulator's variable lookups. The emulator
2853  * passes a reserved variable identifier and optional built-in array index.
2854  */
2855 static uint64_t
2856 dtrace_dif_variable(dtrace_mstate_t *mstate, dtrace_state_t *state, uint64_t v,
2857     uint64_t ndx)
2858 {
2859     /*
2860     * If we're accessing one of the uncached arguments, we'll turn this
2861     * into a reference in the args array.
2862     */
2863     if (v >= DIF_VAR_ARG0 && v <= DIF_VAR_ARG9) {
2864         ndx = v - DIF_VAR_ARG0;
2865         v = DIF_VAR_ARGS;
2866     }

2867     switch (v) {
2868     case DIF_VAR_ARGS:
2869         if (!(mstate->dtms_access & DTRACE_ACCESS_ARGS)) {
2870             cpu_core[CPU->cpu_id].cpuc_dtrace_flags |=
2871                 CPU_DTRACE_KPRIV;
2872             return (0);
2873         }
2874     }

```

```

2876     ASSERT(mstate->dtms_present & DTRACE_MSTATE_ARGS);
2877     if (ndx >= sizeof (mstate->dtms_arg))
2878         sizeof (mstate->dtms_arg[0])) {
2879         int aframes = mstate->dtms_probe->dtpr_aframes + 2;
2880         dtrace_provider_t *pv;
2881         uint64_t val;

2883         pv = mstate->dtms_probe->dtpr_provider;
2884         if (pv->dtpv_pops.dtps_getargval != NULL)
2885             val = pv->dtpv_pops.dtps_getargval(pv->dtpv_arg,
2886             mstate->dtms_probe->dtpr_id,
2887             mstate->dtms_probe->dtpr_arg, ndx, aframes);
2888         else
2889             val = dtrace_getarg(ndx, aframes);

2891         /*
2892          * This is regrettably required to keep the compiler
2893          * from tail-optimizing the call to dtrace_getarg().
2894          * The condition always evaluates to true, but the
2895          * compiler has no way of figuring that out a priori.
2896          * (None of this would be necessary if the compiler
2897          * could be relied upon to _always_ tail-optimize
2898          * the call to dtrace_getarg() -- but it can't.)
2899         */
2900         if (mstate->dtms_probe != NULL)
2901             return (val);

2903         ASSERT(0);
2904     }

2906     return (mstate->dtms_arg[ndx]);
2907 }

2908 case DIF_VARUREGS: {
2909     klpw_t *lwp;
2910
2911     if (!dtrace_priv_proc(state, mstate))
2912         return (0);
2913
2914     if ((lwp = curthread->t_lwp) == NULL) {
2915         DTRACE_CPUFLAG_SET(CPU_DTRACE_BADADDR);
2916         cpu_core[CPU->cpu_id].cpuc_dtrace_illval = NULL;
2917         return (0);
2918     }
2919
2920     return (dtrace_getreg(lwp->lwp_regs, ndx));
2921 }

2923 case DIF_VARVMREGS: {
2924     uint64_t rval;
2925
2926     if (!dtrace_priv_kernel(state))
2927         return (0);
2928
2929     DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
2930
2931     rval = dtrace_getvmreg(ndx,
2932         &cpu_core[CPU->cpu_id].cpuc_dtrace_flags);
2933
2934     DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
2935
2936     return (rval);
2937 }

2939 case DIF_VAR_CURTHREAD:
2940     if (!dtrace_priv_proc(state, mstate))

```

```

2895         if (!dtrace_priv_kernel(state))
2941             return (0);
2942         return ((uint64_t)(uintptr_t)curthread);
2944
2945     case DIF_VAR_TIMESTAMP:
2946         if (!(mstate->dtms_present & DTRACE_MSTATE_TIMESTAMP)) {
2947             mstate->dtms_timestamp = dtrace_gethrtime();
2948             mstate->dtms_present |= DTRACE_MSTATE_TIMESTAMP;
2949         }
2950         return (mstate->dtms_timestamp);
2951
2952     case DIF_VAR_VTIMESTAMP:
2953         ASSERT(dtrace_vtime_references != 0);
2954         return (curthread->t_dtrace_vtime);
2955
2956     case DIF_VAR_WALLTIMESTAMP:
2957         if (!(mstate->dtms_present & DTRACE_MSTATE_WALLTIMESTAMP)) {
2958             mstate->dtms_walltimestamp = dtrace_gethrstime();
2959             mstate->dtms_present |= DTRACE_MSTATE_WALLTIMESTAMP;
2960         }
2961         return (mstate->dtms_walltimestamp);
2962
2963     case DIF_VAR_IPL:
2964         if (!dtrace_priv_kernel(state))
2965             return (0);
2966         if (!(mstate->dtms_present & DTRACE_MSTATE_IPL)) {
2967             mstate->dtms_ipl = dtrace_getipl();
2968             mstate->dtms_present |= DTRACE_MSTATE_IPL;
2969         }
2970         return (mstate->dtms_ipl);
2971
2972     case DIF_VAR_EPID:
2973         ASSERT(mstate->dtms_present & DTRACE_MSTATE_EPID);
2974         return (mstate->dtms_epid);
2975
2976     case DIF_VAR_ID:
2977         ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
2978         return (mstate->dtms_probe->dtpr_id);
2979
2980     case DIF_VAR_STACKDEPTH:
2981         if (!dtrace_priv_kernel(state))
2982             return (0);
2983         if (!(mstate->dtms_present & DTRACE_MSTATE_STACKDEPTH)) {
2984             int aframes = mstate->dtms_probe->dtpr_aframes + 2;
2985             mstate->dtms_stackdepth = dtrace_getstackdepth(aframes);
2986             mstate->dtms_present |= DTRACE_MSTATE_STACKDEPTH;
2987         }
2988         return (mstate->dtms_stackdepth);
2989
2990     case DIF_VAR_USTACKDEPTH:
2991         if (!dtrace_priv_proc(state, mstate))
2992             return (0);
2993         if (!(mstate->dtms_present & DTRACE_MSTATE_USTACKDEPTH)) {
2994             /*
2995             * See comment in DIF_VAR_PID.
2996             */
2997             if (DTRACE_ANCHORED(mstate->dtms_probe) &&
2998                 CPU_ON_INTR(CPU)) {
2999                 mstate->dtms_ustackdepth = 0;
3000             } else {
3001                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3002                 mstate->dtms_ustackdepth =
3003                     dtrace_getustackdepth();
3004                 DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3005             }

```

```

3006                         mstate->dtms_present |= DTRACE_MSTATE_USTACKDEPTH;
3007
3008         }
3009         return (mstate->dtms_ustackdepth);
3010
3011     case DIF_VAR_CALLER:
3012         if (!dtrace_priv_kernel(state))
3013             return (0);
3014         if (!(mstate->dtms_present & DTRACE_MSTATE_CALLER)) {
3015             int aframes = mstate->dtms_probe->dtpr_aframes + 2;
3016             if (!DTRACE_ANCHORED(mstate->dtms_probe)) {
3017                 /*
3018                 * If this is an unanchored probe, we are
3019                 * required to go through the slow path:
3020                 * dtrace_caller() only guarantees correct
3021                 * results for anchored probes.
3022                 */
3023             pc_t caller[2];
3024             dtrace_getpcstack(caller, 2, aframes,
3025                               (uint32_t *)(uintptr_t)mstate->dtms_arg[0]);
3026             mstate->dtms_caller = caller[1];
3027         } else if ((mstate->dtms_caller ==
3028             dtrace_caller(aframes)) == -1) {
3029             /*
3030             * We have failed to do this the quick way;
3031             * we must resort to the slower approach of
3032             * calling dtrace_getpcstack().
3033             */
3034         pc_t caller;
3035             dtrace_getpcstack(&caller, 1, aframes, NULL);
3036             mstate->dtms_caller = caller;
3037         }
3038         mstate->dtms_present |= DTRACE_MSTATE_CALLER;
3039         return (mstate->dtms_caller);
3040
3041     case DIF_VAR_UCALLER:
3042         if (!dtrace_priv_proc(state, mstate))
3043             return (0);
3044         if (!(mstate->dtms_present & DTRACE_MSTATE_UCALLER)) {
3045             uint64_t ustack[3];
3046             /*
3047             * dtrace_getpcstack() fills in the first uint64_t
3048             * with the current PID. The second uint64_t will
3049             * be the program counter at user-level. The third
3050             * uint64_t will contain the caller, which is what
3051             * we're after.
3052             */
3053             ustack[2] = NULL;
3054             DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3055             dtrace_getpcstack(ustack, 3);
3056             DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3057             mstate->dtms_ucaller = ustack[2];
3058             mstate->dtms_present |= DTRACE_MSTATE_UCALLER;
3059         }
3060         return (mstate->dtms_ucaller);
3061
3062     case DIF_VAR_PROBEPROV:
3063         ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3064         return (dtrace_dif_varstr(
3065

```

```

3072             (uintptr_t)mstate->dtms_probe->dtpr_provider->dtpv_name,
3073             state, mstate));
3075
3076     case DIF_VAR_PROBEMOD:
3077         ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3078         return (dtrace_dif_varstr(
3079             (uintptr_t)mstate->dtms_probe->dtpr_mod,
3080             state, mstate));
3081
3082     case DIF_VAR_PROBEFUNC:
3083         ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3084         return (dtrace_dif_varstr(
3085             (uintptr_t)mstate->dtms_probe->dtpr_func,
3086             state, mstate));
3087
3088     case DIF_VAR_PROBENAME:
3089         ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3090         return (dtrace_dif_varstr(
3091             (uintptr_t)mstate->dtms_probe->dtpr_name,
3092             state, mstate));
3093
3094     case DIF_VAR_PID:
3095         if (!dtrace_priv_proc(state, mstate))
3096             return (0);
3097
3098         /*
3099          * Note that we are assuming that an unanchored probe is
3100          * always due to a high-level interrupt. (And we're assuming
3101          * that there is only a single high level interrupt.)
3102         */
3103         if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3104             return (pid0.pid_id);
3105
3106         /*
3107          * It is always safe to dereference one's own t_procp pointer:
3108          * it always points to a valid, allocated proc structure.
3109          * Further, it is always safe to dereference the p_pidp member
3110          * of one's own proc structure. (These are truisms because
3111          * threads and processes don't clean up their own state --
3112          * they leave that task to whomever reaps them.)
3113         */
3114         return ((uint64_t)curthread->t_procp->p_pidp->pid_id);
3115
3116     case DIF_VAR_PPID:
3117         if (!dtrace_priv_proc(state, mstate))
3118             return (0);
3119
3120         /*
3121          * See comment in DIF_VAR_PID.
3122         */
3123         if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3124             return (pid0.pid_id);
3125
3126         /*
3127          * It is always safe to derference one's own t_procp pointer:
3128          * it always points to a valid, allocated proc structure.
3129          * (This is true because threads don't clean up their own
3130          * state -- they leave that task to whomever reaps them.)
3131         */
3132         return ((uint64_t)curthread->t_procp->p_ppid);
3133
3134     case DIF_VAR_TID:
3135         /*
3136          * See comment in DIF_VAR_PID.
3137         */
3138         if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))

```

```

3138                     return (0);
3140
3141                     return ((uint64_t)curthread->t_tid);
3142
3143     case DIF_VAR_EXECNAME:
3144         if (!dtrace_priv_proc(state, mstate))
3145             return (0);
3146
3147         /*
3148          * See comment in DIF_VAR_PID.
3149         */
3150         if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3151             return ((uint64_t)(uintptr_t)p0.p_user.u_comm);
3152
3153         /*
3154          * It is always safe to dereference one's own t_procp pointer:
3155          * it always points to a valid, allocated proc structure.
3156          * (This is true because threads don't clean up their own
3157          * state -- they leave that task to whomever reaps them.)
3158         */
3159         return (dtrace_dif_varstr(
3160             (uintptr_t)curthread->t_procp->p_user.u_comm,
3161             state, mstate));
3162
3163     case DIF_VAR_ZONENAME:
3164         if (!dtrace_priv_proc(state, mstate))
3165             return (0);
3166
3167         /*
3168          * See comment in DIF_VAR_PID.
3169         */
3170         if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3171             return ((uint64_t)(uintptr_t)p0.p_zone->zone_name);
3172
3173         /*
3174          * It is always safe to dereference one's own t_procp pointer:
3175          * it always points to a valid, allocated proc structure.
3176          * (This is true because threads don't clean up their own
3177          * state -- they leave that task to whomever reaps them.)
3178         */
3179         return (dtrace_dif_varstr(
3180             (uintptr_t)curthread->t_procp->p_zone->zone_name,
3181             state, mstate));
3182
3183     case DIF_VAR_UID:
3184         if (!dtrace_priv_proc(state, mstate))
3185             return (0);
3186
3187         /*
3188          * See comment in DIF_VAR_PID.
3189         */
3190         if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3191             return ((uint64_t)p0.p_cred->cr_uid);
3192
3193         /*
3194          * It is always safe to dereference one's own t_procp pointer:
3195          * it always points to a valid, allocated proc structure.
3196          * (This is true because threads don't clean up their own
3197          * state -- they leave that task to whomever reaps them.)
3198          *
3199          * Additionally, it is safe to dereference one's own process
3200          * credential, since this is never NULL after process birth.
3201         */
3202         return ((uint64_t)curthread->t_procp->p_cred->cr_uid);
3203
3204     case DIF_VAR_GID:

```

```

3204     if (!dtrace_priv_proc(state, mstate))
3205         return (0);
3206
3207     /*
3208      * See comment in DIF_VAR_PID.
3209      */
3210     if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3211         return ((uint64_t)p0.p_cred->cr_gid);
3212
3213     /*
3214      * It is always safe to dereference one's own t_procp pointer:
3215      * it always points to a valid, allocated proc structure.
3216      * (This is true because threads don't clean up their own
3217      * state -- they leave that task to whomever reaps them.)
3218      *
3219      * Additionally, it is safe to dereference one's own process
3220      * credential, since this is never NULL after process birth.
3221      */
3222     return ((uint64_t)curthread->t_procp->p_cred->cr_gid);
3223
3224 case DIF_VAR_ERRNO:
3225     klwp_t *lwp;
3226     if (!dtrace_priv_proc(state, mstate))
3227         return (0);
3228
3229     /*
3230      * See comment in DIF_VAR_PID.
3231      */
3232     if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3233         return (0);
3234
3235     /*
3236      * It is always safe to dereference one's own t_lwp pointer in
3237      * the event that this pointer is non-NULL. (This is true
3238      * because threads and lwp's don't clean up their own state --
3239      * they leave that task to whomever reaps them.)
3240      */
3241     if ((lwp = curthread->t_lwp) == NULL)
3242         return (0);
3243
3244     return ((uint64_t)lwp->lwp_errno);
3245 }
3246 default:
3247     DTRACE_CPUFLAG_SET(CPU_DTRACE_ILOP);
3248     return (0);
3249 }
3250 }



---


unchanged_portion_omitted_

4783 /*
4784  * Emulate the execution of DTrace IR instructions specified by the given
4785  * DIF object. This function is deliberately void of assertions as all of
4786  * the necessary checks are handled by a call to dtrace_difo_validate().
4787 */
4788 static uint64_t
4789 dtrace_dif_emulate(dtrace_difo_t *difo, dtrace_mstate_t *mstate,
4790                     dtrace_vstate_t *vstate, dtrace_state_t *state)
4791 {
4792     const dif_instr_t *text = difo->dtdo_buf;
4793     const uint_t textlen = difo->dtdo_len;
4794     const char *strtab = difo->dtdo_strtab;
4795     const uint64_t *inttab = difo->dtdo_inttab;
4796
4797     uint64_t rval = 0;
4798     dtrace_statvar_t *svar;
4799     dtrace_dstate_t *dstate = &vstate->dtvs_dynvars;

```

```

4800     dtrace_dify_t *v;
4801     volatile uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
4802     volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
4803
4804     dtrace_key_t tupregs[DIF_DTR_NREGS + 2]; /* +2 for thread and id */
4805     uint64_t regs[DIF_DIR_NREGS];
4806     uint64_t *tmp;
4807
4808     uint8_t cc_n = 0, cc_z = 0, cc_v = 0, cc_c = 0;
4809     int64_t cc_r;
4810     uint_t pc = 0, id, opc;
4811     uint8_t ttop = 0;
4812     dif_instr_t instr;
4813     uint_t r1, r2, rd;
4814
4815     /*
4816      * We stash the current DIF object into the machine state: we need it
4817      * for subsequent access checking.
4818      */
4819     mstate->dtms_difo = difo;
4820
4821     regs[DIF_REG_R0] = 0; /* %r0 is fixed at zero */
4822
4823     while (pc < textlen && !(flags & CPU_DTRACE_FAULT)) {
4824         opc = pc;
4825
4826         instr = text[pc++];
4827         r1 = DIF_INSTR_R1(instr);
4828         r2 = DIF_INSTR_R2(instr);
4829         rd = DIF_INSTR_RD(instr);
4830
4831         switch (DIF_INSTR_OP(instr)) {
4832             case DIF_OP_OR:
4833                 regs[rd] = regs[r1] | regs[r2];
4834                 break;
4835             case DIF_OP_XOR:
4836                 regs[rd] = regs[r1] ^ regs[r2];
4837                 break;
4838             case DIF_OP_AND:
4839                 regs[rd] = regs[r1] & regs[r2];
4840                 break;
4841             case DIF_OP_SLL:
4842                 regs[rd] = regs[r1] << regs[r2];
4843                 break;
4844             case DIF_OP_SRL:
4845                 regs[rd] = regs[r1] >> regs[r2];
4846                 break;
4847             case DIF_OP_SUB:
4848                 regs[rd] = regs[r1] - regs[r2];
4849                 break;
4850             case DIF_OP_ADD:
4851                 regs[rd] = regs[r1] + regs[r2];
4852                 break;
4853             case DIF_OP_MUL:
4854                 regs[rd] = regs[r1] * regs[r2];
4855                 break;
4856             case DIF_OP_SDIV:
4857                 if (regs[r2] == 0) {
4858                     regs[rd] = 0;
4859                     *flags |= CPU_DTRACE_DIVZERO;
4860                 } else {
4861                     regs[rd] = (int64_t)regs[r1] /
4862                               (int64_t)regs[r2];
4863                 }
4864                 break;

```

```

4866
4867     case DIF_OP_UDIV:
4868         if (regs[r2] == 0) {
4869             regs[rd] = 0;
4870             *flags |= CPU_DTRACE_DIVZERO;
4871         } else {
4872             regs[rd] = regs[r1] / regs[r2];
4873         }
4874         break;

4875     case DIF_OP_SREM:
4876         if (regs[r2] == 0) {
4877             regs[rd] = 0;
4878             *flags |= CPU_DTRACE_DIVZERO;
4879         } else {
4880             regs[rd] = (int64_t)regs[r1] %
4881                         (int64_t)regs[r2];
4882         }
4883         break;

4884     case DIF_OP_UREM:
4885         if (regs[r2] == 0) {
4886             regs[rd] = 0;
4887             *flags |= CPU_DTRACE_DIVZERO;
4888         } else {
4889             regs[rd] = regs[r1] % regs[r2];
4890         }
4891         break;

4892     case DIF_OP_NOT:
4893         regs[rd] = ~regs[r1];
4894         break;
4895     case DIF_OP_MOV:
4896         regs[rd] = regs[r1];
4897         break;
4898     case DIF_OP_CMP:
4899         cc_r = regs[r1] - regs[r2];
4900         cc_n = cc_r < 0;
4901         cc_z = cc_r == 0;
4902         cc_v = 0;
4903         cc_c = regs[r1] < regs[r2];
4904         break;
4905     case DIF_OP_TST:
4906         cc_n = cc_v = cc_c = 0;
4907         cc_z = regs[r1] == 0;
4908         break;
4909     case DIF_OP_BA:
4910         pc = DIF_INSTR_LABEL(instr);
4911         break;
4912     case DIF_OP_BB:
4913         if (cc_z)
4914             pc = DIF_INSTR_LABEL(instr);
4915         break;
4916     case DIF_OP_BNE:
4917         if (cc_z == 0)
4918             pc = DIF_INSTR_LABEL(instr);
4919         break;
4920     case DIF_OP_BG:
4921         if ((cc_z | (cc_n ^ cc_v)) == 0)
4922             pc = DIF_INSTR_LABEL(instr);
4923         break;
4924     case DIF_OP_BGU:
4925         if ((cc_c | cc_z) == 0)
4926             pc = DIF_INSTR_LABEL(instr);
4927         break;
4928     case DIF_OP_BGE:
4929         if ((cc_n ^ cc_v) == 0)
4930

```

```

4931                                         pc = DIF_INSTR_LABEL(instr);
4932                                         break;
4933     case DIF_OP_BGEU:
4934         if (cc_c == 0)
4935             pc = DIF_INSTR_LABEL(instr);
4936         break;
4937     case DIF_OP_BL:
4938         if (cc_n ^ cc_v)
4939             pc = DIF_INSTR_LABEL(instr);
4940         break;
4941     case DIF_OP_BLU:
4942         if (cc_c)
4943             pc = DIF_INSTR_LABEL(instr);
4944         break;
4945     case DIF_OP_BLE:
4946         if (cc_z | (cc_n ^ cc_v))
4947             pc = DIF_INSTR_LABEL(instr);
4948         break;
4949     case DIF_OP_BLEU:
4950         if (cc_c | cc_z)
4951             pc = DIF_INSTR_LABEL(instr);
4952         break;
4953     case DIF_OP_RLDSB:
4954         if (!dtrace_canload(regs[r1], 1, mstate, vstate))
4955             if (!dtrace_canstore(regs[r1], 1, mstate, vstate)) {
4956                 *flags |= CPU_DTRACE_KPRIV;
4957                 *illval = regs[r1];
4958                 break;
4959             }
4960             /*FALLTHROUGH*/
4961     case DIF_OP_LDSB:
4962         regs[rd] = (int8_t)dtrace_load8(regs[r1]);
4963         break;
4964     case DIF_OP_RLD SH:
4965         if (!dtrace_canload(regs[r1], 2, mstate, vstate))
4966             if (!dtrace_canstore(regs[r1], 2, mstate, vstate)) {
4967                 *flags |= CPU_DTRACE_KPRIV;
4968                 *illval = regs[r1];
4969                 break;
4970             }
4971             /*FALLTHROUGH*/
4972     case DIF_OP_LD SH:
4973         regs[rd] = (int16_t)dtrace_load16(regs[r1]);
4974         break;
4975     case DIF_OP_RLD SW:
4976         if (!dtrace_canload(regs[r1], 4, mstate, vstate))
4977             if (!dtrace_canstore(regs[r1], 4, mstate, vstate)) {
4978                 *flags |= CPU_DTRACE_KPRIV;
4979                 *illval = regs[r1];
4980                 break;
4981             }
4982             /*FALLTHROUGH*/
4983     case DIF_OP_LDSW:
4984         regs[rd] = (int32_t)dtrace_load32(regs[r1]);
4985         break;
4986     case DIF_OP_RLDUB:
4987         if (!dtrace_canload(regs[r1], 1, mstate, vstate))
4988             if (!dtrace_canstore(regs[r1], 1, mstate, vstate)) {
4989                 *flags |= CPU_DTRACE_KPRIV;
4990                 *illval = regs[r1];
4991                 break;
4992             }
4993             /*FALLTHROUGH*/
4994     case DIF_OP_LDUB:
4995         regs[rd] = dtrace_load8(regs[r1]);
4996         break;

```

```

4982
4983     case DIF_OP_RLDUH:
4984         if (!dtrace_canload(regs[r1], 2, mstate, vstate))
4985             if (!dtrace_cancole(regs[r1], 2, mstate, vstate)) {
4986                 *flags |= CPU_DTRACE_KPRIV;
4987                 *illval = regs[r1];
4988                 break;
4989             }
4990             /*FALLTHROUGH*/
4991     case DIF_OP_LDUH:
4992         regs[rd] = dtrace_load16(regs[r1]);
4993         break;
4994     case DIF_OP_RLDUW:
4995         if (!dtrace_canload(regs[r1], 4, mstate, vstate))
4996             if (!dtrace_cancole(regs[r1], 4, mstate, vstate)) {
4997                 *flags |= CPU_DTRACE_KPRIV;
4998                 *illval = regs[r1];
4999                 break;
5000             }
5001             /*FALLTHROUGH*/
5002     case DIF_OP_LDUW:
5003         regs[rd] = dtrace_load32(regs[r1]);
5004         break;
5005     case DIF_OP_RLDX:
5006         if (!dtrace_canload(regs[r1], 8, mstate, vstate))
5007             if (!dtrace_cancole(regs[r1], 8, mstate, vstate)) {
5008                 *flags |= CPU_DTRACE_KPRIV;
5009                 *illval = regs[r1];
5010                 break;
5011             }
5012             /*FALLTHROUGH*/
5013     case DIF_OP_LDX:
5014         regs[rd] = dtrace_load64(regs[r1]);
5015         break;
5016     case DIF_OP_ULDSB:
5017         regs[rd] = (int8_t)
5018             dtrace_fuword8((void *)(uintptr_t)regs[r1]);
5019         break;
5020     case DIF_OP_ULDSH:
5021         regs[rd] = (int16_t)
5022             dtrace_fuword16((void *)(uintptr_t)regs[r1]);
5023         break;
5024     case DIF_OP_ULDSW:
5025         regs[rd] = (int32_t)
5026             dtrace_fuword32((void *)(uintptr_t)regs[r1]);
5027         break;
5028     case DIF_OP_ULDUB:
5029         regs[rd] =
5030             dtrace_fuword8((void *)(uintptr_t)regs[r1]);
5031         break;
5032     case DIF_OP_ULDUH:
5033         regs[rd] =
5034             dtrace_fuword16((void *)(uintptr_t)regs[r1]);
5035         break;
5036     case DIF_OP_ULDUW:
5037         regs[rd] =
5038             dtrace_fuword32((void *)(uintptr_t)regs[r1]);
5039         break;
5040     case DIF_OP_ULDX:
5041         regs[rd] =
5042             dtrace_fuword64((void *)(uintptr_t)regs[r1]);
5043         break;
5044     case DIF_OP_RET:
5045         rval = regs[rd];
5046         pc = textlen;
5047         break;
5048     case DIF_OP_NOP:
5049

```

```

5050
5051         break;
5052     case DIF_OP_SETX:
5053         regs[rd] = inttab[DIF_INSTR_INTEGER(instr)];
5054         break;
5055     case DIF_OP_SETS:
5056         regs[rd] = (uint64_t)(uintptr_t)
5057             (strtab + DIF_INSTR_STRING(instr));
5058         break;
5059     case DIF_OP_SCMP:
5060         size_t sz = state->dts_options[DTRACEOPT_STRSIZE];
5061         uintptr_t s1 = regs[r1];
5062         uintptr_t s2 = regs[r2];
5063
5064         if (s1 != NULL &&
5065             !dtrace_strcanload(s1, sz, mstate, vstate))
5066             break;
5067         if (s2 != NULL &&
5068             !dtrace_strcanload(s2, sz, mstate, vstate))
5069             break;
5070         cc_r = dtrace_strncmp((char *)s1, (char *)s2, sz);
5071
5072         cc_n = cc_r < 0;
5073         cc_z = cc_r == 0;
5074         cc_v = cc_c = 0;
5075         break;
5076     case DIF_OP_LDGA:
5077         regs[rd] = dtrace_dif_variable(mstate, state,
5078             r1, regs[r2]);
5079         break;
5080     case DIF_OP_LDGS:
5081         id = DIF_INSTR_VAR(instr);
5082
5083         if (id >= DIF_VAR_OTHER_UBASE) {
5084             uintptr_t a;
5085
5086             id -= DIF_VAR_OTHER_UBASE;
5087             svar = vstate->dtvs_globals[id];
5088             ASSERT(svar != NULL);
5089             v = &svar->dtsv_var;
5090
5091             if (!(v->dtdv_type.dtdt_flags & DIF_TF_BYREF)) {
5092                 regs[rd] = svar->dtsv_data;
5093                 break;
5094             }
5095
5096             a = (uintptr_t)svar->dtsv_data;
5097
5098             if ((*(uint8_t *)a == UINT8_MAX) {
5099                 /*
5100                  * If the 0th.byte is set to UINT8_MAX
5101                  * then this is to be treated as a
5102                  * reference to a NULL variable.
5103                  */
5104                 regs[rd] = NULL;
5105             } else {
5106                 regs[rd] = a + sizeof (uint64_t);
5107             }
5108
5109             break;
5110         }
5111
5112         regs[rd] = dtrace_dif_variable(mstate, state, id, 0);
5113         break;

```

```

5102     case DIF_OP_STGS:
5103         id = DIF_INSTR_VAR(instr);
5104
5105         ASSERT(id >= DIF_VAR_OTHER_UBASE);
5106         id -= DIF_VAR_OTHER_UBASE;
5107
5108         svar = vstate->dtvs_globals[id];
5109         ASSERT(svar != NULL);
5110         v = &svar->dtsv_var;
5111
5112         if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5113             uintptr_t a = (uintptr_t)svar->dtsv_data;
5114
5115             ASSERT(a != NULL);
5116             ASSERT(svar->dtsv_size != 0);
5117
5118             if (regs[rd] == NULL) {
5119                 *(uint8_t *)a = UINT8_MAX;
5120                 break;
5121             } else {
5122                 *(uint8_t *)a = 0;
5123                 a += sizeof(uint64_t);
5124             }
5125             if (!dtrace_vcanload(
5126                 (void *)(uintptr_t)regs[rd], &v->dtdv_type,
5127                 mstate, vstate))
5128                 break;
5129
5130             dtrace_vcopy((void *)(uintptr_t)regs[rd],
5131                         (void *)a, &v->dtdv_type);
5132             break;
5133         }
5134
5135         svar->dtsv_data = regs[rd];
5136         break;
5137
5138     case DIF_OP_LDTA:
5139         /*
5140          * There are no DTrace built-in thread-local arrays at
5141          * present. This opcode is saved for future work.
5142          */
5143         *flags |= CPU_DTRACE_ILLOP;
5144         regs[rd] = 0;
5145         break;
5146
5147     case DIF_OP_LDLS:
5148         id = DIF_INSTR_VAR(instr);
5149
5150         if (id < DIF_VAR_OTHER_UBASE) {
5151             /*
5152              * For now, this has no meaning.
5153              */
5154             regs[rd] = 0;
5155             break;
5156         }
5157
5158         id -= DIF_VAR_OTHER_UBASE;
5159
5160         ASSERT(id < vstate->dtvs_nlocals);
5161         ASSERT(vstate->dtvs_locals != NULL);
5162
5163         svar = vstate->dtvs_locals[id];
5164         ASSERT(svar != NULL);
5165         v = &svar->dtsv_var;
5166
5167         if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {

```

```

5168             uintptr_t a = (uintptr_t)svar->dtsv_data;
5169             size_t sz = v->dtdv_type.dtdt_size;
5170
5171             sz += sizeof(uint64_t);
5172             ASSERT(svar->dtsv_size == NCPU * sz);
5173             a += CPU->cpu_id * sz;
5174
5175             if (*(uint8_t *)a == UINT8_MAX) {
5176                 /*
5177                  * If the 0th byte is set to UINT8_MAX
5178                  * then this is to be treated as a
5179                  * reference to a NULL variable.
5180                  */
5181             } else {
5182                 regs[rd] = NULL;
5183             }
5184             regs[rd] = a + sizeof(uint64_t);
5185             break;
5186         }
5187
5188         ASSERT(svar->dtsv_size == NCPU * sizeof(uint64_t));
5189         tmp = (uint64_t *) (uintptr_t)svar->dtsv_data;
5190         regs[rd] = tmp[CPU->cpu_id];
5191         break;
5192
5193
5194     case DIF_OP_STLS:
5195         id = DIF_INSTR_VAR(instr);
5196
5197         ASSERT(id >= DIF_VAR_OTHER_UBASE);
5198         id -= DIF_VAR_OTHER_UBASE;
5199         ASSERT(id < vstate->dtvs_nlocals);
5200
5201         ASSERT(vstate->dtvs_locals != NULL);
5202         svar = vstate->dtvs_locals[id];
5203         ASSERT(svar != NULL);
5204         v = &svar->dtsv_var;
5205
5206         if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5207             uintptr_t a = (uintptr_t)svar->dtsv_data;
5208             size_t sz = v->dtdv_type.dtdt_size;
5209
5210             sz += sizeof(uint64_t);
5211             ASSERT(svar->dtsv_size == NCPU * sz);
5212             a += CPU->cpu_id * sz;
5213
5214             if (regs[rd] == NULL) {
5215                 *(uint8_t *)a = UINT8_MAX;
5216                 break;
5217             } else {
5218                 *(uint8_t *)a = 0;
5219                 a += sizeof(uint64_t);
5220             }
5221
5222             if (!dtrace_vcanload(
5223                 (void *)(uintptr_t)regs[rd], &v->dtdv_type,
5224                 mstate, vstate))
5225                 break;
5226
5227             dtrace_vcopy((void *)(uintptr_t)regs[rd],
5228                         (void *)a, &v->dtdv_type);
5229             break;
5230
5231         }
5232
5233         ASSERT(svar->dtsv_size == NCPU * sizeof(uint64_t));
5234         tmp = (uint64_t *) (uintptr_t)svar->dtsv_data;

```

```

5234         tmp[CPU->cpu_id] = regs[rd];
5235         break;
5236
5237     case DIF_OP_LDTS: {
5238         dtrace_dynvar_t *dvar;
5239         dtrace_key_t *key;
5240
5241         id = DIF_INSTR_VAR(instr);
5242         ASSERT(id >= DIF_VAR_OTHER_UBASE);
5243         id -= DIF_VAR_OTHER_UBASE;
5244         v = &vstate->dtvs_tlocals[id];
5245
5246         key = &tupregs[DIF_DTR_NREGS];
5247         key[0].dttk_value = (uint64_t)id;
5248         key[0].dttk_size = 0;
5249         DTRACE_TLS_THRKEY(key[1].dttk_value);
5250         key[1].dttk_size = 0;
5251
5252         dvar = dtrace_dynvar(dstate, 2, key,
5253             sizeof(uint64_t), DTRACE_DYNVAR_NOALLOC,
5254             mstate, vstate);
5255
5256         if (dvar == NULL) {
5257             regs[rd] = 0;
5258             break;
5259         }
5260
5261         if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5262             regs[rd] = (uint64_t)(uintptr_t)dvar->dtdv_data;
5263         } else {
5264             regs[rd] = *((uint64_t *)dvar->dtdv_data);
5265         }
5266
5267         break;
5268     }
5269
5270     case DIF_OP_STTS: {
5271         dtrace_dynvar_t *dvar;
5272         dtrace_key_t *key;
5273
5274         id = DIF_INSTR_VAR(instr);
5275         ASSERT(id >= DIF_VAR_OTHER_UBASE);
5276         id -= DIF_VAR_OTHER_UBASE;
5277
5278         key = &tupregs[DIF_DTR_NREGS];
5279         key[0].dttk_value = (uint64_t)id;
5280         key[0].dttk_size = 0;
5281         DTRACE_TLS_THRKEY(key[1].dttk_value);
5282         key[1].dttk_size = 0;
5283         v = &vstate->dtvs_tlocals[id];
5284
5285         dvar = dtrace_dynvar(dstate, 2, key,
5286             v->dtdv_type.dtdt_size > sizeof(uint64_t) ?
5287             v->dtdv_type.dtdt_size : sizeof(uint64_t),
5288             regs[rd] ? DTRACE_DYNVAR_ALLOC :
5289             DTRACE_DYNVAR_DEALLOC, mstate, vstate);
5290
5291         /*
5292          * Given that we're storing to thread-local data,
5293          * we need to flush our predicate cache.
5294         */
5295         curthread->t_predcache = NULL;
5296
5297         if (dvar == NULL)
5298             break;

```

```

5300         if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5301             if (!dtrace_vcanload(
5302                 (void *)(uintptr_t)regs[rd],
5303                 &v->dtdv_type, mstate, vstate))
5304                 break;
5305
5306             dtrace_vcopy((void *)(uintptr_t)regs[rd],
5307                         dvar->dtdv_data, &v->dtdv_type);
5308         } else {
5309             *((uint64_t *)dvar->dtdv_data) = regs[rd];
5310         }
5311
5312         break;
5313     }
5314
5315     case DIF_OP_SRA:
5316         regs[rd] = (int64_t)regs[r1] >> regs[r2];
5317         break;
5318
5319     case DIF_OP_CALL:
5320         dtrace_dif_subr(DIF_INSTR_SUBR(instr), rd,
5321                         regs, tupregs, ttop, mstate, state);
5322         break;
5323
5324     case DIF_OP_PUSHTR:
5325         if (ttop == DIF_DTR_NREGS) {
5326             *flags |= CPU_DTRACE_TUPOFLOW;
5327             break;
5328         }
5329
5330         if (r1 == DIF_TYPE_STRING) {
5331             /*
5332              * If this is a string type and the size is 0,
5333              * we'll use the system-wide default string
5334              * size. Note that we are _not_ looking at
5335              * the value of the DTRACEOPT_STRSIZE option;
5336              * had this been set, we would expect to have
5337              * a non-zero size value in the "pushtr".
5338             */
5339             tupregs[ttop].dttk_size =
5340                 dtrace_strlen((char *)(uintptr_t)regs[rd],
5341                               regs[r2] ? regs[r2] :
5342                               dtrace_strsize_default) + 1;
5343         } else {
5344             tupregs[ttop].dttk_size = regs[r2];
5345         }
5346
5347         tupregs[ttop++].dttk_value = regs[rd];
5348         break;
5349
5350     case DIF_OP_PUSHTV:
5351         if (ttop == DIF_DTR_NREGS) {
5352             *flags |= CPU_DTRACE_TUPOFLOW;
5353             break;
5354         }
5355
5356         tupregs[ttop].dttk_value = regs[rd];
5357         tupregs[ttop++].dttk_size = 0;
5358         break;
5359
5360     case DIF_OP_POPTS:
5361         if (ttop != 0)
5362             ttop--;
5363         break;
5364
5365     case DIF_OP_FLUSHTS:

```

```

5366         ttop = 0;
5367         break;

5369     case DIF_OP_LDCAA:
5370     case DIF_OP_LDTAA: {
5371         dtrace_dynvar_t *dvar;
5372         dtrace_key_t *key = tupregs;
5373         uint_t nkeys = ttop;

5375         id = DIF_INSTR_VAR(instr);
5376         ASSERT(id >= DIF_VAR_OTHER_UBASE);
5377         id -= DIF_VAR_OTHER_UBASE;

5379         key[nkeys].dttk_value = (uint64_t)id;
5380         key[nkeys++].dttk_size = 0;

5382         if (DIF_INSTR_OP(instr) == DIF_OP_LDTAA) {
5383             DTRACE_TLS_THRKEY(key[nkeys].dttk_value);
5384             key[nkeys++].dttk_size = 0;
5385             v = &vstate->dtvs_tlocals[id];
5386         } else {
5387             v = &vstate->dtvs_globals[id]->dtsv_var;
5388         }

5390         dvar = dtrace_dynvar(dstate, nkeys, key,
5391             v->dtdv_type.dtdt_size > sizeof (uint64_t) ?
5392             v->dtdv_type.dtdt_size : sizeof (uint64_t),
5393             DTRACE_DYNVAR_NOALLOC, mstate, vstate);

5395         if (dvar == NULL) {
5396             regs[rd] = 0;
5397             break;
5398         }

5399         if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5400             regs[rd] = (uint64_t)(uintptr_t)dvar->dtdv_data;
5401         } else {
5402             regs[rd] = *((uint64_t *)dvar->dtdv_data);
5403         }
5404     }

5405         break;
5406     }

5407 }

5409 case DIF_OP_STGAA:
5410 case DIF_OP_STTAA: {
5411     dtrace_dynvar_t *dvar;
5412     dtrace_key_t *key = tupregs;
5413     uint_t nkeys = ttop;

5415     id = DIF_INSTR_VAR(instr);
5416     ASSERT(id >= DIF_VAR_OTHER_UBASE);
5417     id -= DIF_VAR_OTHER_UBASE;

5419     key[nkeys].dttk_value = (uint64_t)id;
5420     key[nkeys++].dttk_size = 0;

5422     if (DIF_INSTR_OP(instr) == DIF_OP_STTAA) {
5423         DTRACE_TLS_THRKEY(key[nkeys].dttk_value);
5424         key[nkeys++].dttk_size = 0;
5425         v = &vstate->dtvs_tlocals[id];
5426     } else {
5427         v = &vstate->dtvs_globals[id]->dtsv_var;
5428     }

5430     dvar = dtrace_dynvar(dstate, nkeys, key,
5431             v->dtdv_type.dtdt_size > sizeof (uint64_t) ?

```

```

5432         v->dtdv_type.dtdt_size : sizeof (uint64_t),
5433         regs[rd] ? DTRACE_DYNVAR_ALLOC :
5434             DTRACE_DYNVAR DEALLOC, mstate, vstate);

5436         if (dvar == NULL)
5437             break;

5439         if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5440             if (!dtrace_vcanload(
5441                 (void *) (uintptr_t)regs[rd], &v->dtdv_type,
5442                 mstate, vstate))
5443                 break;

5445             dtrace_vcopy((void *) (uintptr_t)regs[rd],
5446                         dvar->dtdv_data, &v->dtdv_type);
5447         } else {
5448             *((uint64_t *) dvar->dtdv_data) = regs[rd];
5449         }
5450     }

5451         break;
5452     }

5454 case DIF_OP_ALLOCS: {
5455     uintptr_t ptr = P2ROUNDUP(mstate->dtms_scratch_ptr, 8);
5456     size_t size = ptr - mstate->dtms_scratch_ptr + regs[r1];

5458     /*
5459      * Rounding up the user allocation size could have
5460      * overflowed large, bogus allocations (like -1ULL) to
5461      * 0.
5462      */
5463     if (size < regs[r1] || !DTRACE_INSCRATCH(mstate, size)) {
5464         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
5465         regs[rd] = NULL;
5466         break;
5467     }

5468     dtrace_bzero((void *) mstate->dtms_scratch_ptr, size);
5469     mstate->dtms_scratch_ptr += size;
5470     regs[rd] = ptr;
5471     break;
5472 }

5473 }

5474 }

5476 case DIF_OP_COPIYS: {
5477     if (!dtrace_canstore(regs[rd], regs[r2],
5478                         mstate, vstate)) {
5479         *flags |= CPU_DTRACE_BADADDR;
5480         *illval = regs[rd];
5481         break;
5482     }

5484     if (!dtrace_canload(regs[r1], regs[r2], mstate, vstate))
5485         break;

5487     dtrace_bcopy((void *) (uintptr_t)regs[r1],
5488                 (void *) (uintptr_t)regs[rd], (size_t)regs[r2]);
5489     break;
5490 }

5491 case DIF_OP_STB: {
5492     if (!dtrace_canstore(regs[rd], 1, mstate, vstate)) {
5493         *flags |= CPU_DTRACE_BADADDR;
5494         *illval = regs[rd];
5495         break;
5496     }

5497     *((uint8_t *) (uintptr_t)regs[rd]) = (uint8_t)regs[r1];
5498 }
```

```
5498     break;
5499
5500     case DIF_OP_STH:
5501         if (!dtrace_canstore(regs[rd], 2, mstate, vstate)) {
5502             *flags |= CPU_DTRACE_BADADDR;
5503             *illval = regs[rd];
5504             break;
5505         }
5506         if (regs[rd] & 1) {
5507             *flags |= CPU_DTRACE_BADALIGN;
5508             *illval = regs[rd];
5509             break;
5510         }
5511         *((uint16_t *) (uintptr_t)regs[rd]) = (uint16_t)regs[r1];
5512         break;
5513
5514     case DIF_OP_STW:
5515         if (!dtrace_canstore(regs[rd], 4, mstate, vstate)) {
5516             *flags |= CPU_DTRACE_BADADDR;
5517             *illval = regs[rd];
5518             break;
5519         }
5520         if (regs[rd] & 3) {
5521             *flags |= CPU_DTRACE_BADALIGN;
5522             *illval = regs[rd];
5523             break;
5524         }
5525         *((uint32_t *) (uintptr_t)regs[rd]) = (uint32_t)regs[r1];
5526         break;
5527
5528     case DIF_OP_STX:
5529         if (!dtrace_canstore(regs[rd], 8, mstate, vstate)) {
5530             *flags |= CPU_DTRACE_BADADDR;
5531             *illval = regs[rd];
5532             break;
5533         }
5534         if (regs[rd] & 7) {
5535             *flags |= CPU_DTRACE_BADALIGN;
5536             *illval = regs[rd];
5537             break;
5538         }
5539         *((uint64_t *) (uintptr_t)regs[rd]) = regs[r1];
5540         break;
5541     }
5542 }
5543
5544 if (!(*flags & CPU_DTRACE_FAULT))
5545     return (rval);
5546
5547 mstate->dtms_fltoffs = opc * sizeof (dif_instr_t);
5548 mstate->dtms_present |= DTRACE_MSTATE_FLTOFFS;
5549
5550 return (0);
5551 }
```

unchanged portion omitted