/\*

\* kutrace\_mod.c

\*

\* Author: Richard Sites <dick.sites@gmail.com>

\*

\* This program is free software: you can redistribute it and/or modify

\* it under the terms of the GNU General Public License version 2 as

\* published by the Free Software Foundation.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program. If not, see <http://www.gnu.org/licenses/>.

\*

\* Signed-off-by: Richard Sites <dick.sites@gmail.com>

\*/

/\*

\* A module that implements kernel/user tracing

\* dsites 2023.02.18

\*

\* See include/linux/kutrace.h for struct definitions

\*

\* Most patches will be something like

\* kutrace1(event, arg) which calls trace\_1 here

\*

\*/

/\*

\* kutrace.c -- kernel/user tracing implementation

\* dsites 2019.02.14 Reworked for the 4.19 kernel, from dclab\_trace.c

\* dsites 2020.02.04 fixed getclaim(n) bug for n > 1

\* dsites 2020.10.30 Add packet trace parameters

\* use something like

\* sudo insmod kutrace\_mod.ko tracemb=20 pktmask=0x0000000f pktmatch=0xd1c517e5

\* default is the above

\* pktmask=0 traces nothing, pktmask=-1 traces all (no pktmatch needed)

\* dsites 2021.09.25 Add Rpi-4B 64-bit support

\* dsites 2023.02.13 Add fast 4KB trace buffer extraction

\* dsites 2023.02.13 Change module version number to 4

\* dsites 2023.02.16 Merge in TSDELTA code from FreeBSD version

\*

\*/

#include <linux/kutrace.h>

#include <linux/capability.h>

#include <linux/cpufreq.h>

#include <linux/delay.h>

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/percpu.h>

#include <linux/sched.h>

#include <linux/spinlock.h>

#include <linux/stat.h>

#include <linux/string.h>

#include <linux/types.h> /\* u64, among others \*/

#include <linux/uaccess.h>

#include <linux/vmalloc.h>

#include <asm/atomic.h>

#include <asm/uaccess.h>

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Richard L Sites");

// Added 2023.02.13. Move these into the kernel at next build

#ifndef KUTRACE\_CMD\_SET4KB

#define KUTRACE\_CMD\_SET4KB 12

#endif

#ifndef KUTRACE\_CMD\_GET4KB

#define KUTRACE\_CMD\_GET4KB 13

#endif

#ifndef KUTRACE\_CMD\_GETIPC4KB

#define KUTRACE\_CMD\_GETIPC4KB 14

#endif

#ifndef KUTRACE\_TSDELTA

#define KUTRACE\_TSDELTA 0x21D /\* Delta to advance timestamp \*/

#endif

#**define TEST\_LLC\_MISS 1**

// GCC compiler options to distinguish build targets

// Use to get these:

// gcc -dM -E -march=native - < /dev/null

/\* Add others as you find and test them \*/

#if defined(\_\_x86\_64\_\_)

#define Isx86\_64 1

#else

#define Isx86\_64 0

#endif

#if defined(\_\_znver1)

#define IsAmd\_64 Isx86\_64

#define IsIntel\_64 0

#else

#define IsAmd\_64 0

#define IsIntel\_64 Isx86\_64

#endif

#if defined(\_\_aarch64\_\_)

#define IsArm\_64 1

#else

#define IsArm\_64 0

#endif

/\* AMD-specific defines \*/

/\*--------------------------------\*/

/\* From Open-Source Register Reference For AMD Family 17h Processors \*/

/\* Models 00h-2Fh \*/

/\* rdtsc counts cycles, no setup needed \*/

/\* IRPerfCount counts instructions retired, once set up \*/

#define IRPerfCount 0xC00000E9

#define RYZEN\_HWCR 0xC0010015

#define IRPerfEn (1L << 30)

/\* PStateStat<2:0> gives current P-state of a core \*/

/\* PStateDefn<13:8> Did gives frequency divisor in increments of 1/8 \*/

/\* PStateDefn<7:0> Fid gives frequency in increments of 25 \*/

/\* I think all this boils down to freq = Fid \* 200 / Did, but it could be 266.67 \*/

#define PStateStat 0xC0010063

#define PStateDef0 0xC0010064

#define PStateDef1 0xC0010065

#define PStateDef2 0xC0010066

#define PStateDef3 0xC0010067

#define PStateDef4 0xC0010068

#define PStateDef5 0xC0010069

#define PStateDef6 0xC001006A

#define PStateDef7 0xC001006B

#define PStat\_MASK 0x07LU

#define CpuDid\_SHIFT 8

#define CpuDid\_MASK 0x3FLU

#define CpuFid\_SHIFT 0

#define CpuFid\_MASK 0xFFLU

// amd notes

// FIDVID\_STATUS HwPstate

// MSRC001\_006[4...B] [P-state [7:0]] (Core::X86::Msr::PStateDef)

// freq = <7:0> \* 25 MHz \* CpuDid in <13:8> VCO

// From https://developer.amd.com/wp-content/resources/56255\_3\_03.PDF

// sudo watch -n 1 cpupower monitor

/\* Intel-specific defines \*/

/\*--------------------------------\*/

/\* From Intel¬Æ 64 and IA-32 Architectures Software Developer‚Äôs Manual \*/

/\* Volume 4: Model-Specific Registers \*/

/\* rdtsc counts cycles, no setup needed \*/

/\* IA32\_FIXED\_CTR0 counts instructions retired, once set up \*/

#define IA32\_FIXED\_CTR0 0x309

#define IA32\_FIXED\_CTR\_CTRL 0x38D

#define EN0\_OS (1L << 0)

#define EN0\_Usr (1L << 1)

#define EN0\_Anythread (1L << 2)

#define EN0\_PMI (1L << 3)

#define EN0\_ALL (EN0\_OS | EN0\_Usr | EN0\_Anythread | EN0\_PMI)

#define IA32\_PERF\_GLOBAL\_CTRL 0x38F

#define EN\_FIXED\_CTR0 (1L << 32)

/\* Constants for reading LLC\_MISS \*/

**#define IA32\_PERFEVTSEL0 0x186**

**#define IA32\_PERFEVTSEL1 0x187**

**#define IA32\_PERFEVTSEL2 0x188**

**#define IA32\_PERFEVTSEL3 0x189**

**#define PMC\_USR\_EN (1 << 16)**

**#define PMC\_OS\_EN (1 << 17)**

**#define PMC\_EDGE (1 << 18)**

**#define PMC\_PINCTRL (1 << 19)**

**#define PMC\_INT\_EN (1 << 20)**

**#define PMC\_EN (1 << 22)**

**#define PMC\_INV (1 << 23)**

**#define PMC\_BITS (0xFF << 16)**

**#define C\_LLC\_MISS (0x00 << 24)**

**#define U\_LLC\_MISS (0x41 << 8)**

**#define E\_LLC\_MISS (0x2E << 0)**

**#define C\_INST\_RET (0x00 << 24)**

**#define U\_INST\_RET (0x00 << 8)**

**#define E\_INST\_RET (0xC0 << 0)**

**#define IA32\_PMC0 0x0C1**

**#define IA32\_PMC1 0x0C2**

/\* #define IA32\_PERF\_GLOBAL\_CTRL 0x38F \*/

#define PMC0\_EN (1L << 0)

#define PMC1\_EN (1L << 1)

/\* MSR\_IA32\_PERF\_STATUS<15:8> gives current CPU frequency in increments of 100 MHz \*/

#define MSR\_PERF\_STATUS 0x198

#define FID\_SHIFT 8

#define FID\_MASK 0xFFL

/\* Arm-speficic defines \*/

/\*--------------------------------\*/

/\* Old 32-bit defines in linux-4.19.19/arch/arm/kernel/perf\_event\_v6.c \*/

#if IsArm\_64

/\* This is for 64-bit ARM \*/

typedef long long int int64;

typedef long long unsigned int uint64;

#define FLX "%016llx"

#define FLD "%lld"

#define FUINTPTRX "%016lx"

#define CL(x) x##LL

#define CLU(x) x##LLU

#define ATOMIC\_READ atomic64\_read

#define ATOMIC\_SET atomic64\_set

#define ATOMIC\_ADD\_RETURN atomic64\_add\_return

#elif Isx86\_64

/\* This is for 64-bit X86 \*/

typedef long int int64;

typedef long unsigned int uint64;

#define FLX "%016lx"

#define FLD "%ld"

#define FUINTPTRX "%016lx"

#define CL(x) x##L

#define CLU(x) x##LU

#define ATOMIC\_READ atomic64\_read

#define ATOMIC\_SET atomic64\_set

#define ATOMIC\_ADD\_RETURN atomic64\_add\_return

#else

#error Need type defines for your architecture

#endif

#if IsAmd\_64

#define BCLK\_FREQ 200LU /\* CPU Ryzen base clock, assume 25 MHz \* 8 \*/

#elif IsIntel\_64

#define BCLK\_FREQ 100LU /\* CPU Intel base clock, assume 100 MHz \*/

#else

#define BCLK\_FREQ 0LU /\* CPU RPi, frequency sampling not implemented -- change notifications used \*/

#endif

/\* Forward declarations \*/

static u64 kutrace\_control(u64 command, u64 arg);

static int \_\_init kutrace\_mod\_init(void);

/\* For the flags byte in traceblock[1] \*/

#define IPC\_Flag CLU(0x80)

#define WRAP\_Flag CLU(0x40)

#define LLC\_Flag CLU(0x20)

/\* Incoming arg to do\_reset \*/

#define DO\_IPC 1

#define DO\_WRAP 2

**#define DO\_LLC 4**

/\* Module parameter: default how many MB of kernel trace memory to reserve \*/

/\* This is for the standalone, non-module version \*/

/\* static const long int kTraceMB = 32; \*/

/\* Version number of this kernel tracing code \*/

/\* 2023.02.13 Incremented to 4 for fast 4KB trace buffer extraction \*/

static const u64 kModuleVersionNumber = 4;

/\* A few global variables \*/

/\* IPC Instructions per cycle flag \*/

static bool do\_ipc; /\* Initially false \*/

/\* Wraparound tracing vs. stop when buffer is full \*/

static bool do\_wrap; /\* Initially false \*/

/\* Current offset to use for fast 4KB trace buffer extraction get4kb and getipc4kb \*/

/\* Set by KUTRACE\_CMD\_SET4KB call \*/

static u64 get4kb\_subscr; /\* Initially zero \*/

/\* Module parameter: default how many MB of kernel trace memory to reserve \*/

static long int tracemb = 2;

static long int check = 1;

/\* Module parameters: packet filtering. Initially match just dclab RPC markers \*/

static long int pktmask = 0x0000000f;

static long int pktmatch = 0xd1c517e5;

module\_param(tracemb, long, S\_IRUSR);

MODULE\_PARM\_DESC(tracemb, "MB of kernel trace memory to reserve (2)");

module\_param(check, long, S\_IRUSR);

MODULE\_PARM\_DESC(check, "0: no checking, 1: require PTRACE capability for DoControl (1)");

module\_param(pktmask, long, S\_IRUSR);

MODULE\_PARM\_DESC(pktmask, "Bit-per-byte of which bytes to use in hash");

module\_param(pktmatch, long, S\_IRUSR);

MODULE\_PARM\_DESC(pktmatch, "Matching hash value");

/\* These four are exported by our patched kernel.

\* See linux-4.19.19/kernel/kutrace/kutrace.c

\*/

extern bool kutrace\_tracing;

extern struct kutrace\_ops kutrace\_global\_ops;

extern u64\* kutrace\_pid\_filter;

extern struct kutrace\_nf kutrace\_net\_filter;

DECLARE\_PER\_CPU(struct kutrace\_traceblock, kutrace\_traceblock\_per\_cpu);

/\*

\* Individual trace entries are at least one u64, with this format:

\*

\* +-------------------+-----------+-------+-------+-------+-------+

\* | timestamp | event | delta | retval| arg0 |

\* +-------------------+-----------+-------+-------+-------+-------+

\* 20 12 8 8 16

\*

\* timestamp: low 20 bits of some free-running time counter in the

\* 10-40 MHz range. For ARM, this is the 32 MHz cntvct\_el0.

\* event: traced event number, syscall N, sysreturn N, etc.

\* See user-mode kutrace\_lib.h for the full set.

\* matching call and return events differ just in one event bit.

\* delta: for optimized call-return, return timestamp - call timestamp,

\* else zero.

\* retval: for optimized call-return, the low 8 bits of the return value,

\* else zero.

\* arg0: for syscall, the low 16 bits of the first argument to the syscall,

\* else zero.

\*

\* Multi-u64 entries have a count 1-8 in the middle 4 bits of event.

\* These events are all in the range 0x000 to 0x1ff with the middle

\* four bits non-zero.

\*

\* The first word of each 64KB block has this format:

\* +-------+-------------------------------------------------------+

\* | cpu# | full timestamp |

\* +-------+-------------------------------------------------------+

\* 56 0

\*

\* The second word of each 64KB block has this format:

\* +-------+-------------------------------------------------------+

\* | flags | gettimeofday() value to be filled in by user code |

\* +-------+-------------------------------------------------------+

\* 56 0

\*

\*/

#define ARG\_MASK CLU(0x00000000ffffffff)

#define ARG0\_MASK CLU(0x000000000000ffff)

#define RETVAL\_MASK CLU(0x0000000000ff0000)

#define DELTA\_MASK CLU(0x00000000ff000000)

#define EVENT\_MASK CLU(0x00000fff00000000)

#define TIMESTAMP\_MASK CLU(0xfffff00000000000)

#define EVENT\_DELTA\_RETVAL\_MASK (EVENT\_MASK | DELTA\_MASK | RETVAL\_MASK)

#define EVENT\_RETURN\_BIT CLU(0x0000020000000000)

#define EVENT\_LENGTH\_FIELD\_MASK CLU(0x000000000000000f)

#define UNSHIFTED\_RETVAL\_MASK CLU(0x00000000000000ff)

#define UNSHIFTED\_DELTA\_MASK CLU(0x00000000000000ff)

#define UNSHIFTED\_EVENT\_MASK CLU(0x0000000000000fff)

#define UNSHIFTED\_TIMESTAMP\_MASK CLU(0x00000000000fffff)

#define UNSHIFTED\_EVENT\_RETURN\_BIT CLU(0x0000000000000200)

#define UNSHIFTED\_EVENT\_HAS\_RETURN\_MASK CLU(0x0000000000000c00)

#define MIN\_EVENT\_WITH\_LENGTH CLU(0x010)

#define MAX\_EVENT\_WITH\_LENGTH CLU(0x1ff)

#define MAX\_DELTA\_VALUE 255

#define MAX\_PIDNAME\_LENGTH 16

#define RETVAL\_SHIFT 16

#define DELTA\_SHIFT 24

#define EVENT\_SHIFT 32

#define TIMESTAMP\_SHIFT 44

#define EVENT\_LENGTH\_FIELD\_SHIFT 4

#define FULL\_TIMESTAMP\_MASK CLU(0x00ffffffffffffff)

#define CPU\_NUMBER\_SHIFT 56

#define GETTIMEOFDAY\_MASK CLU(0x00ffffffffffffff)

#define FLAGS\_SHIFT 56

/\* For deciding that large timestamp advance is really a late store \*/

/\* with backward time. \*/

static const u64 kLateStoreThresh = 0x00000000000e0000LLU;

/\*

\* Trace memory is consumed backward, high to low

\* This allows valid test for full block even if an interrupt routine

\* switches to a new block mid-test. The condition tracebase == NULL

\* means that initialization needs to be called.

\*

\* Per-CPU trace blocks are 64KB, contining 8K u64 items. A trace entry is

\* 1-8 items. Trace entries do not cross block boundaries.

\*

\*/

char \*tracebase; /\* Initially NULL address of kernel trace memory \*/

u64 \*traceblock\_high; /\* just off high end of trace memory \*/

u64 \*traceblock\_limit; /\* at low end of trace memory \*/

u64 \*traceblock\_next; /\* starts at high, moves down to limit \*/

bool did\_wrap\_around;

/\*

**\* Trace memory layout without IPC/LLC tracing.**

\* tracebase

\* traceblock\_limit traceblock\_next traceblock\_high

\* | | |

\* v v v

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* | / / / / / / / / / / / / / / / | |

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* <==== allocated blocks grow down

\*

\*

**\* Trace memory layout with IPC/LLC tracing. IPC/LLC bytes go into lower 1/8.**

\* tracebase

\* | traceblock\_limit traceblock\_next traceblock\_high

\* | | | |

\* v v v v

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* |////| | / / / / / / / / / / / | |

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* <== <==== allocated blocks grow down

\* IPC/LLC bytes

\*/

DEFINE\_RAW\_SPINLOCK(kutrace\_lock);

/\* Trace block size in bytes = 64KB \*/

#define KUTRACEBLOCKSHIFT (16)

#define KUTRACEBLOCKSIZE (1 << KUTRACEBLOCKSHIFT)

/\* Trace block size in u64 words \*/

#define KUTRACEBLOCKSHIFTU64 (KUTRACEBLOCKSHIFT - 3)

#define KUTRACEBLOCKSIZEU64 (1 << KUTRACEBLOCKSHIFTU64)

**/\* IPC/LLC block size in u8 bytes \*/**

#define KUIPCBLOCKSHIFTU8 (KUTRACEBLOCKSHIFTU64 - 3)

#define KUIPCBLOCKSIZEU8 (1 << KUIPCBLOCKSHIFTU8)

/\* IPC design \*/

/\* Map IPC \* 8 [0.0 .. 3.75] into sorta-log value \*/

static const u64 kIpcMapping[64] = {

0,1,2,3, 4,5,6,7, 8,8,9,9, 10,10,11,11,

12,12,12,12, 13,13,13,13, 14,14,14,14, 15,15,15,15,

15,15,15,15, 15,15,15,15, 15,15,15,15, 15,15,15,15,

15,15,15,15, 15,15,15,15, 15,15,15,15, 15,15,15,15

};

/\* Map IPC= inst\_retired / cycles to sorta-log four bits \*/

/\* NOTE: delta\_cycles is in increments of cycles/64. The arithmetic \*/

/\* below compensates for this. \*/

/\* 0, 1/8, 1/4, 3/8, 1/2, 5/8, 3/4, 7/8, 1, 5/4, 3/2, 7/4, 2, 5/2, 3, 7/2 \*/

inline u64 get\_granular(u64 delta\_inst, u64 delta\_cycles) {

u32 del\_inst, del\_cycles, ipc;

if ((delta\_cycles & ~1) == 0) return 0; /\* Too small to matter; avoid zdiv \*/

/\* Do 32-bit divide to save ~10 CPU cycles vs. 64-bit \*/

/\* With ~20ms guaranteed max interval, no overflow problems \*/

del\_inst = (u32)delta\_inst;

#if IsArm\_64

/\* "cycle" counter is 24MHz, cycles are 2400 MHz, so one count = 2400/24 = 100 cycles \*/

/\* Call it 96 (less than 1% error). To get 8\*inst/cycles for the divide below, we mul by 8/96 = 1/12 \*/

del\_cycles = (u32)(delta\_cycles \* 12); /\* cycles/96 to cycles/8 \*/

#else

del\_cycles = (u32)(delta\_cycles << 3); /\* cycles/64 to cycles/8 \*/

#endif

ipc = del\_inst / del\_cycles; /\* gives IPC\*8 \*/

return kIpcMapping[ipc & 0x3F]; /\* Truncate unexpected IPC >= 8.0 \*/

}

**/\***

**\* Convert delta LLC miss count into floor(log2(delta))-1 for 8 <= delta < 64K**

**\* i.e. 1..7=>1, 8..15=>2, 16..31=>3 ... 32K..64K-1=>14**

**\* Map delta=0 to 0, 64K<=delta to 15**

**\* Note that we map 2\*\*0, 2\*\*1, and 2\*\*2 all to the same result, using the**

**\* missing two code points instead to extend our range to 64K+ misses**

**\* (i.e. 4MB with 64-byte cache lines)**

**\*/**

**inline u64 get\_granular\_llc(u64 delta\_llc) {**

**u64 retval;**

**if (delta\_llc == 0) return 0; /\* common case, we hope \*/**

**if ((delta\_llc & ~CLU(0x7)) == 0) return 1; /\* 1..7 \*/**

**if ((delta\_llc & ~CLU(0xFFFF)) != 0) return 15; /\* 64K+ \*/**

**/\* 1 + floor(log2(delta)) \*/**

**retval = 0;**

**if ((delta\_llc & CLU(0xFF00)) != 0) {**

**delta\_llc >>= 8;**

**retval += 8;**

**}**

**if ((delta\_llc & CLU(0x00F0)) != 0) {**

**delta\_llc >>= 4;**

**retval += 4;**

**}**

**if ((delta\_llc & CLU(0x000C)) != 0) {**

**delta\_llc >>= 2;**

**retval += 2;**

**}**

**if ((delta\_llc & CLU(0x0002)) != 0) {**

**retval += 1;**

**}**

**return retval - 1;**

**}**

**/\* Machine-specific register Access utilities \*/**

**/\*----------------------------------------------------------------------------\*/**

**#if Isx86\_64**

**/\* RDMSR Read a 64-bit value from a MSR. \*/**

**/\* The A constraint stands for concatenation of registers EAX and EDX. \*/**

**static inline u64 rdMSR(u32 msr) {**

**u32 lo, hi;**

**asm volatile( "rdmsr" : "=a"(lo), "=d"(hi) : "c"(msr) );**

**return ((u64)lo) | (((u64)hi) << 32);**

**}**

**/\* WRMSR Write a 64-bit value to a MSR. \*/**

**/\* The A constraint stands for concatenation of registers EAX and EDX. \*/**

**static inline void wrMSR(u32 msr, u64 value)**

**{**

**u32 lo = value;**

**u32 hi = value >> 32;**

**asm volatile( "wrmsr" : : "a"(lo), "d"(hi), "c"(msr) );**

**}**

**#endif**

**/\* Set up global state for reading time, retired, freq \*/**

**/\*----------------------------------------------------------------------------\*/**

**/\* Set up global state for reading scaled CPU cycles \*/**

**/\* This needs to run once on each CPU core \*/**

**/\* For ARM, make sure it increments every 64 cycles, not 1 \*/**

**static void ku\_setup\_timecount(void)**

**{**

**#if IsArm\_64**

**/\* No setup needed for cntvct \*/**

**#elif Isx86\_64**

**/\* No setup needed for rdtsc \*/**

**#elif IsArm\_64**

**/\* No setup needed; count every 1 cycle for ccnt is the default \*/**

**#else**

**#endif**

**}**

**/\* Set up global state for reading instructions retired \*/**

**/\* This needs to run once on each CPU core \*/**

**static void ku\_setup\_inst\_retired(void)**

**{**

**#if IsAmd\_64**

**u64 inst\_ret\_enable;**

**/\* Enable fixed inst\_ret counter \*/**

**inst\_ret\_enable = rdMSR(RYZEN\_HWCR);**

**printk(KERN\_INFO " kutrace\_mod rdMSR(RYZEN\_HWCR) = %016llx\n", inst\_ret\_enable);**

**inst\_ret\_enable |= IRPerfEn;**

**wrMSR(RYZEN\_HWCR, inst\_ret\_enable);**

**#elif IsIntel\_64**

**u64 inst\_ret\_ctrl;**

**u64 inst\_ret\_enable;**

**/\* cpuCount\_HW\_INSTRUCTIONS = 1<<30 \*/**

**/\* Configure fixed inst\_ret counter in IA32\_FIXED\_CTR\_CTRL \*/**

**/\* count both kernel and user, count per-CPU-thread, no interrupt \*/**

**inst\_ret\_ctrl = rdMSR(IA32\_FIXED\_CTR\_CTRL);**

**printk(KERN\_INFO " kutrace\_mod rdMSR(IA32\_FIXED\_CTR\_CTRL) = %016llx\n", inst\_ret\_ctrl);**

**inst\_ret\_ctrl &= ~EN0\_ALL;**

**inst\_ret\_ctrl |= (EN0\_OS | EN0\_Usr);**

**wrMSR(IA32\_FIXED\_CTR\_CTRL, inst\_ret\_ctrl);**

**/\* Enable fixed inst\_ret counter in IA32\_PERF\_GLOBAL\_CTRL \*/**

**inst\_ret\_enable = rdMSR(IA32\_PERF\_GLOBAL\_CTRL);**

**printk(KERN\_INFO " kutrace\_mod rdMSR(IA32\_PERF\_GLOBAL\_CTRL) = %016llx\n", inst\_ret\_enable);**

**inst\_ret\_enable |= EN\_FIXED\_CTR0;**

**wrMSR(IA32\_PERF\_GLOBAL\_CTRL, inst\_ret\_enable);**

**#elif IsArm\_64**

**/\* Setup needed for instruction counting \*/**

**/\* set up pmevtyper0<15:0> to count INST\_RETIRED =0x08 \*/**

**/\* set up pmcntenset<0>=1 to enable \*/**

**u64 evtcount = 8; /\* INST\_RETIRED \*/**

**u64 r = 0;**

**asm volatile("mrs %x0, pmcr\_el0" : "=r" (r));**

**asm volatile("msr pmcr\_el0, %x0" : : "r" (r | 1)); /\* enable pmu \*/**

**asm volatile("msr pmevtyper2\_el0, %x0" : : "r" (evtcount)); /\* count inst\_retired \*/**

**asm volatile("mrs %x0, pmcntenset\_el0" : "=r" (r));**

**asm volatile("msr pmcntenset\_el0, %x0" : : "r" (r|1<<2)); /\* enable cntr[2] \*/**

**#else**

**#error Define ku\_setup\_inst\_retired for your architecture**

**#endif**

**}**

**/\* Set up global state for reading instructions retired \*/**

**/\* This needs to run once on each CPU core \*/**

**static void ku\_setup\_llc\_miss(void)**

**{**

**#if IsIntel\_64**

**u64 llc\_miss\_sel;**

**u64 llc\_miss\_enable;**

**/\* Count LLC\_MISS, both user and os; enable counting \*/**

**/\* llc\_miss\_sel = rdMSR(IA32\_PERFEVTSEL1); \*/**

**llc\_miss\_sel = (PMC\_USR\_EN | PMC\_OS\_EN | PMC\_EN) |**

**(C\_LLC\_MISS | U\_LLC\_MISS | E\_LLC\_MISS);**

**// (C\_INST\_RET | U\_INST\_RET | E\_INST\_RET);**

**wrMSR(IA32\_PERFEVTSEL1, llc\_miss\_sel);**

**/\* Enable fixed llc\_miss counter in IA32\_PERF\_GLOBAL\_CTRL \*/**

**llc\_miss\_enable = rdMSR(IA32\_PERF\_GLOBAL\_CTRL);**

**llc\_miss\_enable |= PMC1\_EN;**

**wrMSR(IA32\_PERF\_GLOBAL\_CTRL, llc\_miss\_enable);**

**#elif IsArm\_64**

**/\* Setup needed for instruction counting \*/**

**/\* set up pmevtyper0<15:0> to count LLC\_MISSES =0x33 \*/**

**/\* set up pmcntenset<0>=1 to enable \*/**

**u64 evtcount = 0x33; /\* INST\_RETIRED \*/**

**u64 r = 0;**

**asm volatile("mrs %x0, pmcr\_el0" : "=r" (r));**

**asm volatile("msr pmcr\_el0, %x0" : : "r" (r | 1)); /\* enable pmu \*/**

**asm volatile("msr pmevtyper2\_el0, %x0" : : "r" (evtcount)); /\* count LLC misses \*/**

**asm volatile("mrs %x0, pmcntenset\_el0" : "=r" (r));**

**asm volatile("msr pmcntenset\_el0, %x0" : : "r" (r|1<<2)); /\* enable cntr[2] \*/**

**#else**

**#error Define ku\_setup\_llc\_miss for your architecture**

**#endif**

**}**

/\* Set up global state for reading CPU frequency \*/

/\* This needs to run once on each CPU core \*/

static void ku\_setup\_cpu\_freq(void)

{

/\* No setup for AMD, Intel, RPi4 \*/

}

/\*x86-64 or Arm-specific time counter, ideally 30-60 MHz (16-32 nsec) \*/

/\* Arm64 RPi4 returns 54MHz counts, 18.52ns \*/

/\* x86-64 version returns constant rdtsc() >> 6 to give ~20ns resolution \*/

/\* Read a time counter \*/

/\* This is performance critical -- every trace entry \*/

/\* Ideally, this counts at a constant rate of 16-32 nsec per count. \*/

/\*----------------------------------------------------------------------------\*/

inline u64 ku\_get\_timecount(void)

{

u64 timer\_value;

#if IsArm\_64

asm volatile("mrs %x0, cntvct\_el0" : "=r"(timer\_value));

#elif Isx86\_64

/\* If you change this shift amount, change it in kutrace\_lib.cc also \*/

timer\_value = rdtsc() >> 6; /\* Both AMD and Intel \*/

#else

#error Define the time counter for your architecture

timer\_value = 0;

#endif

return timer\_value;

}

/\* Read instructions retired counter \*/

/\* This is performance critical -- every trace entry if tracking IPC \*/

/\*----------------------------------------------------------------------------\*/

inline u64 ku\_get\_inst\_retired(void)

{

#if IsAmd\_64

u32 a = 0, d = 0;

int ecx = IRPerfCount; /\* What counter it selects, AMD \*/

\_\_asm \_\_volatile("rdmsr" : "=a"(a), "=d"(d) : "c"(ecx));

return ((u64)a) | (((u64)d) << 32);

#elif IsIntel\_64

u32 a = 0, d = 0;

int ecx = IA32\_FIXED\_CTR0; /\* What counter it selects, Intel \*/

\_\_asm \_\_volatile("rdmsr" : "=a"(a), "=d"(d) : "c"(ecx));

return ((u64)a) | (((u64)d) << 32);

#elif IsArm\_64

u64 value = 0;

/\* set up pmevtyper2<15:0> to count INST\_RETIRED =0x08 \*/

/\* set up pmcntenset<0>=1<<2 to enable \*/

asm volatile("mrs %x0, pmevcntr2\_el0" : "=r" (value));

return value;

#else

#error Define inst\_retired for your architecture

return 0;

#endif

}

**/\* Read LLC misses counter \*/**

**/\* This is performance critical -- every trace entry if tracking LLC \*/**

**/\*----------------------------------------------------------------------------\*/**

**inline u64 ku\_get\_llc\_miss(void)**

**{**

**#if IsIntel\_64**

**u32 a = 0, d = 0;**

**int ecx = IA32\_PMC1; /\* What counter it selects, Intel \*/**

**\_\_asm \_\_volatile("rdmsr" : "=a"(a), "=d"(d) : "c"(ecx));**

**return ((u64)a) | (((u64)d) << 32);**

**#elif IsArm\_64**

**u64 value = 0;**

**/\* set up pmevtyper2<15:0> to count LLC\_MISSES =0x33 \*/**

**/\* set up pmcntenset<0>=1<<2 to enable \*/**

**asm volatile("mrs %x0, pmevcntr2\_el0" : "=r" (value));**

**return value;**

**#else**

**#error Define llc\_miss for your architecture**

**return 0;**

**#endif**

**}**

/\* Read current CPU frequency \*/

/\* Not performance critical -- once every timer interrupt \*/

/\*----------------------------------------------------------------------------\*/

inline u64 ku\_get\_cpu\_freq(void) {

#if !BCLK\_FREQ

return 0;

#elif IsAmd\_64

/\* Sample the CPU clock frequency and include with PC sample \*/

u64 curr = rdMSR(PStateStat) & PStat\_MASK;

u64 freq = rdMSR(PStateDef0 + curr);

u64 fid = (freq >> CpuFid\_SHIFT) & CpuFid\_MASK;

u64 did = (freq >> CpuDid\_SHIFT) & CpuDid\_MASK;

freq = (fid \* BCLK\_FREQ) / did;

return freq;

#elif IsIntel\_64

u64 freq = rdMSR(MSR\_PERF\_STATUS);

freq = (freq >> FID\_SHIFT) & FID\_MASK;

freq \*= BCLK\_FREQ; /\* base clock in MHz \*/

return freq;

#else

#error Define cpu\_freq for your architecture

return 0;

#endif

}

/\* Return true for large time advance that should be treated as small backward time \*/

inline bool LateStoreOrLarge(u64 delta\_cycles) {

return delta\_cycles > kLateStoreThresh;

}

/\* Make sure name length fits in 1..8 u64's \*/

//\* Return true if out of range \*/

inline bool is\_bad\_len(int len)

{

return (len < 1) | (len > 8);

}

/\* Make sure name length fits in 1 + 1..8 u64's \*/

/\* Return true if out of range \*/

inline bool is\_bad\_len\_plus(int len)

{

return (len < 1) | (len > 9);

}

/\* Turn off tracing. (We cannot wait here) \*/

/\* Return tracing bit \*/

static u64 do\_trace\_off(void)

{

kutrace\_tracing = false;

return kutrace\_tracing;

}

/\* Turn on tracing. We can only get here if all is set up \*/

/\* Trace buffer must be allocated and initialized \*/

/\* Return tracing bit \*/

static u64 do\_trace\_on(void)

{

kutrace\_tracing = true;

return kutrace\_tracing;

}

/\* Flush all partially-filled trace blocks, filling them up \*/

/\* Tracing must be off \*/

/\* Return number of words zeroed \*/

static u64 do\_flush(void)

{

u64 \*p;

int cpu;

int zeroed = 0;

kutrace\_tracing = false; /\* Should already be off \*/

for\_each\_online\_cpu(cpu)

{

struct kutrace\_traceblock \*tb =

&per\_cpu(kutrace\_traceblock\_per\_cpu, cpu);

u64 \*next\_item = (u64 \*)ATOMIC\_READ(&tb->next);

u64 \*limit\_item = tb->limit;

if (next\_item == NULL)

continue;

if (limit\_item == NULL)

continue;

for (p = next\_item; p < limit\_item; ++p)

{

\*p = 0;

++zeroed;

}

ATOMIC\_SET(&tb->next, (uintptr\_t)limit\_item);

}

return zeroed;

}

/\* Return number of filled trace blocks \*/

/\* Next can overshoot limit when we are full \*/

/\* Tracing will usually be on \*/

/\* NOTE: difference of two u64\* values is 1/8 of what you might be thinking \*/

static u64 do\_stat(void)

{

if (did\_wrap\_around || (traceblock\_next < traceblock\_limit))

return (u64)(traceblock\_high -

traceblock\_limit) >> KUTRACEBLOCKSHIFTU64;

else

return (u64)(traceblock\_high -

traceblock\_next) >> KUTRACEBLOCKSHIFTU64;

}

/\* Return number of filled trace words \*/

/\* Tracing must be off and flush must have been called \*/

/\* NOTE: difference of two u64\* values is 1/8 of what you might be thinking \*/

static u64 get\_count(void)

{

u64 retval;

kutrace\_tracing = false;

if (did\_wrap\_around || (traceblock\_next < traceblock\_limit))

retval = (u64)(traceblock\_high - traceblock\_limit);

else

retval = (u64)(traceblock\_high - traceblock\_next);

return retval;

}

/\* Read and return one u64 word of trace data, working down from top.

\* This is called 1M times to dump 1M trace words (8MB), but it is called

\* by a user program that is writing all this to disk, thus is constrained

\* by disk I/O speed. So we don't care that this is somewhat inefficient

\*

\* traceblock\_limit traceblock\_next traceblock\_high

\* | | |

\* v v v

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* | / / / / / / / / / / / / / / / | 3 2 1 0 |

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* <==== allocated blocks grow down

\*/

/\* Read and return one u64 word of trace data, working down from top.

\* This is called 1M times to dump 1M trace words (8MB), but it is called

\* by a user program that is writing all this to disk, so is constrained

\* by disk I/O speed. So we don't care that this is somewhat inefficient

\*/

/\* Tracing must be off and flush must have been called \*/

static u64 get\_word(u64 subscr)

{

u64 blocknum, u64\_within\_block;

u64 \*blockp;

kutrace\_tracing = false; /\* Should already be off \*/

if (subscr >= get\_count()) return 0;

blocknum = subscr >> KUTRACEBLOCKSHIFTU64;

u64\_within\_block = subscr & ((1 << KUTRACEBLOCKSHIFTU64) - 1);

blockp = traceblock\_high - ((blocknum + 1) << KUTRACEBLOCKSHIFTU64);

/\* printk(KERN\_INFO "get\_word[%lld] %016llx\n", subscr, blockp[u64\_within\_block]); \*/

return blockp[u64\_within\_block];

}

/\* Read and return one u64 word of IPC data, working down from top.

\*

\* Trace memory layout with IPC tracing. IPC bytes go into lower 1/8.

\* tracebase

\* | traceblock\_limit traceblock\_next traceblock\_high

\* | | | |

\* v v v v

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* |////| | / / / / / / / / / / / | |

\* +-------+-------+------+--------+-------+-------+-------+-------+

\* <== <==== allocated blocks grow down

\* IPC bytes

\*/

/\* Tracing must be off and flush must have been called \*/

/\* We map linear IPCword numbers 0..get\_count-1 to IPC block and offset, \*/

/\* with blocks growing downward. If mains trace blocks are 64KB, \*/

/\* IPC blocks are 8KB \*/

/\* Even though they are byte entries, we read them out as u64's \*/

static u64 get\_ipc\_word(u64 subscr)

{

u64 blocknum, u64\_within\_block;

u64 \*blockp;

kutrace\_tracing = false;

/\* IPC word count is 1/8 of main trace count \*/

if (subscr >= (get\_count() >> 3))

return 0;

blocknum = subscr >> KUIPCBLOCKSHIFTU8;

u64\_within\_block = subscr & ((1 << KUIPCBLOCKSHIFTU8) - 1);

/\* IPC blocks count down from traceblock\_limit \*/

blockp = traceblock\_limit - ((blocknum + 1) << KUIPCBLOCKSHIFTU8);

return blockp[u64\_within\_block];

}

/\*

\* Fast 4KB trace buffer extraction: Copies 4KB at once to user space.

\* Because the syscall that gets here supplies only one argument, we use it as

\* the user-space target buffer address. For the kernel-space buffer offset we

\* use get4kb\_subscr.

\*

\* Caller must do KUTRACE\_CMD\_SET4KB to initialize this

\* and then must do KUTRACE\_CMD\_GET4KB and optionally KUTRACE\_CMD\_GETIPC4KB

\* for every block of 4KB/sizeof(traceword) = 512 words.

\*

\* 4KB extraction is measured to be about 165x faster than 8B extraction per

\* 64KB traceblock. Including buffered writes toward disk/SSD, this achieves

\* about 1400MB/s vs 25MB/s overall on an Intel i3 at 3.9 GHz, so about 55

\* times faster, i.e. 55x less CPU time during extraction.

\*

\* Calls to the 4KB commands when run against an older module will return ~0.

\* It is up to the caller to detect this and use the word-at-a-time routines.

\*

\*/

/\* Read and return one 4KB block of trace data, working down from top,

\* then increment get4kb\_subscr by 4KB/sizeof(traceword) = 512.

\* This is called 1M/512 = 2K times to dump 1M trace words (8MB).

\* Returns 0 for success, ~0 for unimplemented, and 1..4096 for partial copy.

\* Tracing must be off and flush must have been called

\*/

static u64 get\_4kb(u64 arg)

{

u64 blocknum, u64\_within\_block;

u64 \*blockp;

void \_\_user \*to\_user\_ptr;

const void \*from\_kernel\_ptr;

if (get4kb\_subscr >= get\_count())

return 4096;

blocknum = get4kb\_subscr >> KUTRACEBLOCKSHIFTU64;

u64\_within\_block = get4kb\_subscr & ((1 << KUTRACEBLOCKSHIFTU64) - 1);

blockp = traceblock\_high - ((blocknum + 1) << KUTRACEBLOCKSHIFTU64);

/\* printk(KERN\_INFO "get\_4kb[%lld] %016llx\n", get4kb\_subscr, blockp[u64\_within\_block]); \*/

to\_user\_ptr = (void \_\_user \*)arg;

from\_kernel\_ptr = (const void \*)(&blockp[u64\_within\_block]);

return copy\_to\_user(to\_user\_ptr, from\_kernel\_ptr, 4096);

}

/\* Read and return one 4KB block of IPC data, working down from top.

\* This is called 1M/512 = 2K times to dump 1M IPC words (8MB).

\* Returns 0 for success, ~0 for unimplemented, and 1..4096 for partial copy.

\* Tracing must be off and flush must have been called

\*/

static u64 get\_ipc\_4kb(u64 arg)

{

u64 blocknum, u64\_within\_block;

u64 \*blockp;

void \_\_user \*to\_user\_ptr;

const void \*from\_kernel\_ptr;

/\* IPC word count is 1/8 of main trace count \*/

if (get4kb\_subscr >= (get\_count() >> 3))

return 4096;

blocknum = get4kb\_subscr >> KUIPCBLOCKSHIFTU8;

u64\_within\_block = get4kb\_subscr & ((1 << KUIPCBLOCKSHIFTU8) - 1);

/\* IPC blocks count down from traceblock\_limit \*/

blockp = traceblock\_limit - ((blocknum + 1) << KUIPCBLOCKSHIFTU8);

to\_user\_ptr = (void \_\_user \*)arg;

from\_kernel\_ptr = (const void \*)(&blockp[u64\_within\_block]);

return copy\_to\_user(to\_user\_ptr, from\_kernel\_ptr, 4096);

}

/\* We are called with preempt disabled \*/

/\* We are called with interrupts disabled \*/

/\* We are called holding the lock that guards traceblock\_next \*/

/\* Cannot do printf or anything else here that could block \*/

static u64 \*initialize\_trace\_block(u64 \*init\_me, bool very\_first\_block,

struct kutrace\_traceblock \*tb)

{

u64 \*myclaim = NULL;

u64 cpu = smp\_processor\_id();

/\* For every traceblock, insert current process ID and name. This \*/

/\* gives the proper context when wraparound is enabled \*/

struct task\_struct \*curr = current;

/\* First word is rdtsc (time counter) with CPU# placed in top byte \*/

u64 block\_init\_counter = ku\_get\_timecount();

init\_me[0] = (block\_init\_counter & FULL\_TIMESTAMP\_MASK) |

(cpu << CPU\_NUMBER\_SHIFT);

/\* Second word is going to be corresponding gettimeofday(), \*/

/\* filled in via postprocessing \*/

/\* We put some flags in the top byte, though. x080 = do\_ipc bit \*/

init\_me[1] = 0;

if (do\_ipc) {

init\_me[1] |= (IPC\_Flag << FLAGS\_SHIFT);

**#ifdef TEST\_LLC\_MISS**

**init\_me[1] |= (LLC\_Flag << FLAGS\_SHIFT);**

**#endif**

}

if (do\_wrap)

init\_me[1] |= (WRAP\_Flag << FLAGS\_SHIFT);

/\* We don't know if we actually wrapped until the end. \*/

/\* See KUTRACE\_CMD\_GETCOUNT \*/

/\* For very first trace block, also insert six NOPs at [2..7]. \*/

/\* The dump to disk code will overwrite the first pair with \*/

/\* start timepair and the next with stop timepair. [6..7] unused \*/

if (very\_first\_block) {

init\_me[2] = CLU(0);

init\_me[3] = CLU(0);

init\_me[4] = CLU(0);

init\_me[5] = CLU(0);

init\_me[6] = CLU(0);

init\_me[7] = CLU(0);

myclaim = &init\_me[8];

} else {

myclaim = &init\_me[2];

}

/\* Every block has PID and pidname at the front \*/

/\* This requires a change for V3 in postprocessing \*/

/\* I feel like I should burn one more word here to make 4, \*/

/\* so entire front is 12/6 entries instead of 11/5... \*/

myclaim[0] = curr->pid;

myclaim[1] = 0;

memcpy(&myclaim[2], curr->comm, MAX\_PIDNAME\_LENGTH);

myclaim += 4;

/\* Next len words are the claimed space for an entry \*/

/\* Last 8 words of a block set to NOPs (0) \*/

init\_me[KUTRACEBLOCKSIZEU64 - 8] = 0;

init\_me[KUTRACEBLOCKSIZEU64 - 7] = 0;

init\_me[KUTRACEBLOCKSIZEU64 - 6] = 0;

init\_me[KUTRACEBLOCKSIZEU64 - 5] = 0;

init\_me[KUTRACEBLOCKSIZEU64 - 4] = 0;

init\_me[KUTRACEBLOCKSIZEU64 - 3] = 0;

init\_me[KUTRACEBLOCKSIZEU64 - 2] = 0;

init\_me[KUTRACEBLOCKSIZEU64 - 1] = 0;

/\* If this is the very first traceblock for this CPU, set up the MSRs \*/

/\* If there are 12 CPU cores (6 physical 2x hyperthreaded) this will happen 12 times \*/

{

bool first\_block\_per\_cpu = (tb->prior\_cycles == 0);

if (first\_block\_per\_cpu) {

ku\_setup\_timecount();

**#ifdef TEST\_LLC\_MISS**

**ku\_setup\_llc\_miss();**

**#else**

**ku\_setup\_inst\_retired();**

**#endif**

ku\_setup\_cpu\_freq();

tb->prior\_cycles = 1; /\* mark it as initialized \*/

#if IsArm\_64

{

struct cpufreq\_policy \*policy = cpufreq\_cpu\_get\_raw(cpu);

/\* For Rpi4, put current CPU freq (MHz) into block at high half of myclaim[-4] \*/

if (policy) {

u64 cpu\_freq\_mhz = policy->cur / 1000; /\* Khz to MHz \*/

myclaim[-4] |= (cpu\_freq\_mhz << 32);

/\*printk(KERN\_INFO "cpu %lld freq = %lld MHz\n", cpu, cpu\_freq\_mhz);\*/

}

}

#endif

}

}

return myclaim;

}

/\* We are called with preempt disabled \*/

/\* We are called with interrupts disabled \*/

/\* We are called holding the lock that guards traceblock\_next \*/

static u64 \*really\_get\_slow\_claim(int len, struct kutrace\_traceblock \*tb)

{

u64 \*myclaim = NULL;

bool very\_first\_block = (traceblock\_next == traceblock\_high);

/\* Allocate a new traceblock. Allocations grow downward. \*/

traceblock\_next -= KUTRACEBLOCKSIZEU64;

if (traceblock\_next < traceblock\_limit) {

if (do\_wrap) {

/\* Wrap to traceblock[1], not [0] \*/

did\_wrap\_around = true;

traceblock\_next = traceblock\_high -

2 \* KUTRACEBLOCKSIZEU64;

/\* Clear pid filter. \*/

/\* It is unfortunate to do this while holding a \*/

/\* lock and also holding off interrupts... \*/

memset(kutrace\_pid\_filter, 0, 1024 \* sizeof(u64));

} else {

/\* All full. Stop and get out. \*/

kutrace\_tracing = false;

return myclaim;

}

}

/\* Need to do this before setting next/limit if same CPU could get \*/

/\* an interrupt and use uninitilized block \*/

/\* It is unfortunate to do this while holding a lock and also \*/

/\* holding off interrupts... \*/

/\* Most of the cost is two cache misses, so maybe 200 nsec \*/

myclaim = initialize\_trace\_block(traceblock\_next, very\_first\_block, tb);

/\* Set up the next traceblock pointers, reserving \*/

/\* first N + len words \*/

ATOMIC\_SET(&tb->next, (uintptr\_t)(myclaim + len));

tb->limit = traceblock\_next + KUTRACEBLOCKSIZEU64;

return myclaim;

}

/\* Reserve space for one entry of 1..9 u64 words \*/

/\* If trace buffer is full, return NULL or wrap around \*/

/\* We allow this to be used with tracing off so we can initialize a trace file \*/

/\* In that case, tb->next and tb->limit are NULL \*/

/\* We are called with preempt disabled \*/

static u64 \*get\_slow\_claim(int len, struct kutrace\_traceblock \*tb)

{

unsigned long flags;

u64 \*limit\_item;

u64 \*myclaim = NULL;

if (is\_bad\_len(len)) {

kutrace\_tracing = false;

printk(KERN\_INFO "is\_bad\_len 1\n");

return NULL;

}

/\* This gets the lock that protects traceblock\_next and \*/

/\* disables interrupts \*/

raw\_spin\_lock\_irqsave(&kutrace\_lock, flags);

/\* Nothing else can be touching tb->limit now \*/

limit\_item = tb->limit;

/\* add\_return returns the updated pointer; we want the prior \*/

/\* so subtract len \*/

myclaim = ((u64 \*)ATOMIC\_ADD\_RETURN(len \* sizeof(u64), &tb->next)) -

len;

/\* FIXED BUG: myclaim + len \*/

if (((myclaim + len) >= limit\_item) || (limit\_item == NULL)) {

/\* Normal case: \*/

/\* the claim we got still doesn't fit in its block \*/

myclaim = really\_get\_slow\_claim(len, tb);

}

/\* Rare: If some interrupt already allocated a new traceblock, \*/

/\* fallthru to here \*/

/\* Free lock; re-enable interrupts if they were enabled on entry \*/

raw\_spin\_unlock\_irqrestore(&kutrace\_lock, flags);

return myclaim;

}

/\* Reserve space for one entry of 1..9 u64 words, normally lockless \*/

/\* If trace buffer is full, return NULL. Caller MUST check \*/

/\* We allow this to be used with tracing off so we can initialize a trace file \*/

/\* We are called with preempt disabled \*/

static u64 \*get\_claim(int len, struct kutrace\_traceblock\* tb)

{

u64 \*limit\_item = NULL;

u64 \*limit\_item\_again = NULL;

u64 \*myclaim = NULL;

if (is\_bad\_len\_plus(len)) {

kutrace\_tracing = false;

return NULL;

}

/\* Fast path \*/

/\* We may get interrupted at any point here and the interrupt routine

\* may create a trace entry, and it may even allocate a new

\* traceblock.

\* This code must carefully either reserve an exclusive area to use or

\* must call the slow path.

\*/

/\* Note that next and limit may both be NULL at initial use. \*/

/\* If they are, take the slow path without accessing. \*/

do {

limit\_item = tb->limit;

if (limit\_item == NULL)

break;

/\* add\_return returns the updated pointer; we want the \*/

/\* prior so subtract len \*/

myclaim =

((u64 \*)ATOMIC\_ADD\_RETURN(len \* sizeof(u64), &tb->next)) - len;

limit\_item\_again = tb->limit;

if (limit\_item == limit\_item\_again)

break; /\* All is good \*/

/\* An interrupt occurred \*and\* changed blocks \*/

if ((myclaim < limit\_item\_again) &&

((limit\_item\_again - KUTRACEBLOCKSIZEU64) <= myclaim))

/\* Claim is in new block -- use it \*/

break;

/\* Else claim is at end of old block -- abandon it, and try again \*/

} while (true);

/\* Make sure the entire allocation fits \*/

if ((myclaim + len) >= limit\_item\_again) {

/\* Either this is the first claim for a CPU \*/

/\* with limit\_item, limit\_item\_again, and myclaim all null, or \*/

/\* the claim we got doesn't fit in its block. Allocate a new block. \*/

myclaim = get\_slow\_claim(len, tb);

}

return myclaim;

}

/\*

\* In recording a trace event, it is possible for an interrupt to happen after

\* KUtrace code takes the event timestamp and before it claims the storage location.

\* In this case, the interupt handling will recursively record several events

\* before returning to the original KUtrace path, which then claims a location

\* and stores the original event with its earlier timestamp. This is called a

\* "late store." When that happens, the reconstruciton in rawtoevent needs to

\* decide whether time went forward by almost the entire 20-bit wraparound

\* period, or went backward by some amount.

\*

\* To resolve this ambiguity, we declare that a time gap of 7/8 of the wraparound

\* period is forward time and the high 1/8 is backward time associated with an

\* otherwise undetectable backward time.

\*

\* To mark forward time in that 1/8 (and above), we add a TSDELTA entry to the

\* trace. The exact compare for late store must be identical in kutrace\_mod.c

\* and in rawtoevent.cc.

\*

\*/

/\* Get a claim. If delta\_cycles is large, claim one more word and insert TSDELTA entry \*/

/\* NOTE: tsdelta is bogus for very first entry per CPU. \*/

/\* First per CPU is indicated by tb->prior\_cycles == 0 \*/

/\* We are called with preempt disabled \*/

inline u64\* get\_claim\_with\_tsdelta(u64 now, u64 delta\_cycles,

int len, struct kutrace\_traceblock\* tb) {

u64 \*claim;

/\* Check if time between events almost wraps above the 20-bit timestamp \*/

if (LateStoreOrLarge(delta\_cycles) && (tb->prior\_cycles != 0)) {

/\* Uncommon case. Add timestamp delta entry before original entry \*/

claim = get\_claim(1 + len, tb);

if (claim != NULL) {

claim[0] = (now << TIMESTAMP\_SHIFT) |

((u64)KUTRACE\_TSDELTA << EVENT\_SHIFT) |

(delta\_cycles & ARG\_MASK);

++claim; /\* Start of space for original entry \*/

}

} else {

/\* Common case \*/

claim = get\_claim(len, tb); /\* Start of space for original entry \*/

}

return claim;

}

/\* Return prior trace word for this CPU or NULL \*/

/\* We are called with preempt disabled \*/

inline static u64 \*get\_prior(struct kutrace\_traceblock \*tb)

{

u64 \*next\_item;

u64 \*limit\_item;

/\* Note that next and limit may both be NULL at initial use. \*/

/\* If they are, or any other problem, return NULL \*/

/\* get\_cpu\_var disables preempt \*/

tb = &get\_cpu\_var(kutrace\_traceblock\_per\_cpu);

next\_item = (u64 \*)ATOMIC\_READ(&tb->next);

limit\_item = tb->limit;

put\_cpu\_var(kutrace\_traceblock\_per\_cpu);

if (next\_item < limit\_item)

return next\_item - 1; /\* ptr to prior entry \*/

return NULL;

}

/\* Calculate and insert four-bit IPC value. Shift puts in lo/hi part of a byte \*/

**inline void do\_ipc\_calc(u64 \*claim, u64 delta\_cycles,**

**struct kutrace\_traceblock\* tb, bool shift) {**

u64 inst\_ret;

u64 delta\_inst;

u64 ipc;

u8\* ipc\_byte\_addr;

if (!do\_ipc) {return;}

**#ifdef TEST\_LLC\_MISS**

**inst\_ret = ku\_get\_llc\_miss();**

**#else**

**/\* There will be random large differences the first time; we don't care. \*/**

**inst\_ret = ku\_get\_inst\_retired();**

**#endif**

delta\_inst = inst\_ret - tb->prior\_inst\_retired;

tb->prior\_inst\_retired = inst\_ret;

/\* NOTE: pointer arithmetic divides claim by 8, giving the byte offset we want \*/

ipc\_byte\_addr = (u8\*)(tracebase) + (claim - (u64\*)(tracebase));

#**ifdef TEST\_LLC\_MISS**

**ipc = get\_granular\_llc(delta\_inst);**

**#else**

**ipc = get\_granular(delta\_inst, delta\_cycles);**

**#endif**

if (shift)

ipc\_byte\_addr[0] |= ipc << 4;

else

ipc\_byte\_addr[0] = ipc;

}

/\*

\* arg1: (arrives with timestamp = 0x00000)

\* +-------------------+-----------+---------------+-------+-------+

\* | timestamp | event | delta | retval| arg0 |

\* +-------------------+-----------+---------------+-------+-------+

\* 20 12 8 8 16

\*/

/\* Insert one u64 trace entry, for current CPU \*/

/\* Tracing may be otherwise off \*/

/\* Return number of words inserted \*/

**static u64 insert\_1(u64 arg1)**

{

u64 \*claim;

struct kutrace\_traceblock\* tb;

u64 delta\_cycles;

u64 retval = 0;

u64 now = ku\_get\_timecount();

tb = &get\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* hold off preempt \*/

delta\_cycles = now - tb->prior\_cycles;

/\* Allocate one word \*/

claim = get\_claim\_with\_tsdelta(now, delta\_cycles, 1, tb);

/\* This update must be after the first getclaim per CPU \*/

tb->prior\_cycles = now;

if (claim != NULL) {

claim[0] = arg1 | (now << TIMESTAMP\_SHIFT);

/\* IPC option. Changes CPU overhead from ~1/4% to ~3/4% \*/

**do\_ipc\_calc(claim, delta\_cycles, tb, false);**

retval = 1;

}

put\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* release preempt \*/

return retval;

}

/\* Insert one u64 Return trace entry with small retval, for current CPU \*/

/\* Optimize by combining with just-previous entry if the matching call \*/

/\* and delta\_t fits. The optimization is likely, so we don't worry about \*/

/\* the overhead if we can't optimize \*/

/\* Tracing may be otherwise off \*/

/\* Return number of words inserted \*/

**static u64 insert\_1\_retopt(u64 arg1)**

{

struct kutrace\_traceblock\* tb;

u64 \*prior\_entry;

u64 now = ku\_get\_timecount();

/\* No need to hold off preempt here, but get\_cpu/put\_cpu do anyway \*/

/\* It doesn't matter if we get migrated because we are not allocating a new entry \*/

tb = &get\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* hold off preempt \*/

prior\_entry = get\_prior(tb);

if (prior\_entry != NULL) {

/\* Want N=matching call, high bytes of return value = 0 \*/

u64 diff = (\*prior\_entry ^ arg1) & EVENT\_DELTA\_RETVAL\_MASK;

u64 prior\_t = \*prior\_entry >> TIMESTAMP\_SHIFT;

u64 delta\_t = (now - prior\_t) & UNSHIFTED\_TIMESTAMP\_MASK;

/\* EVENT\_RETURN\_BIT distinguishes call from return \*/

if ((diff == EVENT\_RETURN\_BIT) && (delta\_t <= MAX\_DELTA\_VALUE))

{

/\* Successful optimization tests. Combine ret with call. \*/

/\* This happens about 90-95% of the time \*/

u64 opt\_ret;

/\* make sure delta\_t is nonzero to flag there is an optimized ret \*/

if (delta\_t == 0)

delta\_t = 1;

opt\_ret = (delta\_t << DELTA\_SHIFT) |

((arg1 & UNSHIFTED\_RETVAL\_MASK) << RETVAL\_SHIFT);

\*prior\_entry |= opt\_ret;

/\* IPC option. Changes CPU overhead from ~1/4% to ~3/4% \*/

**do\_ipc\_calc(prior\_entry, delta\_t, tb, true);**

put\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* release preempt \*/

return 0;

}

}

put\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* release preempt \*/

/\* Otherwise, fall into normal insert\_1 \*/

**return insert\_1(arg1);**

}

/\* Insert a two-word u64 trace entry, for current CPU \*/

/\* The entry is exactly a PC\_TEMP sample \*/

/\* Tracing may be otherwise off \*/

/\* Return number of words inserted \*/

**static u64 insert\_2(u64 arg1, u64 arg2)**

{

u64 \*claim;

struct kutrace\_traceblock\* tb;

u64 delta\_cycles;

u64 now = ku\_get\_timecount();

tb = &get\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* hold off preempt \*/

delta\_cycles = now - tb->prior\_cycles;

/\* Allocate two words \*/

claim = get\_claim\_with\_tsdelta(now, delta\_cycles, 2, tb);

/\* This update must be after the first getclaim per CPU \*/

tb->prior\_cycles = now;

put\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* release preempt \*/

if (claim != NULL)

{

claim[0] = arg1 | (now << TIMESTAMP\_SHIFT);

claim[1] = arg2;

return 2;

}

return 0;

}

/\* For event codes 010..1FF, length is middle hex digit. All others 1 \*/

static u64 entry\_len(u64 word)

{

u64 n = (word >> EVENT\_SHIFT) & UNSHIFTED\_EVENT\_MASK;

if (n > MAX\_EVENT\_WITH\_LENGTH)

return 1;

if (n < MIN\_EVENT\_WITH\_LENGTH)

return 1;

return (n >> EVENT\_LENGTH\_FIELD\_SHIFT) & EVENT\_LENGTH\_FIELD\_MASK;

}

/\* Insert one trace entry of 1..8 u64 words, for current CPU \*/

/\* word is actually a const u64\* pointer to kernel space array of \*/

/\* exactly len u64 \*/

/\* Tracing may be otherwise off \*/

/\* Return number of words inserted \*/

static u64 insert\_n\_krnl(u64 word)

{

const u64 \*krnlptr = (const u64 \*)word;

u64 len = entry\_len(krnlptr[0]); /\* length in u64, 1..8 \*/

u64 \*claim;

struct kutrace\_traceblock\* tb;

u64 delta\_cycles;

u64 now = ku\_get\_timecount();

tb = &get\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* hold off preempt \*/

delta\_cycles = now - tb->prior\_cycles;

/\* Allocate N words \*/

claim = get\_claim\_with\_tsdelta(now, delta\_cycles, len, tb);

/\* This update must be after the first getclaim per CPU \*/

tb->prior\_cycles = now;

put\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* release preempt \*/

if (claim != NULL) {

claim[0] = krnlptr[0] | (now << TIMESTAMP\_SHIFT);

memcpy(&claim[1], &krnlptr[1], (len - 1) \* sizeof(u64));

return len;

}

return 0;

}

/\* Insert one trace entry of 1..8 u64 words, for current CPU \*/

/\* word is actually a const u64\* pointer to user space array of \*/

/\* exactly eight u64 \*/

/\* NOTE: Always copies eight words, even if actual length is smaller \*/

/\* Tracing may be otherwise off \*/

/\* Return number of words inserted \*/

static u64 insert\_n\_user(u64 word)

{

const uintptr\_t tempword = word; /\* 32- or 64-bit pointer \*/

const u64 \*userptr = (const u64 \*)tempword;

u64 len;

u64 \*claim;

struct kutrace\_traceblock\* tb;

u64 delta\_cycles;

u64 now;

u64 uncopied\_bytes;

u64 temp[8];

/\* This call may sleep or otherwise context switch \*/

/\* It may fail if passed a bad user-space pointer. Don't do that. \*/

temp[0] = 0;

uncopied\_bytes = raw\_copy\_from\_user(temp, userptr, 8 \* sizeof(u64));

if (uncopied\_bytes > 0)

return 0;

len = entry\_len(temp[0]); /\* length in u64, 1..8 \*/

now = ku\_get\_timecount();

tb = &get\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* hold off preempt \*/

delta\_cycles = now - tb->prior\_cycles;

/\* Allocate N words \*/

claim = get\_claim\_with\_tsdelta(now, delta\_cycles, len, tb);

/\* This update must be after the first getclaim per CPU \*/

tb->prior\_cycles = now;

put\_cpu\_var(kutrace\_traceblock\_per\_cpu); /\* release preempt \*/

if (claim != NULL) {

temp[0] |= (now << TIMESTAMP\_SHIFT);

memcpy(claim, temp, len \* sizeof(u64));

return len;

}

return 0;

}

/\*

\* pid filter is an array of 64K bits, arranged as 1024 u64. It

\* cleared. When tracing context switches in kernel/sched/core.c, the

\* intended use is to check if the bit corresponding to next->pid & 0xffff is

\* off and if so put the process name next->comm[TASK\_COMM\_LEN]; from

\* task\_struct into the trace as a pid\_name entry, then set the bit.

\*/

/\* Reset tracing state to start a new clean trace \*/

/\* Tracing must be off. tracebase must be non-NULL \*/

/\* traceblock\_next always points \*just above\* the next block to use \*/

/\* When empty, traceblock\_next == traceblock\_high \*/

/\* when full, traceblock\_next == traceblock\_limit \*/

/\* Return 0 \*/

static u64 do\_reset(u64 flags)

{

int cpu;

/\* printk(KERN\_INFO " kutrace\_trace reset(%016llx) called\n", flags); \*/

/\* Turn off tracing -- should already be off \*/

kutrace\_tracing = false; /\* Should already be off \*/

do\_ipc = ((flags & DO\_IPC) != 0);

do\_wrap = ((flags & DO\_WRAP) != 0);

/\* Clear pid filter \*/

memset(kutrace\_pid\_filter, 0, 1024 \* sizeof(u64));

/\* Set up trace buffer into a series of blocks of 64KB each \*/

traceblock\_high = (u64 \*)(tracebase + (tracemb << 20));

traceblock\_limit = (u64 \*)(tracebase);

/\* First trace item inserted will cause first new block \*/

traceblock\_next = traceblock\_high;

did\_wrap\_around = false;

if (do\_ipc) {

/\* Reserve lower 1/8 of trace buffer for IPC bytes \*/

/\* Strictly speaking, this should be 1/9. We waste a little space. \*/

traceblock\_limit = (u64\*)(tracebase + (tracemb << (20 - 3)));

}

/\* Set up spinlock as available \*/

raw\_spin\_lock\_init(&kutrace\_lock);

/\* Set up per-CPU limits to immediately allocate a block \*/

for\_each\_online\_cpu(cpu) {

struct kutrace\_traceblock \*tb =

&per\_cpu(kutrace\_traceblock\_per\_cpu, cpu);

ATOMIC\_SET(&tb->next, (uintptr\_t)NULL);

tb->limit = NULL;

tb->prior\_cycles = 0; // IPC design

tb->prior\_inst\_retired = 0; // IPC design

}

return 0;

}

/\* Called from kernel patches \*/

/\* Caller is responsible for making sure event fits in 12 bits and \*/

/\* arg fits in 16 bits for syscall/ret and 32 bits otherwise \*/

**static /\*asmlinkage\*/ void trace\_1(u64 event, u64 arg)**

{

if (!kutrace\_tracing)

return;

/\* Check for possible return optimization \*/

if (((event & UNSHIFTED\_EVENT\_RETURN\_BIT) != 0) &&

((event & UNSHIFTED\_EVENT\_HAS\_RETURN\_MASK) != 0))

{

/\* We have a return entry 011x, 101x, 111x: 6/7, a/b, e/f \*/

if (((arg + 128l) & ~UNSHIFTED\_RETVAL\_MASK) == 0) {

/\* Signed retval fits into a byte, [-128..127] \*/

**insert\_1\_retopt((event << EVENT\_SHIFT) | arg);**

return;

}

}

/\* Non-optimized insert \*/

**insert\_1((event << EVENT\_SHIFT) | (arg & CLU(0xffffffff)));**

}

/\* Called from kernel patches \*/

/\* ONLY called to insert PC sample at timer interrupt \*/

/\* arg1 is unused (0), arg2 is the 64-bit PC sample \*/

static void trace\_2(u64 event, u64 arg1, u64 arg2)

{

u64 freq;

if (!kutrace\_tracing)

return;

/\* dsites 2021.04.05 insert CPU frequency \*/

freq = ku\_get\_cpu\_freq();

**insert\_2((event << EVENT\_SHIFT) | freq, arg2);**

}

/\* Called from kernel patches \*/

static void trace\_many(u64 event, u64 len, const char \*arg)

{

uintptr\_t tempptr; /\* 32- or 64-bit address \*/

u64 temp[8];

if (!kutrace\_tracing)

return;

/\* Turn off tracing if bogus length \*/

if (is\_bad\_len(len)) {

kutrace\_tracing = false;

return;

}

memcpy(temp, arg, len \* sizeof(u64));

temp[0] |= (event | (len << EVENT\_LENGTH\_FIELD\_SHIFT)) << EVENT\_SHIFT;

tempptr = (uintptr\_t)&temp[0];

insert\_n\_krnl((u64)tempptr);

}

/\* Syscall from user space via kernel patch \*/

static u64 kutrace\_control(u64 command, u64 arg)

{

/\*

\* printk(KERN\_INFO " kutrace\_control: %08x %08x %08x %08x\n",

\* (u32)(command & 0xFFFFFFFF), (u32)(command >> 32),

\* (u32)(arg & 0xFFFFFFFF), (u32)(arg >> 32));

\*/

if (tracebase == NULL) {

/\* Error! \*/

printk(KERN\_INFO " kutrace\_control called with no trace buffer.\n");

kutrace\_tracing = false;

return ~CLU(0);

}

/\* If checking, disallow calls from tasks without CAP\_SYS\_PTRACE \*/

// XXX: We assume that the current user has the capability to trace.

// Android does not seem to export the has\_capability function for use

// with kernel modules

// if (check && !has\_capability(current, CAP\_SYS\_PTRACE))

// return ~CLU(0);

/\* Generally, more likely calls are near the front of this list \*/

if (command == KUTRACE\_CMD\_OFF) {

return do\_trace\_off();

} else if (command == KUTRACE\_CMD\_INSERT1) {

/\* If not tracing, insert nothing \*/

if (!kutrace\_tracing)

return 0;

**return insert\_1(arg);**

} else if (command == KUTRACE\_CMD\_INSERTN) {

/\* If not tracing, insert nothing \*/

if (!kutrace\_tracing)

return 0;

return insert\_n\_user(arg);

} else if (command == KUTRACE\_CMD\_GETWORD) {

return get\_word(arg);

} else if (command == KUTRACE\_CMD\_GETIPCWORD) {

return get\_ipc\_word(arg);

} else if (command == KUTRACE\_CMD\_ON) {

return do\_trace\_on();

} else if (command == KUTRACE\_CMD\_FLUSH) {

return do\_flush();

} else if (command == KUTRACE\_CMD\_RESET) {

return do\_reset(arg);

} else if (command == KUTRACE\_CMD\_STAT) {

return do\_stat();

} else if (command == KUTRACE\_CMD\_GETCOUNT) {

if (did\_wrap\_around) {

/\* Convey that we actually wrapped \*/

return ~get\_count();

} else {

return get\_count();

}

} else if (command == KUTRACE\_CMD\_TEST) {

return kutrace\_tracing; /\* Just 0/1 for tracing off/on \*/

} else if (command == KUTRACE\_CMD\_VERSION) {

return kModuleVersionNumber;

} else if (command == ~KUTRACE\_CMD\_INSERT1) {

/\* Allow kutrace\_control to insert entries with tracing off \*/

**return insert\_1(arg);**

} else if (command == ~KUTRACE\_CMD\_INSERTN) {

/\* Allow kutrace\_control to insert entries with tracing off \*/

return insert\_n\_user(arg);

} else if (command == KUTRACE\_CMD\_SET4KB) {

/\* This returns 0 for success. \*/

/\* Older module versions will return ~0 for unknown command \*/

get4kb\_subscr = arg;

return 0;

} else if (command == KUTRACE\_CMD\_GET4KB) {

return get\_4kb(arg);

} else if (command == KUTRACE\_CMD\_GETIPC4KB) {

return get\_ipc\_4kb(arg);

}

/\* Else quietly return -1 \*/

return ~CLU(0);

}

/\*

\* For the compiled-into-the-kernel design, call this at first

\* kutrace\_control call to set up trace buffers, etc.

\*/

static int \_\_init kutrace\_mod\_init(void)

{

printk(KERN\_INFO "\nkutrace\_trace hello =====================\n");

kutrace\_tracing = false;

kutrace\_pid\_filter = (u64 \*)vmalloc(1024 \* sizeof(u64));

printk(KERN\_INFO " vmalloc kutrace\_pid\_filter " FUINTPTRX "\n",

(uintptr\_t)kutrace\_pid\_filter);

if (!kutrace\_pid\_filter)

return -1;

tracebase = vmalloc(tracemb << 20);

printk(KERN\_INFO " vmalloc kutrace\_tracebase(%ld MB) " FUINTPTRX " %s\n",

tracemb,

(uintptr\_t)tracebase,

(tracebase == NULL) ? "FAIL" : "OK");

if (!tracebase) {

vfree(kutrace\_pid\_filter);

return -1;

}

/\* Set up TCP packet filter \*/

/\* Filter forms a hash over masked first N=24 bytes of packet payload \*/

/\* and looks for zero result. The hash is just u32 XOR along with \*/

/\* an initial value. pktmask gives mask bit-per-byte, and pktmatch \*/

/\* gives the expected result over those bytes. It is the \*/

/\* inital hash value, to give a simple zero test at the end. \*/

if (pktmask == 0) {

// Match nothing

kutrace\_net\_filter.hash\_mask[0] = 0LLU;

kutrace\_net\_filter.hash\_mask[1] = 0LLU;

kutrace\_net\_filter.hash\_mask[2] = 0LLU;

kutrace\_net\_filter.hash\_init = 1; // hash will always be zero

} else if (pktmask == -1) {

// Match everything

kutrace\_net\_filter.hash\_mask[0] = 0LLU;

kutrace\_net\_filter.hash\_mask[1] = 0LLU;

kutrace\_net\_filter.hash\_mask[2] = 0LLU;

kutrace\_net\_filter.hash\_init = 0; // hash will always be zero

} else {

int i;

u8 \*msk = (u8\*)(kutrace\_net\_filter.hash\_mask);

for (i = 0; i < 24; ++i) {

if ((pktmask >> i) & 1) {msk[i] = 0xFF;}

else {msk[i] = 0x00;}

}

kutrace\_net\_filter.hash\_init = (u64)(pktmatch);

}

printk(KERN\_INFO " mask %016llx", kutrace\_net\_filter.hash\_mask[0]);

printk(KERN\_INFO " mask %016llx", kutrace\_net\_filter.hash\_mask[1]);

printk(KERN\_INFO " mask %016llx", kutrace\_net\_filter.hash\_mask[2]);

printk(KERN\_INFO " == %016llx", kutrace\_net\_filter.hash\_init);

#if IsAmd\_64

printk(KERN\_INFO "IsAmd\_64");

#endif

#if IsIntel\_64

printk(KERN\_INFO "IsIntel\_64");

#endif

#if IsArm\_64

printk(KERN\_INFO "IsArm\_64");

#endif

/\* Set up global tracing data state \*/

/\* Very first traceblock alloc per CPU will do this, but we need \*/

/\* the timecount set up before we write teh first trace entry \*/

ku\_setup\_timecount();

ku\_setup\_inst\_retired();

ku\_setup\_cpu\_freq();

do\_reset(0);

printk(KERN\_INFO " kutrace\_tracing = %d\n", kutrace\_tracing);

/\* Finally, connect up the routines that can change the state \*/

kutrace\_global\_ops.kutrace\_trace\_1 = &trace\_1;

kutrace\_global\_ops.kutrace\_trace\_2 = &trace\_2;

kutrace\_global\_ops.kutrace\_trace\_many = &trace\_many;

kutrace\_global\_ops.kutrace\_trace\_control = &kutrace\_control;

printk(KERN\_INFO " &kutrace\_global\_ops: " FUINTPTRX "\n", (uintptr\_t)(&kutrace\_global\_ops));

printk(KERN\_INFO " kutrace\_trace All done init successfully!\n");

return 0;

}

static void \_\_exit kutrace\_mod\_exit(void)

{

int cpu;

printk(KERN\_INFO "kutrace\_mod Winding down =====================\n");

/\* Turn off tracing and quiesce \*/

kutrace\_tracing = false;

msleep(20); /\* wait 20 msec for any pending tracing to finish \*/

printk(KERN\_INFO " kutrace\_tracing=false\n");

/\* Disconnect allthe routiens that can change state \*/

kutrace\_global\_ops.kutrace\_trace\_1 = NULL;

kutrace\_global\_ops.kutrace\_trace\_2 = NULL;

kutrace\_global\_ops.kutrace\_trace\_many = NULL;

kutrace\_global\_ops.kutrace\_trace\_control = NULL;

printk(KERN\_INFO " kutrace\_global\_ops = NULL\n");

/\* Clear out all the pointers to trace data \*/

for\_each\_online\_cpu(cpu) {

struct kutrace\_traceblock\* tb = &per\_cpu(kutrace\_traceblock\_per\_cpu, cpu);

printk(KERN\_INFO " kutrace\_traceblock\_per\_cpu[%d] = NULL\n", cpu);

ATOMIC\_SET(&tb->next, (uintptr\_t)NULL);

tb->limit = NULL;

tb->prior\_cycles = 0; // IPC design

tb->prior\_inst\_retired = 0; // IPC design

}

traceblock\_high = NULL;

traceblock\_limit = NULL;

traceblock\_next = NULL;

/\* Now that nothing points to it, free memory \*/

if (tracebase) {vfree(tracebase);}

if (kutrace\_pid\_filter) {vfree(kutrace\_pid\_filter);}

kutrace\_pid\_filter = NULL;

printk(KERN\_INFO " kutrace\_tracebase = NULL\n");

printk(KERN\_INFO " kutrace\_pid\_filter = NULL\n");

printk(KERN\_INFO "kutrace\_\_mod Goodbye\n");

}

module\_init(kutrace\_mod\_init);

module\_exit(kutrace\_mod\_exit);