

# **Kernel function instrumentation - tool analysis**

Nicholas Mc Guire

Opentech EDV-Research GmbH

February 12, 2005

## Contents

<b>1. Kernel function instrumentation - tool analysis</b>	<b>1</b>
1.1. Source . . . . .	1
1.2. Patch file . . . . .	1
1.3. Patch analysis . . . . .	1
1.4. Basic technology . . . . .	2
1.5. Installation . . . . .	7
1.6. Data aquisition . . . . .	8
1.7. Dynamic Data aquisitionion (post boot) . . . . .	9
1.8. Data interpretation . . . . .	13
1.9. Performanc Impact . . . . .	13
<b>2. Conclusion</b>	<b>13</b>
<b>3. List of Acronyms</b>	<b>15</b>

*Contents*

---

Version	Author	Date	Comment
1.0	Nicholas Mc Guire	Jan 2005	First shot
1.1	Georg Schiesser	18 Jan 2005	converted to TEX document

## 1. Kernel function instrumentation - tool analysis

### 1.1. Source

- `kfi-0.8.tar.gz` - analysis tools and a minimum documentation (`README.kfi`)
- patch for 2.4.20 (only ?) not too invasive - coded as standalone modules.

dependencies: `bigphysarea` (if you want to get usable dynamic traces out of this)

### 1.2. Patch file

- `driver/char/kfi` - isolated kernel function instrumentation "driver"
- `kernel/sys.c` - a few `kfi_dump_log` inserted
- `init/main.c` - this patch seems useless - all it does is add a call to an empty function - not clear what this is supposed to achieve (TODO: clarify).

patch applies clean against stock 2.4.20, fails uncritically against 2.4.26 (Makefile, and `config.in` - `.rej` files sufficient to patch) `kfi driver` is sufficiently isolated to patch into most likely any 2.4.X kernel.

Name `test4` a bit irritating - there is no doc what `test1-3` were about ...

### 1.3. Patch analysis

Generally not really invasive functionally - well isolated - should be trivial to port.

Patch does not seem to be really arch dependant - currently it seems configurable for `amr`, `mips` and `X86` - but from the structure of the patch it should be fairly simple to move on to any other platform. Hardcoded CPU frequency (again) - bad idea - you need to compile for a specific platform (that at least could be passed as a config option...)

Times are read with low level (hardware dependant but fast) calls to arch specific clocks (i.e. `rdtsc` on `x86`) - rough math to stay in 32bit provided (no `do_gettimeofday` or the like) so this clock is available at a very early stage in the boot process. The records are based on deltas not absolute values (simplifies things as there are no potential overflow issues).

Initial entry is set to 0 so the first value is garbage. This could (should) be trivially fixed by having a hardcoded read of the tsc in `start_kernel` (would help a bit to get boot-times).

Patch does not record any pre- `start_kernel` events (bad) no info on decompression times.

Patch does not personalize the kfi patched kernel - potential collision of modules with unpatched 2.4.20 - `EXTRAVERSION=-kfi` set.

We strongly advice using this at runtime only if core functions are marked with `no_instrument_functions` - brute force instrumentation of each and every function is obviously a performance problem - and actually not that usefull anyway. For boot-time analysis it seems resonable the way static setup is provided (from `start_kernel` to just before `execve /sbin/init`).

The `profile_func_enter/exit` seems quite heavy waitght - might be worth stripping this to the bare minimum possible (something like `func,caller,timestamp` ??- I admittedly don't know exactly what the current implementation does :).

The log entries are quite heavy waigted - the distinction between kernel, interrupt and PID context is quite usefull - its a bit irritating that kenrel threads are not marked any different than processes (TODO: look into adding this in `kfi.c` and in the `kfiresolve.py`).

## 1.4. Basic technology

Function instrumentation is a feature of `gcc` - by compiling applications with the `-finstrument-function` flag each call is preceded and followed by a call to a profiling function.

A simple example of a modified hello world should make this clear easally.

```
#include <stdio.h>

void __attribute__((__no_instrument_function__))
__cyg_profile_func_enter(void *this_fn, void *call_site)
{
    printf("func_enter: function = %p, called by = %p\n",
          this_fn,
          call_site);
}
```

```
void __attribute__((__no_instrument_function__))
__cyg_profile_func_exit(void *this_fn, void *call_site)
{
    printf("func_exit: function = %p, called by = %p\n",
          this_fn,
          call_site);
}

main(){
    printf("hello world\n");
    return 0;
}
```

Compiled with `gcc -finstrument-functions hello.c -o hello` and run as `./hello` we get the output:

```
func_enter: function = 0x8048420, called by = 0x40041936
hello world
func_exit: function = 0x8048420, called by = 0x40041936
```

The instrumentation is done at compile time the normal code:

```
subl    $12, %esp
pushl   $.LC0
call    printf
addl    $16, %esp
```

is surrounded by calls to the profiling enter and exit functions passing the pointer of the caller (`$main`) and the function being called.

```
subl    $8, %esp
pushl   4(%ebp)
pushl   $main
call    __cyg_profile_func_enter
addl    $16, %esp
subl    $12, %esp
pushl   $.LC2
call    printf
addl    $16, %esp
```

```
movl    $0, %ebx
subl    $8, %esp
pushl   4(%ebp)
pushl   $main
call    __cyg_profile_func_exit
addl    $16, %esp
```

Function instrumentation not only is available by coding it directly in the source files, which would be kind of inconvenient, but also can be wrapped up in a library. An example of a library that will produce the same output format for traces that the kernel space implementation is giving (and thus allows to correlate events in kernel and user-space by sorting time-stamps) is given here as an example:

```
/*
 * Compile as shared library with:
 * gcc -fPIC -Wall -g -O2 -shared -o libfunc_profile.so.0 libfunc_profile.c
 *
 * Log all function calls to the logfile in /tmp/func.log (default)
 * the format is chosen to allow usage of kfiresolver.py to reverse the log
 * entries
 */

#include <stdio.h>    /* fprintf */
#include <unistd.h>   /* exit , getpid*/
#include <sys/types.h> /* getpid */
#include <stdlib.h>   /* getenv */

#define _FCNTL_H
#include <bits/fcntl.h>

/* initialize and cleanup logfile(s) on load/unload of lib */
void __func_profile_init(void) __attribute__((constructor));
void __func_profile_exit(void) __attribute__((destructor));

long long int start,last,now;
FILE *logfile;
char default_fname[]="/tmp/func.log";
char *logfile_name;

__inline__ unsigned long long int hwttime(void)
```

```
{
unsigned long long int x;
__asm__ __volatile__("rdtsc\n\t"
:"=A" (x));
return x;
}

void __attribute__((__no_instrument_function__))
__func_profile_init(void)
{
    if ((logfile_name = getenv("PROFILE_LOG")) != 0) {
        printf("using %s\n",logfile_name);
    } else {
logfile_name=default_fname;
        printf("using %s (no PROFILE_LOG set in environment)\n",logfile_name);
    }
}

if((logfile=fopen(logfile_name,"a+")) == NULL )
{
perror("Cannot open logfile\n");
exit(-1);
}

    /* logfile header */
    fprintf(logfile,
" Entry      Delta      PID      Function      Caller\n");
    fprintf(logfile,
"-----      -----      -----      -----      -----\n");

    /* initialize time stamp */
    start=hwttime();
    last=start;
}

void __attribute__((__no_instrument_function__))
__func_profile_exit(void)
{
fclose(logfile);
}

void __attribute__((__no_instrument_function__))
```

```
__cyg_profile_func_enter(void *this_fn, void *call_site)
{
    unsigned long long delta;
    pid_t pid=getpid();
    delta=0LL;

    now=hwttime();
    delta=now-last;
    last=now;

    fprintf(logfile, "%8lu %8lu %7d %08x %08x\n",
        (unsigned long)(now-start),
        (unsigned long)delta,
        pid,
        (unsigned int)this_fn,
        (unsigned int)call_site);
}

void __attribute__((__no_instrument_function__))
__cyg_profile_func_exit(void *this_fn, void *call_site)
{
    unsigned long long delta;
    pid_t pid=getpid();
    delta=0LL;

    now=hwttime();
    delta=now-last;
    last=now;

    fprintf(logfile, "%8lu %8lu %7d %08x %08x\n",
        (unsigned long)(now-start),
        (unsigned long)delta,
        pid,
        (unsigned int)this_fn,
        (unsigned int)call_site);
}
```

With such a library it suffices to recompile user space applications with `-finstrumentn-function -lfunc_profile.so`.

Basically the same sheme is implemented in `driver/char/kfi.c` writing it into a log buffer.

A kernel specific problem that needs to be resolved is that of inline functions. The passing of the address of the caller and the called function is done even for inline functions, resulting in taking the address of the inline functions which is not accessed via call. For functions declared `extern inline` this results in undefined symbol errors. For `static inline` it causes a static version to be compiled in each object file that uses the inline function. So to allow function instrumentation all inline functions are treated as static inline functions - this slows down things even more than would be done just by the overhead of logging data, but this kernel modification is not intended for systems that require high performance.

## 1.5. Installation

Installation of the official patches is more or less broken - patches apply cleanly but compilation fails without quite heavy modifications in the low level code (pre vmlinux stuff).

```
tar -xjf linux-2.4.20.tar.bz2 (kernel.org)
cd linux-2.4.20
patch -p1 < ../kfi-24-test4.patch
make menuconfig
```

```
Kernel hacking ---->
  [ ] Kernel debugging
  [*] Kernel Function Instrumentation (NEW)
  [*] Static Instrumentation Config
```

Note: there is no help available for both of the new options - so here is a minimum summary of what this does.

### Kernel Function Instrumentation

This basically enables kfi (builds the `driver/char/kfi` and inserts `kfi_dump_log` points.

### Static Instrumentation Config

This sets up a linked list as automatic variable instead of dynamically allocating it further more if this option is enable a empty `to_userspace()` function is called before calling `init` - this is used by the static instrumentation config to locate the point where instrumentation should be turned off (so called trigger).

For static instrumentation there is a config file in `drivers/char/kfistatic.conf` that allows setting of instrumentation parameters at compile time (would be nice to have this in the kernel config menu !)

Make procedure of the unmodified patch 2.4.20-test4:

```
make dep
make (fails due to scripts/mkkfirun.pl being mode 644 not 755)
chmod 755 scripts/mkkfirun.pl
make (fails in drivers/ide/ide-cd.h line 440 type : __u8 short -> __u8)
make (compilation completed - further warnings ignored ;)
make modules
make modules_install
make bzImage
```

(fails with undefined references to `__cyg_*...` looks like instrumentation is not implemented correctly - the scope of the entry/exit functions is limited to `vmlinux` (kernel proper) but the low level stuff (`/arch/i386/boot/misc.c` `lib/inflate.c` referenced in `misc.c`) was also compiled with `-finstrument-functions`

Reconfigured without static instrumentation (which would makes it quite unusable for boot-time analysis though) same problem.

Unresolved symbols in `misc.c` - guess `misc.c` should NOT be compiled with `-finstrumentation...` (TODO: need to clarify if this EVER worked with a vanilla 2.4.20 kernel) - workaround turn it off on a per function basis using `__noinstrument` (pain in the but - as you have to recompile the kernel from scratch - which then yields a new set of unresolved symbols ;)

Files cleaned: `lib/inflate.c` `arch/i386/boot/compress/misc.c` - basically all function calls got instrumentation turned off - which is not tragic as `misc.c` is pre `start_kernel` any way and `inflate.c` functions are not called during the system initialization.

## 1.6. Data aquisition

Once it compiles and installs the readme in the `kfi-0.8` tools (`README.kfi`) should be sufficient. The only requirement is `pyhon` -but as data resolution from addresses to names can be done off-line (only need the `System.map` of the profiled system to do it) - `phyton` is no issue.

```
tar -xuf kfi-0.8.tar.gz
cd kfi-0.8
mknod /dev/kfi c 10 51
make
./kfi read 0 > kfiboot.log
vi kfiboot.log
./kfiresolve.py ./kfiboot.log ../linux-2.4.20-kfi/System.map > kfiboot.lst
vi kfiboot.lst
```

Note that in dynamic mode you can only use the read and reset command - all other command will give you IOCTL errors - for dynamic instrumentation logs see the next section. The strange limit of 8092 bytes for static logs seems to stem from the log being on the kernel stack (ugh!)- TODO: fix that .

## 1.7. Dynamic Data acquisition (post boot)

When compiled without static setup it does not work (atleast no out of the box) the command new to kfi will aboard with an EINVAL in IOCTL (NEW\_RUN). Dynamic acquisition looks like its not quite completed yet - the problem with the distributed version is that there are two header files that both define MAX\_RUN\_LOG\_ENTRIES - in kernel space its defined to be 8092 (not 8192) and in user-space kfi.h its set to 20000 - in the IOCTL command switch for NEW\_RUN the passed entry value is checked against kernel side MAX\_RUN\_LOG\_ENTRIES and not to supprising exits with -EINVAL. If set larger than about 6000 (that is clearly below the default 8092 !) it fails due to kmalloc failing (TODO: check and if necessary move to vmalloc)

Once that is fixed it actually kind of works (a bit;)

Hardcode what you want to see in kfi.h (user-space) - the MAX\_RUN\_LOG\_EVENTS must be smaller than the value set in include/linux/kfi.h !

```
make
./kfi reset
./kfi new
new run created, id = 0
./kfi start
runid 0 started
./kfi stop (looks like this is broken !)
STOP ioctl error: Invalid argument
./kfi read > log
-rw-r--r-- 1 root root 217441 2005-01-01 16:56 log
./kfi status
```

Kernel Instrumentation Run ID 0

Logging started at 963768895 usec by system call Logging stopped at 963769264 usec by log full

Filters:

Filter Counters: Total entries filtered = 0 Entries not found = 1

Number of entries after filters = 4096

## 1. Kernel function instrumentation - tool analysis

---

```
./kfi reset
./kfiresolve.py log /usr/src/linux-2.4.20-kfi/System.map > lst
```

lst contains the "call graph" of the booting kernel with timestamps now. The header of the list file is the same as you would get with the ./kfi status command.

Kernel Instrumentation Run ID 0

Logging started at 111102362 usec by system call Logging stopped at 114108365 usec by log full

Filters:

Filter Counters: Total entries filtered = 0 Entries not found = 16

Number of entries after filters = 32768

Entry	Delta	PID	Function	Called At
0	no exit	251	fput	sys_ioctl+0x87
1	no exit	251	do_page_fault	error_code+0x34
1	no exit	251	find_vma	do_page_fault+0x99
1	no exit	251	handle_mm_fault	do_page_fault+0x198
1	no exit	251	pte_alloc	handle_mm_fault+0x54
1	no exit	251	do_no_page	handle_mm_fault+0x7f
1	no exit	251	filemap_nopage	do_no_page+0xa8
1	no exit	251	__find_get_page	filemap_nopage+0xe4

The no exit in the Delta field is due to the delta recorded being 0 - this is the case when the resolution of the kfi\_readclock() is higher than the runtime of the function being traced (TODO: check up on this - looks strange considering they are using the TSC).

```
driver/char/kfi.c:
static inline unsigned long __noinstrument
update_usecs_since_boot(void)
{
    unsigned long machine_cycles, delta;

    machine_cycles = kfi_readclock();
    delta = machine_cycles - last_machine_cycles;
    delta = kfi_clock_to_usecs(delta);
}
```

```
    usecs_since_boot += delta;

    last_machine_cycles = machine_cycles;
    return usecs_since_boot;
}
```

To help interpret data there is a tool in the `kfi-0.8.tar.gz` archive that allows filtering the `lst` data:

```
./kd -h
usage: kd [<options>] <filename>
```

This program parses the output from a set of `kfi` message lines

Options:

```
-h          Show this usage help.
-c <count> Only show the <count> most time-consuming functions
-t <time>  Only show functions with time greater than <time>
-f <format> Show columns indicated by <format> string. Column IDs
           are single characters, with the following meaning:
           F = Function name
           c = Count (number of times function was called)
           t = Time (total time spent in this function)
           a = Average (average time per function call)
           r = Range (minimum and maximum times for a single call)
           s = Sub-time (time spent in sub-routines)
           l = Local time (time not spent in sub-routines)
           m = Max sub-routine (name of sub-routine with max time)
           n = Max sub-routine count (# of times max sub-routine
           was called)
           u = Sub-routine list (this feature is experimental)
           The default column format string is "Fctal"
-l          Show long listing (default format string is "Fctalsmn")
```

i.e. `./kd -c 10 lst` will show the 10 most heavy weight functions being called in the trace.

Function	Count	Time	Average	Local
-----	-----	-----	-----	-----
schedule	278	6010296	21619	6010296

## 1. Kernel function instrumentation - tool analysis

---

schedule_timeout	139	6008929	43229	45
sys_select	169	3005012	17781	26
do_select	169	3004985	17780	36
sys_read	140	3004086	21457	21
tty_read	129	3004050	23287	21
read_chan	129	3004029	23287	30
default_idle	421	3003589	7134	3003589
do_IRQ	429	2657	6	0
handle_IRQ_event	429	1654	3	0

kmalloc switch to bigphysarea:

To improve dynamic tracing kfi was modified to use bigphysarea instead of malloc (which is limited to the inamous 128k) - unfortunately! this does not allow tracing from start\_kernel on bug the startup must be done later:

0	no exit	0	start_kernel	L6+0x0
1	9420	0	setup_arch	start_kernel+0x30
208	9110	0	paging_init	setup_arch+0x1cc
222	9096	0	zone_sizes_init	paging_init+0x42
222	9096	0	free_area_init	zone_sizes_init+0x43
222	9096	0	free_area_init_core	free_area_init+0x4f
222	4474	0	__alloc_bootmem_node	free_area_init_core+0x3b7
222	4474	0	__alloc_bootmem_core	__alloc_bootmem_node+0x49
9421	14609	0	parse_options	start_kernel+0x51
9424	14606	0	checksetup	parse_options+0x18f
9424	14606	0	bigphysarea_setup	checksetup+0xaa
9424	14605	0	__alloc_bootmem	bigphysarea_setup+0x79
9424	14605	0	__alloc_bootmem_core	__alloc_bootmem+0x4e
24067	377850	0	time_init	start_kernel+0x6a

Until after \_\_alloc\_bootmem\_core bigphysarea\_malloc will cause a system lockup/reboot - if bigphysarea should be used to get post-init traces the trigger point would need to be set to timer\_init() instead of boot\_kernel.

For dynamic traces the limit of the log length is limited by physical ram that can be allocated to bigphysarea via kernel parameter bigphysarea=NUMBER\_OF\_PAGES, note that the trace structure entries are 256bytes so 1024 pages would result in a limit of 131072 (128k entries).

### 1.8. Data interpretation

see `kfiboot.log` and `kfiboot.lst` - basically simply run down the list and search for functions that take a long time - there also are structural issues one can find - like heavy invocation of delays. TODO: cleanup.

### 1.9. Performanc Impact

The performance in pact of `kfi` is substantial - it is a diagnostic tool and no runtime debug tool - insofar it is inferior to tools like `lvt`- but for boot-time issues it is superior to instrumented `printk` or `lvt` (the later launches much to late to be of much use for boot-time issues).

Performance impact is run by comparing `lmbench` on an unpatched 2.4.20 kernel with `kfi` being used at turntime (not boot-time - basically because we can't run `lmbench` at boot time).

## 2. Conclusion

KFI is a tool usable for test-runs - its absolute values are not that usable but the relativ values are very usable. The granularity of the results is sufficient to pinpoint potential optimizations very quickly.

A drawback is that it does not trace the low level initialization code befor `start_kernel` (this does not seem to be a config issue - but due to the low level stuff requireing to be compiled without instrumentation (atleast we did not manage to do it without `_noinstrument` in the low level stuff)).

The state of the project is not yet stable - documentation is still incopmplete and the patches don't seem to be clean (yet) - the project is of interest but only can be recomended for use if a local technician sufficiently understands the technology to actually maintain it.

Although only `arm`, `mips` and `x86` are supported porting to further archs should be quite simply as the core of `kfi` is well encapsulated in `drivers/char/kfi.c` .

A clear disadvantage at this point is that the trace configuration for boot-time traces is statically configured and requires recompiling when changed.

`kfi`'s advantage over i.e. `lvt` is that it is non-invasive - the driver is capsulated and the function tracing is done via `gcc`'s build in instrumentation capabilities - thus adding and expanding to new kernel routines or custom drivers is trivial - futhermore the trace is flat with respect to covered functions (if one excludes the issues of inlined funcs) as compared to `lvt` where

## 2. Conclusion

---

the trace points are designed by the developers (although ltt does allow adding custom trace points - its just not automatic). The clear drawback is its impactt on the system - kfi is an analysis tool but no runtime-debug tool like ltt.

kfi is a non-standard tool, although standardization is not a madatory feature it would help with comparing results from other tools.

It also should be noted that the development is still in an early stage and that currently the risk of relying on kfi may be fairly high - if a project wishes to utilize kfi it is recomended that at least one team member actually work into it to a level where a self-sustained maintenance and contiuation is posible.

Dynamic tracing is close to being called brocken - but it should not be too hard to fix it (see above).

KFI is a very interesting technology and hopfully will be continued - its current state is usable with a bit of tuning but it is currently at best beta.

### **3. List of Acronyms**

CVS - Concurrent Version Control

GNU - GNU Not UNIX (recursive acronym)

KFI - Kernel Function Instrumentation

GCC - GNU C Compiler

LTT - Linux Trace Toolkit

## **References**

- [1] - Embedded Linux Kickstart Session, <http://www.opentech.at/documents.html>, 2004.