function tracer guts

By Mike Frysinger

Introduction

Here we will cover the architecture pieces that the common function tracing code relies on for proper functioning. Things are broken down into increasing complexity so that you can start simple and at least get basic functionality.

Note that this focuses on architecture implementation details only. If you want more explanation of a feature in terms of common code, review the common ftrace.txt file.

Prerequisites

Ftrace relies on these features being implemented:
 STACKTRACE_SUPPORT - implement save_stack_trace()
 TRACE_IRQFLAGS_SUPPORT - implement include/asm/irqflags.h

HAVE_FUNCTION_TRACER

You will need to implement the mcount and the ftrace_stub functions.

The exact mcount symbol name will depend on your toolchain. Some call it "mcount", "_mcount", or even "__mcount". You can probably figure it out by running something like:

```
        $ echo 'main(){}' | gcc -x c -S -o - - -pg | grep mcount
                call    mcount
```

We'll make the assumption below that the symbol is "mcount" just to keep things nice and simple in the examples.

Keep in mind that the ABI that is in effect inside of the mcount function is *highly* architecture/toolchain specific. We cannot help you in this regard, sorry. Dig up some old documentation and/or find someone more familiar than you to bang ideas off of. Typically, register usage (argument/scratch/etc...) is a major issue at this point, especially in relation to the location of the mcount call (before/after function prologue). You might also want to look at how glibc has implemented the mcount function for your architecture. It might be (semi-)relevant.

The mcount function should check the function pointer ftrace_trace_function to see if it is set to ftrace_stub. If it is, there is nothing for you to do, so return immediately. If it isn't, then call that function in the same way the mcount function normally calls __mcount_internal -- the first argument is the "frompc" while the second argument is the "selfpc" (adjusted to remove the size of the mcount call that is embedded in the function).

For example, if the function foo() calls bar(), when the bar() function calls mcount(), the arguments mcount() will pass to the tracer are:

        "frompc" - the address bar() will use to return to foo()

"selfpc" - the address bar() (with mcount() size adjustment)

Also keep in mind that this mcount function will be called *a lot*, so optimizing for the default case of no tracer will help the smooth running of your system when tracing is disabled. So the start of the mcount function is typically the bare minimum with checking things before returning. That also means the code flow should usually be kept linear (i.e. no branching in the nop case). This is of course an optimization and not a hard requirement.

Here is some pseudo code that should help (these functions should actually be implemented in assembly):

```
void ftrace_stub(void)
{
        return;
}

void mcount(void)
{
        /* save any bare state needed in order to do initial checking */

        extern void (*ftrace_trace_function)(unsigned long, unsigned long);
        if (ftrace_trace_function != ftrace_stub)
                goto do_trace;

        /* restore any bare state */

        return;

do_trace:

        /* save all state needed by the ABI (see paragraph above) */

        unsigned long frompc = ...;
        unsigned long selfpc = <return address> - MCOUNT_INSN_SIZE;
        ftrace_trace_function(frompc, selfpc);

        /* restore all state needed by the ABI */
}
```

Don't forget to export mcount for modules !

```
extern void mcount(void);
EXPORT_SYMBOL(mcount);
```

HAVE_FUNCTION_TRACE_MCOUNT_TEST

This is an optional optimization for the normal case when tracing is turned off in the system. If you do not enable this Kconfig option, the common ftrace code will take care of doing the checking for you.

To support this feature, you only need to check the function_trace_stop variable in the mcount function. If it is non-zero, there is no tracing to be done at all, so you can return.

This additional pseudo code would simply be:

```
void mcount(void)
{
        /* save any bare state needed in order to do initial checking */

+       if (function_trace_stop)
+               return;

        extern void (*ftrace_trace_function)(unsigned long, unsigned long);
        if (ftrace_trace_function != ftrace_stub)
...
```

HAVE_FUNCTION_GRAPH_TRACER

Deep breath ... time to do some real work. Here you will need to update the mcount function to check ftrace graph function pointers, as well as implement some functions to save (hijack) and restore the return address.

The mcount function should check the function pointers ftrace_graph_return (compare to ftrace_stub) and ftrace_graph_entry (compare to ftrace_graph_entry_stub). If either of those is not set to the relevant stub function, call the arch-specific function ftrace_graph_caller which in turn calls the arch-specific function prepare_ftrace_return. Neither of these function names is strictly required, but you should use them anyway to stay consistent across the architecture ports -- easier to compare & contrast things.

The arguments to prepare_ftrace_return are slightly different than what are passed to ftrace_trace_function. The second argument "selfpc" is the same, but the first argument should be a pointer to the "frompc". Typically this is located on the stack. This allows the function to hijack the return address temporarily to have it point to the arch-specific function return_to_handler. That function will simply call the common ftrace_return_to_handler function and that will return the original return address with which you can return to the original call site.

Here is the updated mcount pseudo code:

```
void mcount(void)
{
...
        if (ftrace_trace_function != ftrace_stub)
                goto do_trace;

+#ifdef CONFIG_FUNCTION_GRAPH_TRACER
+       extern void (*ftrace_graph_return)(...);
+       extern void (*ftrace_graph_entry)(...);
+       if (ftrace_graph_return != ftrace_stub ||
+           ftrace_graph_entry != ftrace_graph_entry_stub)
+               ftrace_graph_caller();
+#endif

        /* restore any bare state */
...
```

Here is the pseudo code for the new ftrace_graph_caller assembly function:

```
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
void ftrace_graph_caller(void)
{
        /* save all state needed by the ABI */

        unsigned long *frompc = &...;
        unsigned long selfpc = <return address> - MCOUNT_INSN_SIZE;
        /* passing frame pointer up is optional -- see below */
        prepare_ftrace_return(frompc, selfpc, frame_pointer);

        /* restore all state needed by the ABI */
}
#endif
```

For information on how to implement prepare_ftrace_return(), simply look at the x86 version (the frame pointer passing is optional; see the next section for more information). The only architecture-specific piece in it is the setup of the fault recovery table (the asm(...) code). The rest should be the same across architectures.

Here is the pseudo code for the new return_to_handler assembly function. Note that the ABI that applies here is different from what applies to the mcount code. Since you are returning from a function (after the epilogue), you might be able to skimp on things saved/restored (usually just registers used to pass return values).

```
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
void return_to_handler(void)
{
        /* save all state needed by the ABI (see paragraph above) */

        void (*original_return_point)(void) = ftrace_return_to_handler();

        /* restore all state needed by the ABI */

        /* this is usually either a return or a jump */
        original_return_point();
}
#endif
```

HAVE_FUNCTION_GRAPH_FP_TEST

An arch may pass in a unique value (frame pointer) to both the entering and exiting of a function. On exit, the value is compared and if it does not match, then it will panic the kernel. This is largely a sanity check for bad code generation with gcc. If gcc for your port sanely updates the frame pointer under different opitmization levels, then ignore this option.

However, adding support for it isn't terribly difficult. In your assembly code that calls prepare_ftrace_return(), pass the frame pointer as the 3rd argument. Then in the C version of that function, do what the x86 port does and pass it along to ftrace_push_return_trace() instead of a stub value of 0.

Similarly, when you call ftrace_return_to_handler(), pass it the frame pointer.

HAVE_FTRACE_NMI_ENTER

If you can't trace NMI functions, then skip this option.

<details to be filled>

HAVE_SYSCALL_TRACEPOINTS

You need very few things to get the syscalls tracing in an arch.

- Support HAVE_ARCH_TRACEHOOK (see arch/Kconfig).
- Have a NR_syscalls variable in <asm/unistd.h> that provides the number
  of syscalls supported by the arch.
- Support the TIF_SYSCALL_TRACEPOINT thread flags.
- Put the trace_sys_enter() and trace_sys_exit() tracepoints calls from ptrace
  in the ptrace syscalls tracing path.
- Tag this arch as HAVE_SYSCALL_TRACEPOINTS.

HAVE_FTRACE_MCOUNT_RECORD

See scripts/recordmcount.pl for more info.

<details to be filled>

HAVE_DYNAMIC_FTRACE

<details to be filled>