Function Tracer Design

Author: Mike Frysinger

> **Caution!**
> This document is out of date. Some of the description below doesn't match current implementation now.

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.

Ideally, everyone who wishes to retain performance while supporting tracing in their kernel should make it all the way to dynamic ftrace support.

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_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 optimization 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_FUNCTION_GRAPH_RET_ADDR_PTR

An arch may pass in a pointer to the return address on the stack. This prevents potential stack unwinding issues where the unwinder gets out of sync with ret_stack and the wrong addresses are reported by ftrace_graph_ret_addr().

Adding support for it is easy: just define the macro in asm/ftrace.h and pass the return address pointer as the 'retp' argument to ftrace_push_return_trace().

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.
- If the system call table on this arch is more complicated than a simple array of addresses of the system calls, implement an arch_syscall_addr to return the address of a given system call.
- If the symbol names of the system calls do not match the function names on this arch, define ARCH_HAS_SYSCALL_MATCH_SYM_NAME in asm/ftrace.h and implement arch_syscall_match_sym_name with the appropriate logic to return true if the function name corresponds with the symbol name.
- Tag this arch as HAVE_SYSCALL_TRACEPOINTS.

HAVE_FTRACE_MCOUNT_RECORD

See scripts/recordmcount.pl for more info. Just fill in the arch-specific details for how to locate the addresses of mcount call sites via objdump. This option doesn't make much sense without also implementing dynamic ftrace.

HAVE_DYNAMIC_FTRACE

You will first need HAVE_FTRACE_MCOUNT_RECORD and HAVE_FUNCTION_TRACER, so scroll your reader back up if you got over eager.

Once those are out of the way, you will need to implement:

- asm/ftrace.h:
    - MCOUNT_ADDR
    - ftrace_call_adjust()
    - struct dyn_arch_ftrace{}
- asm code:
    - mcount() (new stub)

        - ftrace_caller()
        - ftrace_call()
        - ftrace_stub()
    - C code:
        - ftrace_dyn_arch_init()
        - ftrace_make_nop()
        - ftrace_make_call()
        - ftrace_update_ftrace_func()

First you will need to fill out some arch details in your asm/ftrace.h.

Define MCOUNT_ADDR as the address of your mcount symbol similar to:

```
#define MCOUNT_ADDR ((unsigned long)mcount)
```

Since no one else will have a decl for that function, you will need to:

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

You will also need the helper function ftrace_call_adjust(). Most people will be able to stub it out like so:

```
static inline unsigned long ftrace_call_adjust(unsigned long addr)
{
        return addr;
}
```

<details to be filled>

Lastly you will need the custom dyn_arch_ftrace structure. If you need some extra state when runtime patching arbitrary call sites, this is the place. For now though, create an empty struct:

```
struct dyn_arch_ftrace {
        /* No extra data needed */
};
```

With the header out of the way, we can fill out the assembly code. While we did already create a mcount() function earlier, dynamic ftrace only wants a stub function. This is because the mcount() will only be used during boot and then all references to it will be patched out never to return. Instead, the guts of the old mcount() will be used to create a new ftrace_caller() function. Because the two are hard to merge, it will most likely be a lot easier to have two separate definitions split up by #ifdefs. Same goes for the ftrace_stub() as that will now be inlined in ftrace_caller().

Before we get confused anymore, let's check out some pseudo code so you can implement your own stuff in assembly:

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

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

        unsigned long frompc = ...;
        unsigned long selfpc = <return address> - MCOUNT_INSN_SIZE;

ftrace_call:
        ftrace_stub(frompc, selfpc);

        /* restore all state needed by the ABI */

ftrace_stub:
        return;
}
```

This might look a little odd at first, but keep in mind that we will be runtime patching multiple things. First, only functions that we actually want to trace will be patched to call ftrace_caller(). Second, since we only have one tracer active at a time, we will patch the ftrace_caller() function itself to call the specific tracer in question. That is the point of the ftrace_call label.

With that in mind, let's move on to the C code that will actually be doing the runtime patching. You'll need a little knowledge of your arch's opcodes in order to make it through the next section.

Every arch has an init callback function. If you need to do something early on to initialize some state, this is the time to do that. Otherwise, this simple function below should be sufficient for most people:

```
int __init ftrace_dyn_arch_init(void)
{
        return 0;
}
```

There are two functions that are used to do runtime patching of arbitrary functions. The first is used to turn the mcount call site into a nop (which is what helps us retain runtime performance when not tracing). The second is used to turn the mcount call site into a call to an arbitrary location (but typically that is ftracer_caller()). See the general function definition in linux/ftrace.h for the functions:

```
ftrace_make_nop()
ftrace_make_call()
```

The rec->ip value is the address of the mcount call site that was collected by the scripts/recordmcount.pl during build time.

The last function is used to do runtime patching of the active tracer. This will be modifying the assembly code at the location of the ftrace_call symbol inside of the ftrace_caller() function. So you should have sufficient padding at that location to support the new function calls you'll be inserting. Some people will be using a "call" type instruction while others will be using a "branch" type instruction. Specifically, the function is:

```
ftrace_update_ftrace_func()
```

HAVE_DYNAMIC_FTRACE + HAVE_FUNCTION_GRAPH_TRACER

The function grapher needs a few tweaks in order to work with dynamic ftrace. Basically, you will need to:

- update:
    - ftrace_caller()
    - ftrace_graph_call()
    - ftrace_graph_caller()
- implement:
    - ftrace_enable_ftrace_graph_caller()
    - ftrace_disable_ftrace_graph_caller()

<details to be filled>

Quick notes:

- add a nop stub after the ftrace_call location named ftrace_graph_call; stub needs to be large enough to support a call to ftrace_graph_caller()
- update ftrace_graph_caller() to work with being called by the new ftrace_caller() since some semantics may have changed
- ftrace_enable_ftrace_graph_caller() will runtime patch the ftrace_graph_call location with a call to ftrace_graph_caller()
- ftrace_disable_ftrace_graph_caller() will runtime patch the ftrace_graph_call location with nops