1 General information

Name: Marko Vlaić, email: marko.vlaic@fer.hr, mobile: +385916037618

1.1 Possible mentor

Bojan Novković

2 Project information

2.1 Project title

Zero-cost conditional execution mechanism

2.2 Project description

It is common for kernel subsystems to conditionally include functionality, based on compile time and runtime configurations which are relatively infrequently subject to change. Typical examples include: toggling of DTrace probes, boot-time optimizations based on hardware capabilities and inclusion of additional security checks. This is often done by examining the state of a global flag and executing a block of code conditionally based on that state. When this is done in a "hot" (i.e. frequently executed) code path, the overhead of the conditional execution can become significant. Moreover, in situations in which the state of the inspected flags changes rarely, most of the performed checks are redundant.

The goal of this project is to design and implement a low overhead mechanism for conditional execution in contexts in which the branching condition does not change often. The main idea is to replace the check of a flag containing a `false` value with a machine `nop` instruction. When the flag value is toggled to `true` the `nop` instruction is patched at runtime to an unconditional `jump` instruction, changing the execution flow to the conditionally executed block. An inverse procedure is performed when the flag value is toggled back to `false`. This way we make branch selection as cheap as we can, but in return we sacrifice additional time when changing the state of the flag.
Another consideration to make is that this kind of kernel code patching at runtime can have a negative impact on security and correctness, in an SMP environment. One processor trying to execute an instruction which is simultaneously being patched by another processor could cause a wrong or invalid instruction to be executed.
As the mechanism is highly architecture dependent, this project will aim to develop a strong implementation on the x86-64 architecture. Once the x86-64 implementation proves to be solid ports to other architectures will be made. When invoked on non-supported architectures, the mechanism will fall back to the traditional conditional branching approach.

The final goal of the project is to thoroughly test and benchmark the mechanism. First, an artificial benchmark scenario will be created (alternatively an existing one may be selected). The last step will be to apply the mechanism to an already existing portion of kernel code. This will be done for the purposes of demonstration as well as benchmarking in a realistic scenario.

A similar feature exists in the Linux kernel, and is accessible through the `static_keys` api[1], where it proved to have a measurable impact on performance. This will serve as a good source of inspiration, when it comes to implementation details.

2.3 Deliverables

1. A patch implementing the code patching and static branching mechanism for the x86-64 architecture.
2. A patch which applies the developed mechanism to an existing piece of kernel code.

2.4 Tentative implementation plan

2.4.1 Interface description

The proposed mechanism aims to improve performance of code blocks of form shown in Listing 1, with minimal intervention into existing code.

```
if(flag) {
    <condition_true_body>
}
<rest of the function>
```

Listing 1: Target code form

We propose an interface consisting of the elements listed in Table 1.

With these in mind the code in Listing 1 can adopt the new mechanism with little effort, as shown in Listing 2.

```
// somewhere in the global scope
DEFINE_STATIC_FLAG_TRUE(flag);
...
if(static_flag_true(&flag)) {
    <condition_true_body>
}
```

Listing 2: Converted code

Element	Description
`struct static_flag`	A structure which holds the state and ancillary data of a single flag which supports the low cost branching mechanism
`DEFINE_STATIC_FLAG_TRUE(fname)`	A macro which declares and initializes `struct static_flag fname` with a `true` value
`DEFINE_STATIC_FLAG_FALSE(fname)`	A macro which declares and initializes `struct static_flag fname` with a `false` value
`bool static_flag_true(struct static_flag* fp)`	A function which returns true if the flag pointed to by `fp` is set to `true`
`bool static_flag_false(struct static_flag* fp)`	A function which returns true if the flag pointed to by `fp` is set to `false`
`void static_flag_enable(struct static_flag* fp)`	A function which sets the state of the flag pointed to by `fp` to `true` and code patches the appropriate instructions accordingly
`void static_flag_disable(struct static_flag* fp)`	A function which sets the state of the flag pointed to by `fp` to `false` and code patches the appropriate instructions accordingly

Table 1: Elements of the static_flags interface

2.4.2 Static branch selection

The code listed in Listing 1 can be expected to compile to a sequence of machine code instructions similar to the one shown in Listing 3 (on an x86-64 platform).

```
    ...
    mov <flag>, %eax
    test %eax, %eax
    je <false_label>
    <condition_true_body instructions>
false_label:
    <rest of the instructions>
```

Listing 3: Target machine code

A single block of this kind introduces negligible overhead, in the general case. However when found in a "hot" code path, when the memory caches are under pressure or when there is a large number of these kind of checks the performance penalty they bring can become (somewhat) significant.
To reduce this cost we choose the branch direction statically. To begin with, assume the value of `flag` is initially set to `true`. We choose the `true` branch by transforming the instruction sequence in Listing 3 to the one in Listing 4.

```
    ...
    nop
    <condition_true_body instructions>
false_label:
    <rest of the instructions>
```

Listing 4: Statically selected true branch

The instructions dedicated to loading the flag from memory and checking its state are replaced by a single `nop` instruction. This saves one memory access and the memory required to store two instructions.
Later on, when the state of the flag changes to `false`, the instruction sequence in Listing 4 gets patched to the one in Listing 5.

```
    ...
    jmp <false_label>
    <condition_true_body instructions>
false_label:
    <rest of the instructions>
```

Listing 5: Statically selected false branch

Here the `nop` instruction got patched to an unconditional jump.
The assembly code of the actual implementation might differ slightly from the one displayed for the purposes of this discussion, in order to support the interface as defined in Section 2.4.1 (mainly because of the return types of functions `static_flag_true` and `static_flag_false` being `bool`); still no memory access will be required.

2.4.3 Storing locations to be patched

A single `struct static_flag` can, in general, have an arbitrary number of branches inspecting its state to determine the branch direction. We will name code locations at which this occurs inspection points. In order to keep track of all inspection points associated with a flag, a new ELF section, named `__jump_table` will be created. Each entry in the `__jump_table` describes a single inspection point. An inspection point entry holds the following data: address of the instruction to be patched, address of the label to jump to when the `true` branch is not selected and a pointer to the `struct static_flag` the entry is associated with.
To make this possible, we will leverage a gcc feature enabling the `asm goto` statement[2], introduced specifically for use cases of this type. The `asm goto` statement lets inline assembly statements reference C labels. Listing 6 shows a snippet which adds an entry to the `__jump_table`, where the instruction to be patched is a `nop`. Variations of this snippet would be placed into `static_flag_true` and `static_flag_false` functions to add appropriate inspection points when they are invoked.

```
    asm goto (
        "1:"
        "nop\n\t"
        ".pushsection __jump_table,  \"aw\" \n\t"
        ".long 1b \n\t"
        ".long %l[label_false] \n\t"
        ".long %c0 \n\t"
        :: "i" (flag_ptr) :: label_false);
```

Listing 6: Adds inspection point to the jump table

Finally, the `__jump_table` section will need to be added to the appropriate `/sys/conf/ldscript.amd64` linker script.

2.4.4 Patching the code

On kernel, or module, load each defined `struct static_flag` instance will be populated with a list of its associated inspection points. When the state of a `struct static_flag` instance is toggled via the `static_flag_enable` and `static_flag_disable` functions, each of those inspection points will need to be patched accordingly.
To perform the patching safely, we will have to make sure the processor performing the patch has exclusive access to the page in which the instruction is located; otherwise we risk many processor accessing the changing data, one patching it and one trying to execute it for example, which could result in all sorts of bad behaviour. Possible ways of granting exclusive access would be some sort of locking or temporarily stopping execution on other CPUs by dispatching inter-processor interrupts.

2.4.5 Testing the mechanism

Once developed, the mechanism will first have to be tested for correctness. A test suite will be developed in order to ensure the behaviour of code with and without the mechanism is the same, in as many scenarios as possible.
The code will then have to be benchmarked to measure the performance gain. The first step will be to develop an artificial benchmark, full of code blocks resembling the one in Listing 1. This will give us the idea of a possible speedup, in an ideal scenario.
The last step will be to find an appropriate patch of kernel code in which the mechanism will then be applied. This way we will be able to see if there is any real benefit of introducing this interface into existing kernel code.

2.5 Project schedule

If possible, I would like to start working ahead of the official schedule, to compensate for the time lost during my final exam season (June 17. - July 1.). If this is not possible, I would like to apply for the extended coding period.

2.5.1 May 6. - May 12.

- Set up the development environment
- Revise the implementation plan with my mentor
- Consider the interface from a user point of view and make any needed changes
- Think about the way to split code into high-level and architecture-specific

2.5.2 May 13. - May 19.

- Start working on the static branch selection
- Implement the `DEFINE_STATIC_FLAG()` macro
- Develop a proof of concept code block with `asm goto`

2.5.3 May 20. - May 26.

- Continue working on the static branch selection
- Implement the `static_flag_true()` and `static_flag_false()` functions

2.5.4 May 27. - June 2.

- Finish work on the static branch selection
- Perform some rudimentary testing
- Begin researching the mechanisms needed to support code patching

2.5.5 June 3. - June 9.

- Start working on the code patching
- Start work on the single-processor versions of `void static_flag_enable()` and `static_flag_disable()` functions.

2.5.6 June 10. - June 16.

I would like to take this week to focus on studying for my exams.

2.5.7 June 17. - June 30.

Exam season

2.5.8 July 1. - July 7.

- finish working on the single-processor versions of `void static_flag_enable()` and `static_flag_disable()` functions.
- start researching the locking mechansims needed for an SMP version of the code patching mechansim

2.5.9 July 8. - July 14.

- Implement the SMP version of the code patching mechanism

2.5.10 July 15. - July 21.

- Implement the SMP version of the code patching mechanism

2.5.11 July 22. - July 28.

- Work out any remaining issues
- Benchmark the mechanism with an artificial test suite
- Start looking for kernel code suitable to be refactored to use the new interface

2.5.12 July 29. - August 4.

- Study the existing kernel code selected for refactoring
- Rewrite the selected piece of the kernel

2.5.13 August 5. - August 11.

- Final code review with my mentor
- Code polishing and squashing any unforseen bugs

2.5.14 August 12. - August 25.

Two buffer weeks included in case any unexpected delays occur

2.6 Biography

I am a first year M.Sc. student at the Faculty of Electrical Engineering and Computing in Zagreb, Croatia. I am currently trying to focus my attention on systems programming and embedded systems. Unfortunately, I have little experience with operating systems development and FreeBSD, apart from the courses offered at my faculty. I would very much like that to chnage. I feel comfortable reading and writing C code. I had little trouble reading the code of some FreeBSD subsystems. I was also able to write some practice device drivers.

When it comes to relevant work experience, I worked at Ericsson Nikola Tesla for about half a year as a C++ embedded developer.

2.7 Availability

I estimate that I will be able to set aside 20 hours a week in the weeks before the final exam season at my faculty. With the beginning of July i plan on spending about 40 hours per week until the end of the project. I will be available for the whole of July, August and September if it is found to be necessary. I believe that I will be able to finish the project inside of the planned schedule.

2.8 Future work (post-GSoC)

The goal of this project is to develop a foundation for a useful kernel utility. If the benchmarks show performance improvements I would very much like to continue working on the mechanism; the most important part of future work being bringing support for other architectures. Furthermore, the scope of this project will most likely not include support for the use of the mechanism in `kld` modules. This is something i plan on changing, provided the mechanism is proved useful.

3 References

References

[1] https://www.kernel.org/doc/Documentation/static-keys.txt

[2] https://gcc.gnu.org/legacy-ml/gcc-patches/2009-07/msg01556.html