Static Keys

Warning

DEPRECATED API:

The use of 'struct static_key' directly, is now DEPRECATED. In addition static_key_{true,false}() is also DEPRECATED. IE DO NOT use the following:

```
struct static_key false = STATIC_KEY_INIT_FALSE;
struct static_key true = STATIC_KEY_INIT_TRUE;
static_key_true()
static_key_false()
```

The updated API replacements are:

```
DEFINE_STATIC_KEY_TRUE(key);
DEFINE_STATIC_KEY_FALSE(key);
DEFINE_STATIC_KEY_ARRAY_TRUE(keys, count);
DEFINE_STATIC_KEY_ARRAY_FALSE(keys, count);
static_branch_likely()
static_branch_unlikely()
```

Abstract

Static keys allows the inclusion of seldom used features in performance-sensitive fast-path kernel code, via a GCC feature and a code patching technique. A quick example:

```
DEFINE_STATIC_KEY_FALSE(key);

...

if (static_branch_unlikely(&key))
        do unlikely code
else
        do likely code

...
static_branch_enable(&key);
...
static_branch_disable(&key);
...
```

The static_branch_unlikely() branch will be generated into the code with as little impact to the likely code path as possible.

Motivation

Currently, tracepoints are implemented using a conditional branch. The conditional check requires checking a global variable for each tracepoint. Although the overhead of this check is small, it increases when the memory cache comes under pressure (memory cache lines for these global variables may be shared with other memory accesses). As we increase the number of tracepoints in the kernel this overhead may become more of an issue. In addition, tracepoints are often dormant (disabled) and provide no direct kernel functionality. Thus, it is highly desirable to reduce their impact as much as possible. Although tracepoints are the original motivation for this work, other kernel code paths should be able to make use of the static keys facility.

Solution

gcc (v4.5) adds a new 'asm goto' statement that allows branching to a label:

https://gcc.gnu.org/ml/gcc-patches/2009-07/msg01556.html

Using the 'asm goto', we can create branches that are either taken or not taken by default, without the need to check memory. Then, at run-time, we can patch the branch site to change the branch direction.

For example, if we have a simple branch that is disabled by default:

```
if (static_branch_unlikely(&key))
        printk("I am the true branch\n");
```

Thus, by default the 'printk' will not be emitted. And the code generated will consist of a single atomic 'no-op' instruction (5 bytes on x86), in the straight-line code path. When the branch is 'flipped', we will patch the 'no-op' in the straight-line codepath with a 'jump' instruction to the out-of-line true branch. Thus, changing branch direction is expensive but branch selection is basically 'free'. That is the basic tradeoff of this optimization.

This lowlevel patching mechanism is called 'jump label patching', and it gives the basis for the static keys facility.

Static key label API, usage and examples

In order to make use of this optimization you must first define a key:

```
DEFINE_STATIC_KEY_TRUE(key);
```

or:

```
DEFINE_STATIC_KEY_FALSE(key);
```

The key must be global, that is, it can't be allocated on the stack or dynamically allocated at run-time.

The key is then used in code as:

```
if (static_branch_unlikely(&key))
        do unlikely code
else
        do likely code
```

Or:

```
if (static_branch_likely(&key))
        do likely code
else
        do unlikely code
```

Keys defined via DEFINE_STATIC_KEY_TRUE(), or DEFINE_STATIC_KEY_FALSE, may be used in either static_branch_likely() or static_branch_unlikely() statements.

Branch(es) can be set true via:

```
static_branch_enable(&key);
```

or false via:

```
static_branch_disable(&key);
```

The branch(es) can then be switched via reference counts:

```
static_branch_inc(&key);
...
static_branch_dec(&key);
```

Thus, 'static_branch_inc()' means 'make the branch true', and 'static_branch_dec()' means 'make the branch false' with appropriate reference counting. For example, if the key is initialized true, a static_branch_dec(), will switch the branch to false. And a subsequent static_branch_inc(), will change the branch back to true. Likewise, if the key is initialized false, a 'static_branch_inc()', will change the branch to true. And then a 'static_branch_dec()', will again make the branch false.

The state and the reference count can be retrieved with 'static_key_enabled()' and 'static_key_count()'. In general, if you use these functions, they should be protected with the same mutex used around the enable/disable or increment/decrement function.

Note that switching branches results in some locks being taken, particularly the CPU hotplug lock (in order to avoid races against CPUs being brought in the kernel while the kernel is getting patched). Calling the static key API from within a hotplug notifier is thus a sure deadlock recipe. In order to still allow use of the functionality, the following functions are provided:

> static_key_enable_cpuslocked() static_key_disable_cpuslocked() static_branch_enable_cpuslocked() static_branch_disable_cpuslocked()

These functions are *not* general purpose, and must only be used when you really know that you're in the above context, and no other.

Where an array of keys is required, it can be defined as:

```
DEFINE_STATIC_KEY_ARRAY_TRUE(keys, count);
```

or:

```
DEFINE_STATIC_KEY_ARRAY_FALSE(keys, count);
```

4. Architecture level code patching interface, 'jump labels'

There are a few functions and macros that architectures must implement in order to take advantage of this optimization. If there is no architecture support, we simply fall back to a traditional, load, test, and jump sequence. Also, the struct jump_entry table must be at least 4-byte aligned because the static_key->entry field makes use of the two least significant bits.

- `select HAVE_ARCH_JUMP_LABEL`,
  see: arch/x86/Kconfig
- `#define JUMP_LABEL_NOP_SIZE`,
  see: arch/x86/include/asm/jump_label.h
- `__always_inline bool arch_static_branch(struct static_key *key, bool branch)`,
  see: arch/x86/include/asm/jump_label.h

- `__always_inline bool arch_static_branch_jump(struct static_key *key, bool branch),`
  see: arch/x86/include/asm/jump_label.h
- `void arch_jump_label_transform(struct jump_entry *entry, enum jump_label_type type),`
  see: arch/x86/kernel/jump_label.c
- `__init_or_module void arch_jump_label_transform_static(struct jump_entry *entry, enum jump_label_type type),`
  see: arch/x86/kernel/jump_label.c
- `struct jump_entry,`
  see: arch/x86/include/asm/jump_label.h

5. Static keys / jump label analysis, results (x86_64):

As an example, let's add the following branch to 'getppid()', such that the system call now looks like:

```
SYSCALL_DEFINE0(getppid)
{
      int pid;

+     if (static_branch_unlikely(&key))
+             printk("I am the true branch\n");

      rcu_read_lock();
      pid = task_tgid_vnr(rcu_dereference(current->real_parent));
      rcu_read_unlock();

      return pid;
}
```

The resulting instructions with jump labels generated by GCC is:

```
ffffffff81044290 <sys_getppid>:
ffffffff81044290:       55                      push   %rbp
ffffffff81044291:       48 89 e5                mov    %rsp,%rbp
ffffffff81044294:       e9 00 00 00 00          jmpq   ffffffff81044299 <sys_getppid+0x9>
ffffffff81044299:       65 48 8b 04 25 c0 b6    mov    %gs:0xb6c0,%rax
ffffffff810442a0:       00 00
ffffffff810442a2:       48 8b 80 80 02 00 00    mov    0x280(%rax),%rax
ffffffff810442a9:       48 8b 80 b0 02 00 00    mov    0x2b0(%rax),%rax
ffffffff810442b0:       48 8b b8 e8 02 00 00    mov    0x2e8(%rax),%rdi
ffffffff810442b7:       e8 f4 d9 00 00          callq  ffffffff81051cb0 <pid_vnr>
ffffffff810442bc:       5d                      pop    %rbp
ffffffff810442bd:       48 98                   cltq
ffffffff810442bf:       c3                      retq
ffffffff810442c0:       48 c7 c7 e3 54 98 81    mov    $0xffffffff819854e3,%rdi
ffffffff810442c7:       31 c0                   xor    %eax,%eax
ffffffff810442c9:       e8 71 13 6d 00          callq  ffffffff8171563f <printk>
ffffffff810442ce:       eb c9                   jmp    ffffffff81044299 <sys_getppid+0x9>
```

Without the jump label optimization it looks like:

```
ffffffff810441f0 <sys_getppid>:
ffffffff810441f0:       8b 05 8a 52 d8 00       mov    0xd8528a(%rip),%eax        # ffffffff81dc9480 <key
ffffffff810441f6:       55                      push   %rbp
ffffffff810441f7:       48 89 e5                mov    %rsp,%rbp
ffffffff810441fa:       85 c0                   test   %eax,%eax
ffffffff810441fc:       75 27                   jne    ffffffff81044225 <sys_getppid+0x35>
ffffffff810441fe:       65 48 8b 04 25 c0 b6    mov    %gs:0xb6c0,%rax
ffffffff81044205:       00 00
ffffffff81044207:       48 8b 80 80 02 00 00    mov    0x280(%rax),%rax
ffffffff8104420e:       48 8b 80 b0 02 00 00    mov    0x2b0(%rax),%rax
ffffffff81044215:       48 8b b8 e8 02 00 00    mov    0x2e8(%rax),%rdi
ffffffff8104421c:       e8 2f da 00 00          callq  ffffffff81051c50 <pid_vnr>
ffffffff81044221:       5d                      pop    %rbp
ffffffff81044222:       48 98                   cltq
ffffffff81044224:       c3                      retq
ffffffff81044225:       48 c7 c7 13 53 98 81    mov    $0xffffffff81985313,%rdi
ffffffff8104422c:       31 c0                   xor    %eax,%eax
ffffffff8104422e:       e8 60 0f 6d 00          callq  ffffffff81715193 <printk>
ffffffff81044233:       eb c9                   jmp    ffffffff810441fe <sys_getppid+0xe>
ffffffff81044235:       66 66 2e 0f 1f 84 00    data32 nopw %cs:0x0(%rax,%rax,1)
ffffffff8104423c:       00 00 00 00
```

Thus, the disable jump label case adds a 'mov', 'test' and 'jne' instruction vs. the jump label case just has a 'no-op' or 'jmp 0'. (The jmp 0, is patched to a 5 byte atomic no-op instruction at boot-time.) Thus, the disabled jump label case adds:

```
6 (mov) + 2 (test) + 2 (jne) = 10 - 5 (5 byte jump 0) = 5 addition bytes.
```

If we then include the padding bytes, the jump label code saves, 16 total bytes of instruction memory for this small function. In this case the non-jump label function is 80 bytes long. Thus, we have saved 20% of the instruction footprint. We can in fact improve this even further, since the 5-byte no-op really can be a 2-byte no-op since we can reach the branch with a 2-byte jmp. However, we have not yet implemented optimal no-op sizes (they are currently hard-coded).

Since there are a number of static key API uses in the scheduler paths, 'pipe-test' (also known as 'perf bench sched pipe') can be used to show the performance improvement. Testing done on 3.3.0-rc2:

jump label disabled:

```
 Performance counter stats for 'bash -c /tmp/pipe-test' (50 runs):

        855.700314 task-clock                #    0.534 CPUs utilized            ( +-  0.11% )
           200,003 context-switches          #    0.234 M/sec                    ( +-  0.00% )
                 0 CPU-migrations            #    0.000 M/sec                    ( +- 39.58% )
               487 page-faults               #    0.001 M/sec                    ( +-  0.02% )
     1,474,374,262 cycles                    #    1.723 GHz                      ( +-  0.17% )
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
     1,178,049,567 instructions              #    0.80  insns per cycle          ( +-  0.06% )
       208,368,926 branches                  #  243.507 M/sec                    ( +-  0.06% )
         5,569,188 branch-misses             #    2.67% of all branches          ( +-  0.54% )

       1.601607384 seconds time elapsed                                          ( +-  0.07% )
```

jump label enabled:

```
 Performance counter stats for 'bash -c /tmp/pipe-test' (50 runs):

        841.043185 task-clock                #    0.533 CPUs utilized            ( +-  0.12% )
           200,004 context-switches          #    0.238 M/sec                    ( +-  0.00% )
                 0 CPU-migrations            #    0.000 M/sec                    ( +- 40.87% )
               487 page-faults               #    0.001 M/sec                    ( +-  0.05% )
     1,432,559,428 cycles                    #    1.703 GHz                      ( +-  0.18% )
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
     1,175,363,994 instructions              #    0.82  insns per cycle          ( +-  0.04% )
       206,859,359 branches                  #  245.956 M/sec                    ( +-  0.04% )
         4,884,119 branch-misses             #    2.36% of all branches          ( +-  0.85% )

       1.579384366 seconds time elapsed
```

The percentage of saved branches is .7%, and we've saved 12% on 'branch-misses'. This is where we would expect to get the most savings, since this optimization is about reducing the number of branches. In addition, we've saved .2% on instructions, and 2.8% on cycles and 1.4% on elapsed time.