Subject: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Glauber Costa on Thu, 26 Apr 2012 21:24:23 GMT
View Forum Message <> Reply to Message

We call the destroy function when a cgroup starts to be removed,
such as by a rmdir event.

However, because of our reference counters, some objects are still
inflight. Right now, we are decrementing the static_keys at destroy()
time, meaning that if we get rid of the last static_key reference,
some objects will still have charges, but the code to properly
uncharge them won't be run.

This becomes a problem specially if it is ever enabled again, because
now new charges will be added to the staled charges making keeping
it pretty much impossible.

We just need to be careful with the static branch activation:
since there is no particular preferred order of their activation,
we need to make sure that we only start using it after all
call sites are active. This is achieved by having a per-memcg
flag that is only updated after static_key_slow_inc() returns.
At this time, we are sure all sites are active.

This is made per-memcg, not global, for a reason:
it also has the effect of making socket accounting more
consistent. The first memcg to be limited will trigger static_key()
activation, therefore, accounting. But all the others will then be
accounted no matter what. After this patch, only limited memcgs
will have its sockets accounted.

[v2: changed a tcp limited flag for a generic proto limited flag]
[v3: update the current active flag only after the static_key update]
[v4: disarm_static_keys() inside free_work]

Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 include/net/sock.h | 9 ++++++
 mm/memcontrol.c | 31 +++++++++++++++++++-
 net/ipv4/tcp_memcontrol.c | 70 ++----
 3 files changed, 101 insertions(+), 9 deletions(-)

diff --git a/include/net/sock.h b/include/net/sock.h
index b3ebe6b..c5a2010 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -914,6 +914,15 @@ struct cg_proto {
  int   *memory_pressure;

```
  long   *sysctl_mem;
  /*
+  * active means it is currently active, and new sockets should
+  * be assigned to cgroups.
+  *
+  * activated means it was ever activated, and we need to
+  * disarm the static keys on destruction
+  */
+ bool   activated;
+ bool   active;
+ /*
   * memcg field is used to find which memcg we belong directly
   * Each memcg struct can hold more than one cg_proto, so container_of
   * won't really cut.
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b0076cc..53a0815 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -404,6 +404,7 @@ void sock_update_memcg(struct sock *sk)
 {
  if (mem_cgroup_sockets_enabled) {
   struct mem_cgroup *memcg;
+  struct cg_proto *cg_proto;

   BUG_ON(!sk->sk_prot->proto_cgroup);

@@ -423,9 +424,10 @@ void sock_update_memcg(struct sock *sk)

   rcu_read_lock();
   memcg = mem_cgroup_from_task(current);
-  if (!mem_cgroup_is_root(memcg)) {
+  cg_proto = sk->sk_prot->proto_cgroup(memcg);
+  if (!mem_cgroup_is_root(memcg) && cg_proto->active) {
    mem_cgroup_get(memcg);
-   sk->sk_cgrp = sk->sk_prot->proto_cgroup(memcg);
+   sk->sk_cgrp = cg_proto;
   }
   rcu_read_unlock();
  }
@@ -442,6 +444,14 @@ void sock_release_memcg(struct sock *sk)
  }
 }

+static void disarm_static_keys(struct mem_cgroup *memcg)
+{
+#ifdef CONFIG_INET
+ if (memcg->tcp_mem.cg_proto.activated)
+  static_key_slow_dec(&memcg_socket_limit_enabled);
```

```
+#endif
+}
+
 #ifdef CONFIG_INET
 struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 {
@@ -452,6 +462,11 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 }
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
+#else
+static inline void disarm_static_keys(struct mem_cgroup *memcg)
+{
+}
+
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4836,6 +4851,18 @@ static void free_work(struct work_struct *work)
  int size = sizeof(struct mem_cgroup);

  memcg = container_of(work, struct mem_cgroup, work_freeing);
+ /*
+  * We need to make sure that (at least for now), the jump label
+  * destruction code runs outside of the cgroup lock. It is in theory
+  * possible to call the cgroup destruction function outside of that
+  * lock, but it is not yet done. rate limiting plus the deferred
+  * interface for static_branch destruction guarantees that it will
+  * run through schedule_work(), therefore, not holding any cgroup
+  * related lock (this is, of course, until someone decides to write
+  * a schedule_work cgroup :p )
+  */
+
+ disarm_static_keys(memcg);
  if (size < PAGE_SIZE)
   kfree(memcg);
  else
diff --git a/net/ipv4/tcp_memcontrol.c b/net/ipv4/tcp_memcontrol.c
index 1517037..7790008 100644
--- a/net/ipv4/tcp_memcontrol.c
+++ b/net/ipv4/tcp_memcontrol.c
@@ -54,6 +54,8 @@ int tcp_init_cgroup(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
  cg_proto->sysctl_mem = tcp->tcp_prot_mem;
  cg_proto->memory_allocated = &tcp->tcp_memory_allocated;
  cg_proto->sockets_allocated = &tcp->tcp_sockets_allocated;
+ cg_proto->active = false;
+ cg_proto->activated = false;
  cg_proto->memcg = memcg;
```

```
  return 0;
@@ -74,12 +76,43 @@ void tcp_destroy_cgroup(struct mem_cgroup *memcg)
  percpu_counter_destroy(&tcp->tcp_sockets_allocated);

  val = res_counter_read_u64(&tcp->tcp_memory_allocated, RES_LIMIT);
-
- if (val != RESOURCE_MAX)
-  static_key_slow_dec(&memcg_socket_limit_enabled);
 }
 EXPORT_SYMBOL(tcp_destroy_cgroup);

+/*
+ * This is to prevent two writes arriving at the same time
+ * at kmem.tcp.limit_in_bytes.
+ *
+ * There is a race at the first time we write to this file:
+ *
+ * - cg_proto->activated == false for all writers.
+ * - They all do a static_key_slow_inc().
+ * - When we are finally read to decrement the static_keys,
+ *   we'll do it only once per activated cgroup. So we won't
+ *   be able to disable it.
+ *
+ *   Also, after the first caller increments the static_branch
+ *   counter, all others will return right away. That does not mean,
+ *   however, that the update is finished.
+ *
+ *   Without this mutex, it would then be possible for a second writer
+ *   to get to the update site, return
+ *
+ *   When a user updates limit of 2 cgroups at once, following happens.
+ *
+ *    CPU A    CPU B
+ *
+ * if (cg_proto->activated) if (cg->proto_activated)
+ *  static_key_inc()  static_key_inc()
+ *   => set counter 0->1  => set counter 1->2,
+ *       return immediately.
+ *   => hold mutex   => cg_proto->activated = true.
+ *   => overwrite jmps.
+ *
+ * This race was described by Kamezawa Hiroyuki.
+ */
+static DEFINE_MUTEX(tcp_set_limit_mutex);
+
 static int tcp_update_limit(struct mem_cgroup *memcg, u64 val)
 {
```

```
  struct net *net = current->nsproxy->net_ns;
@@ -107,10 +140,33 @@ static int tcp_update_limit(struct mem_cgroup *memcg, u64 val)
   tcp->tcp_prot_mem[i] = min_t(long, val >> PAGE_SHIFT,
           net->ipv4.sysctl_tcp_mem[i]);

- if (val == RESOURCE_MAX && old_lim != RESOURCE_MAX)
-  static_key_slow_dec(&memcg_socket_limit_enabled);
- else if (old_lim == RESOURCE_MAX && val != RESOURCE_MAX)
-  static_key_slow_inc(&memcg_socket_limit_enabled);
+ if (val == RESOURCE_MAX)
+  cg_proto->active = false;
+ else if (val != RESOURCE_MAX) {
+  /*
+   * ->activated needs to be written after the static_key update.
+   *  This is what guarantees that the socket activation function
+   *  is the last one to run. See sock_update_memcg() for details,
+   *  and note that we don't mark any socket as belonging to this
+   *  memcg until that flag is up.
+   *
+   *  We need to do this, because static_keys will span multiple
+   *  sites, but we can't control their order. If we mark a socket
+   *  as accounted, but the accounting functions are not patched in
+   *  yet, we'll lose accounting.
+   *
+   *  We never race with the readers in sock_update_memcg(), because
+   *  when this value change, the code to process it is not patched in
+   *  yet.
+   */
+  mutex_lock(&tcp_set_limit_mutex);
+  if (!cg_proto->activated) {
+   static_key_slow_inc(&memcg_socket_limit_enabled);
+   cg_proto->activated = true;
+  }
+  mutex_unlock(&tcp_set_limit_mutex);
+  cg_proto->active = true;
+ }

  return 0;
 }
--
1.7.7.6
```

Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Tejun Heo on Thu, 26 Apr 2012 21:39:16 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

Overall, I like this approach much better. Just some nits below.

On Thu, Apr 26, 2012 at 06:24:23PM -0300, Glauber Costa wrote:
> @@ -4836,6 +4851,18 @@ static void free_work(struct work_struct *work)
>   int size = sizeof(struct mem_cgroup);
>
>   memcg = container_of(work, struct mem_cgroup, work_freeing);
> + /*
> + * We need to make sure that (at least for now), the jump label
> + * destruction code runs outside of the cgroup lock. It is in theory
> + * possible to call the cgroup destruction function outside of that
> + * lock, but it is not yet done. rate limiting plus the deferred
> + * interface for static_branch destruction guarantees that it will
> + * run through schedule_work(), therefore, not holding any cgroup
> + * related lock (this is, of course, until someone decides to write
> + * a schedule_work cgroup :p)
> + */

Isn't the above a bit too verbose? Wouldn't just stating the locking
dependency be enough?

> + disarm_static_keys(memcg);
>   if (size < PAGE_SIZE)
>   kfree(memcg);
>   else
> diff --git a/net/ipv4/tcp_memcontrol.c b/net/ipv4/tcp_memcontrol.c
> index 1517037..7790008 100644
> --- a/net/ipv4/tcp_memcontrol.c
> +++ b/net/ipv4/tcp_memcontrol.c
> @@ -54,6 +54,8 @@ int tcp_init_cgroup(struct mem_cgroup *memcg, struct cgroup_subsys
*ss)
>   cg_proto->sysctl_mem = tcp->tcp_prot_mem;
>   cg_proto->memory_allocated = &tcp->tcp_memory_allocated;
>   cg_proto->sockets_allocated = &tcp->tcp_sockets_allocated;
> + cg_proto->active = false;
> + cg_proto->activated = false;

Isn't the memory zallocd? I find 0 / NULL / false inits unnecessary
and even misleading (can the memory be non-zero here?). Another side
effect is that it tends to get out of sync as more fields are added.

> +/*
> + * This is to prevent two writes arriving at the same time
> + * at kmem.tcp.limit_in_bytes.
> + *
> + * There is a race at the first time we write to this file:
> + *

> + * - cg_proto->activated == false for all writers.
> + * - They all do a static_key_slow_inc().
> + * - When we are finally read to decrement the static_keys,
                    ^
                    ready

> + *   we'll do it only once per activated cgroup. So we won't
> + *   be able to disable it.
> + *
> + *   Also, after the first caller increments the static_branch
> + *   counter, all others will return right away. That does not mean,
> + *   however, that the update is finished.
> + *
> + *   Without this mutex, it would then be possible for a second writer
> + *   to get to the update site, return

I kinda don't follow the above sentence.

> + *   When a user updates limit of 2 cgroups at once, following happens.
> + *
> + *   CPU A   CPU B
> + *
> + * if (cg_proto->activated) if (cg->proto_activated)
> + * static_key_inc() static_key_inc()
> + *   => set counter 0->1 => set counter 1->2,
> + *   return immediately.
> + *   => hold mutex   => cg_proto->activated = true.
> + *   => overwrite jmps.

Isn't this something which should be solved from static_keys API? Why is this being worked around from memcg? Also, I again hope that the explanation is slightly more concise.

Thanks.

--
tejun

Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Glauber Costa on Thu, 26 Apr 2012 21:58:37 GMT
View Forum Message <> Reply to Message

On 04/26/2012 06:39 PM, Tejun Heo wrote:
> Hello, Glauber.
>
> Overall, I like this approach much better. Just some nits below.
>

> On Thu, Apr 26, 2012 at 06:24:23PM -0300, Glauber Costa wrote:
>> @@ -4836,6 +4851,18 @@ static void free_work(struct work_struct *work)
>> int size = sizeof(struct mem_cgroup);
>>
>> memcg = container_of(work, struct mem_cgroup, work_freeing);
>> + /*
>> + * We need to make sure that (at least for now), the jump label
>> + * destruction code runs outside of the cgroup lock. It is in theory
>> + * possible to call the cgroup destruction function outside of that
>> + * lock, but it is not yet done. rate limiting plus the deferred
>> + * interface for static_branch destruction guarantees that it will
>> + * run through schedule_work(), therefore, not holding any cgroup
>> + * related lock (this is, of course, until someone decides to write
>> + * a schedule_work cgroup :p)
>> + */
>
> Isn't the above a bit too verbose? Wouldn't just stating the locking
> dependency be enough?

I used a lot of verbosity here because it is a tricky and racy issue.
I am fine with trimming the comments if this is considered too much.

>> cg_proto->sysctl_mem = tcp->tcp_prot_mem;
>> cg_proto->memory_allocated =&tcp->tcp_memory_allocated;
>> cg_proto->sockets_allocated =&tcp->tcp_sockets_allocated;
>> + cg_proto->active = false;
>> + cg_proto->activated = false;
>
> Isn't the memory zallocd? I find 0 / NULL / false inits unnecessary
> and even misleading (can the memory be non-zero here?). Another side
> effect is that it tends to get out of sync as more fields are added.

I can take them off.

>
>> +/*
>> + * This is to prevent two writes arriving at the same time
>> + * at kmem.tcp.limit_in_bytes.
>> + *
>> + * There is a race at the first time we write to this file:
>> + *
>> + * - cg_proto->activated == false for all writers.
>> + * - They all do a static_key_slow_inc().
>> + * - When we are finally read to decrement the static_keys,
>>   ^
>>   ready

Thanks.

>> + * we'll do it only once per activated cgroup. So we won't
>> + * be able to disable it.
>> + *
>> + * Also, after the first caller increments the static_branch
>> + * counter, all others will return right away. That does not mean,
>> + * however, that the update is finished.
>> + *
>> + * Without this mutex, it would then be possible for a second writer
>> + * to get to the update site, return
>
> I kinda don't follow the above sentence.

I will try to rephrase it for more clarity. But this is the thing behind
this patchset coming and going with so many attempts:

jump label updates are atomic given a single patch site. But they are
not atomic given multiple patch sites.

In our case, they are pretty spread around. Which means that while some
of them are already patched, some are not. If the socket marking in
sock_update_memcg is done last, we're fine, because all the accounters
test for that. Otherwise, we can misaccount.

To protect against that, we use the "activated" field. But it need to be
lock-protected, otherwise a second writer can arrive here before the
update is finished, update the accounted field, and we're down to the
same problem as before.

>> + * When a user updates limit of 2 cgroups at once, following happens.
>> + *
>> + *   CPU A   CPU B
>> + *
>> + * if (cg_proto->activated) if (cg->proto_activated)
>> + * static_key_inc() static_key_inc()
>> + * => set counter 0->1 => set counter 1->2,
>> + *   return immediately.
>> + * => hold mutex   => cg_proto->activated = true.
>> + * => overwrite jmps.
>
> Isn't this something which should be solved from static_keys API? Why
> is this being worked around from memcg? Also, I again hope that the
> explanation is slightly more concise.
>

At first I though that we could get rid of all this complication by
calling stop machine from the static_branch API. This would all
magically go away. I actually even tried it.

However, reading the code for other architectures (other than x86), I found that they usually rely on the fixed instruction size to just patch an instruction atomically and go home happy.

Using stop machine and the like would slow them down considerably. Not only slow down the static branch update (which is acceptable), but everybody else (which is horrible). It seemed to defeat the purpose of static branches a bit.

The other users of static branches seems to be fine coping with the fact that in cases with multiple-sites, they will spread in time.

Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Tejun Heo on Thu, 26 Apr 2012 22:13:24 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

On Thu, Apr 26, 2012 at 06:58:37PM -0300, Glauber Costa wrote:
> At first I though that we could get rid of all this complication by
> calling stop machine from the static_branch API. This would all
> magically go away. I actually even tried it.
>
> However, reading the code for other architectures (other than x86),
> I found that they usually rely on the fixed instruction size to just
> patch an instruction atomically and go home happy.
>
> Using stop machine and the like would slow them down considerably.
> Not only slow down the static branch update (which is acceptable),
> but everybody else (which is horrible). It seemed to defeat the
> purpose of static branches a bit.
>
> The other users of static branches seems to be fine coping with the
> fact that in cases with multiple-sites, they will spread in time.

No, what I mean is that why can't you do about the same mutexed
activated inside static_key API function instead of requiring every
user to worry about the function returning asynchronously.
ie. synchronize inside static_key API instead of in the callers.

Thanks.

--
tejun

Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Glauber Costa on Thu, 26 Apr 2012 22:17:36 GMT
View Forum Message <> Reply to Message

> No, what I mean is that why can't you do about the same mutexed
> activated inside static_key API function instead of requiring every
> user to worry about the function returning asynchronously.
> ie. synchronize inside static_key API instead of in the callers.
>

Like this?

File Attachments
1) jump_label.patch, downloaded 422 times

Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Tejun Heo on Thu, 26 Apr 2012 22:22:33 GMT
View Forum Message <> Reply to Message

Hello,

On Thu, Apr 26, 2012 at 3:17 PM, Glauber Costa <glommer@parallels.com> wrote:
>
>> No, what I mean is that why can't you do about the same mutexed
>> activated inside static_key API function instead of requiring every
>> user to worry about the function returning asynchronously.
>> ie. synchronize inside static_key API instead of in the callers.
>>
>
> Like this?

Yeah, something like that. If keeping the inc operation a single
atomic op is important for performance or whatever reasons, you can
play some trick with large negative bias value while activation is
going on and use atomic_add_return() to determine both whether it's
the first incrementer and someone else is in the process of
activating.

Thanks.

--
tejun

Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Glauber Costa on Thu, 26 Apr 2012 22:28:39 GMT

On 04/26/2012 07:22 PM, Tejun Heo wrote:
> Hello,
>
> On Thu, Apr 26, 2012 at 3:17 PM, Glauber Costa<glommer@parallels.com> wrote:
>>
>>> No, what I mean is that why can't you do about the same mutexed
>>> activated inside static_key API function instead of requiring every
>>> user to worry about the function returning asynchronously.
>>> ie. synchronize inside static_key API instead of in the callers.
>>>
>>
>> Like this?
>
> Yeah, something like that. If keeping the inc operation a single
> atomic op is important for performance or whatever reasons, you can
> play some trick with large negative bias value while activation is
> going on and use atomic_add_return() to determine both whether it's
> the first incrementer and someone else is in the process of
> activating.
>
> Thanks.
>
We need a broader audience for this, but if I understand the interface
right, those functions should not be called in fast paths at all
(contrary to the static_branch tests)

The static_branch tests can be called from irq context, so we can't just
get rid of the atomic op and use the mutex everywhere, we'd have
to live with both.

I will repost this series, with some more people in the CC list.

Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Tejun Heo on Thu, 26 Apr 2012 22:32:14 GMT

On Thu, Apr 26, 2012 at 3:28 PM, Glauber Costa <glommer@parallels.com> wrote:
> We need a broader audience for this, but if I understand the interface
> right, those functions should not be called in fast paths at all (contrary
> to the static_branch tests)
>
> The static_branch tests can be called from irq context, so we can't just get
> rid of the atomic op and use the mutex everywhere, we'd have
> to live with both.
>

> I will repost this series, with some more people in the CC list.

Great, thanks!

--
tejun
