Subject: Re: [RFC 5/7] use percpu_counters for res_counter usage
Posted by KAMEZAWA Hiroyuki on Fri, 30 Mar 2012 09:33:31 GMT
View Forum Message <> Reply to Message

(2012/03/30 17:04), Glauber Costa wrote:

> This is the bulk of the proposal.
> Updates to the res_counter are done to the percpu area, if we are
> inside what we can call the "safe zone".
>
> The safe zone is whenever we are far enough from the limit to be
> sure this update won't touch it. It is bigger the bigger the system
> is, since it grows with the number of cpus.
>
> However, for unlimited scenarios, this will always be the case.
> In those situations we are sure to never be close to the limit simply
> because the limit is high enough.
>
> Small consumers will also be safe. This includes workloads that
> pin and unpin memory often, but never grow the total size of memory
> by too much.
>
> The memory reported (reads of RES_USAGE) in this way is actually
> more precise than we currently have (Actually would be, if we
> would disable the memcg caches): I am using percpu_counter_sum(),
> meaning the cpu areas will be scanned and accumulated.
>
> percpu_counter_read() can also be used for reading RES_USAGE.
> We could then be off by a factor of batch_size * #cpus. I consider
> this to be not worse than the current situation with the memcg caches.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> ---
> include/linux/res_counter.h | 15 ++++++----
> kernel/res_counter.c | 61 ++++++++++++++++++++++++++++++++++++-------
> 2 files changed, 60 insertions(+), 16 deletions(-)
>
> diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
> index 53b271c..8c1c20e 100644
> --- a/include/linux/res_counter.h
> +++ b/include/linux/res_counter.h
> @@ -25,7 +25,6 @@ struct res_counter {
>   /*
>   * the current resource consumption level
>   */
> - unsigned long long usage;
>   struct percpu_counter usage_pcp;
>   /*

>   * the maximal value of the usage from the counter creation
> @@ -138,10 +137,12 @@ static inline unsigned long long res_counter_margin(struct
res_counter *cnt)
> {
>   unsigned long long margin;
>   unsigned long flags;
> + u64 usage;
>
>   raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
> - if (cnt->limit > cnt->usage)
> - margin = cnt->limit - cnt->usage;
> + usage = __percpu_counter_sum_locked(&cnt->usage_pcp);
> + if (cnt->limit > usage)
> + margin = cnt->limit - usage;
>   else
>   margin = 0;
>   raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
> @@ -160,12 +161,14 @@ res_counter_soft_limit_excess(struct res_counter *cnt)
> {
>   unsigned long long excess;
>   unsigned long flags;
> + u64 usage;
>
>   raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
> - if (cnt->usage <= cnt->soft_limit)
> + usage = __percpu_counter_sum_locked(&cnt->usage_pcp);
> + if (usage <= cnt->soft_limit)
>   excess = 0;
>   else
> - excess = cnt->usage - cnt->soft_limit;
> + excess = usage - cnt->soft_limit;
>   raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
>   return excess;
> }
> @@ -175,7 +178,7 @@ static inline void res_counter_reset_max(struct res_counter *cnt)
>   unsigned long flags;
>
>   raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
> - cnt->max_usage = cnt->usage;
> + cnt->max_usage = __percpu_counter_sum_locked(&cnt->usage_pcp);
>   raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
> }
>
> diff --git a/kernel/res_counter.c b/kernel/res_counter.c
> index 052efaf..8a99943 100644
> --- a/kernel/res_counter.c
> +++ b/kernel/res_counter.c
> @@ -28,9 +28,28 @@ int __res_counter_add(struct res_counter *c, long val, bool fail)

> int ret = 0;
> u64 usage;
>
> + rcu_read_lock();
> +

Hmm... isn't it better to synchronize percpu usage to the main counter
by smp_call_function() or some at set limit ? after set 'global' mode ?

set global mode
smp_call_function(drain all pcp counters to main counter)
set limit.
unset global mode

> + if (val < 0) {
> + percpu_counter_add(&c->usage_pcp, val);
> + rcu_read_unlock();
> + return 0;
> + }

Memo:
memcg's uncharge path is batched ....so..it will be bigger than
percpu_counter_batch() in most of cases. (And lock conflict is enough low.)

> +
> + usage = percpu_counter_read(&c->usage_pcp);
> +
> + if (percpu_counter_read(&c->usage_pcp) + val <
> +   (c->limit + num_online_cpus() * percpu_counter_batch)) {

c->limit - num_online_cpus() * percpu_counter_batch ?

Anyway, you can pre-calculate this value at cpu hotplug event..

> + percpu_counter_add(&c->usage_pcp, val);
> + rcu_read_unlock();
> + return 0;
> + }
> +
> + rcu_read_unlock();
> +
> raw_spin_lock(&c->usage_pcp.lock);

>
> - usage = c->usage;
> + usage = __percpu_counter_sum_locked(&c->usage_pcp);

Hmm.... this part doesn't seem very good.
I don't think for_each_online_cpu() here will not be a way to the final win.
Under multiple hierarchy, you may need to call for_each_online_cpu() in each level.

Can't you update percpu counter's core logic to avoid using for_each_online_cpu() ?
For example, if you know what cpus have caches, you can use that cpu mask...

Memo:
Current implementation of memcg's percpu counting is reserving usage before its real use.
In usual, the kernel don't have to scan percpu caches and just drain caches from cpus
reserving usages if we need to cancel reserved usages. (And it's automatically canceled
when cpu's memcg changes.)

And 'reserving' avoids caching in multi-level counters,....it updates multiple counters
in batch and memcg core don't need to walk res_counter ancestors in fast path.

Considering res_counter's characteristics
 - it has _hard_ limit
 - it can be tree and usages are propagated to ancestors
 - all ancestors has hard limit.

Isn't it better to generalize 'reserving resource' model ?
You can provide 'precise usage' to the user by some logic.

>
>   if (usage + val > c->limit) {
>   c->failcnt++;
> @@ -39,9 +58,9 @@ int __res_counter_add(struct res_counter *c, long val, bool fail)
>   goto out;
>   }
>
> - usage += val;
>
> - c->usage = usage;
> + c->usage_pcp.count += val;
> +
>   if (usage > c->max_usage)
>   c->max_usage = usage;
>
> @@ -115,14 +134,28 @@ int res_counter_set_limit(struct res_counter *cnt,
>   unsigned long long limit)
> {
>   unsigned long flags;

> - int ret = -EBUSY;
> + int ret = 0;
> + u64 usage;
> + bool allowed;
>
> + /*
> + * This is to prevent conflicts with people reading
> + * from the pcp counters
> + */
> + synchronize_rcu();

>   raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);

> - if (cnt->usage <= limit) {
> - cnt->limit = limit;
> - ret = 0;
> +
> + usage = __percpu_counter_sum_locked(&cnt->usage_pcp);
> + if (usage >= limit) {
> + allowed = false;
> + ret = -EBUSY;
> + goto out;
>   }
> +
> + cnt->limit = limit;
> +out:
>   raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
> +
>   return ret;
> }
>
> @@ -130,8 +163,6 @@ static inline unsigned long long *
> res_counter_member(struct res_counter *counter, int member)
> {
>   switch (member) {
> - case RES_USAGE:
> - return &counter->usage;
>   case RES_MAX_USAGE:
>   return &counter->max_usage;
>   case RES_LIMIT:
> @@ -153,7 +184,11 @@ u64 res_counter_read_u64(struct res_counter *counter, int member)
>   u64 ret;
>
>   raw_spin_lock_irqsave(&counter->usage_pcp.lock, flags);
> - ret = *res_counter_member(counter, member);
> + if (member == RES_USAGE) {
> + synchronize_rcu();

Can we user synchronize_rcu() under spin_lock ?
I don't think this synchronize_rcu() is required.
percpu counter is not precise in its nature. __percpu_counter_sum_locked() will be enough.

> + ret = __percpu_counter_sum_locked(&counter->usage_pcp);
> + } else
> + ret = *res_counter_member(counter, member);
>   raw_spin_unlock_irqrestore(&counter->usage_pcp.lock, flags);
>
>   return ret;
> @@ -161,6 +196,12 @@ u64 res_counter_read_u64(struct res_counter *counter, int member)
> #else
> u64 res_counter_read_u64(struct res_counter *counter, int member)
> {
> + if (member == RES_USAGE) {
> + u64 ret;
> + synchronize_rcu();

ditto.

> + ret = percpu_counter_sum(&counter->usage_pcp);
> + return ret;
> + }
>   return *res_counter_member(counter, member);
> }
> #endif

Thanks,
-Kame