Subject: [RFC 0/7] Initial proposal for faster res_counter updates
Posted by Glauber Costa on Fri, 30 Mar 2012 08:04:38 GMT
View Forum Message <> Reply to Message

Hi,

Here is my take about how we can make res_counter updates faster.
Keep in mind this is a bit of a hack intended as a proof of concept.

The pros I see with this:

* free updates in non-constrained paths. non-constrained paths includes
  unlimited scenarios, but also ones in which we are far from the limit.

* No need to have a special cache mechanism in memcg. The problem with
  the caching is my opinion, is that we will forward-account pages, meaning
  that we'll consider accounted pages we never used. I am not sure
  anyone actually ran into this, but in theory, this can fire events
  much earlier than it should.

But the cons:

* percpu counters have signed quantities, so this would limit us 4G.
  We can add a shift and then count pages instead of bytes, but we
  are still in the 16T area here. Maybe we really need more than that.

* some of the additions here may slow down the percpu_counters for
  users that don't care about our usage. Things about min/max tracking
  enter in this category.

* growth of the percpu memory.

It is still not clear for me if we should use percpu_counters as this
patch implies, or if we should just replicate its functionality.

I need to go through at least one more full round of auditing before
making sure the locking is safe, specially my use of synchronize_rcu().

As for measurements, the cache we have in memcg kind of distort things.
I need to either disable it, or find the cases in which it is likely
to lose and benchmark them, such as deep hierarchy concurrent updates
with common parents.

I also included a possible optimization that can be done when we
are close to the limit to avoid the initial tests altogether, but
it needs to be extended to avoid scanning the percpu areas as well.

In summary, if this is to be carried forward, it definitely needs

some love. It should be, however, more than enough to make the
proposal clear.

Comments are appreciated.

Glauber Costa (7):
  split percpu_counter_sum
  consolidate all res_counter manipulation
  bundle a percpu counter into res_counters and use its lock
  move res_counter_set limit to res_counter.c
  use percpu_counters for res_counter usage
  Add min and max statistics to percpu_counter
  Global optimization

 include/linux/percpu_counter.h | 3 +
 include/linux/res_counter.h | 63 ++++++----------
 kernel/res_counter.c | 151 +++++++++++++++++++++++++++++++++-------
 lib/percpu_counter.c | 16 ++++-
 4 files changed, 151 insertions(+), 82 deletions(-)

--
1.7.4.1

Subject: [RFC 2/7] consolidate all res_counter manipulation
Posted by Glauber Costa on Fri, 30 Mar 2012 08:04:40 GMT
View Forum Message <> Reply to Message

This patch moves all the locked updates done to res_counter to
__res_counter_add(). It gets flags for the special cases like nofail(),
and a negative value of the increment means uncharge.

This will be useful later when we start doing percpu_counter updates.

Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 kernel/res_counter.c | 59 +++++++++++++++++++++++++++++++-------------------
 1 files changed, 28 insertions(+), 31 deletions(-)

diff --git a/kernel/res_counter.c b/kernel/res_counter.c
index b8a3d6a..ecb4aad 100644
--- a/kernel/res_counter.c
+++ b/kernel/res_counter.c
@@ -22,17 +22,32 @@ void res_counter_init(struct res_counter *counter, struct res_counter
*parent)
  counter->parent = parent;
 }

```
-int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
+int __res_counter_add(struct res_counter *c, long val, bool fail)
 {
- if (counter->usage + val > counter->limit) {
-  counter->failcnt++;
-  return -ENOMEM;
+ int ret = 0;
+ u64 usage;
+
+ spin_lock(&c->lock);
+
+ usage = c->usage;
+
+ if (usage + val > c->limit) {
+  c->failcnt++;
+  ret = -ENOMEM;
+  if (fail)
+   goto out;
  }

- counter->usage += val;
- if (counter->usage > counter->max_usage)
-  counter->max_usage = counter->usage;
- return 0;
+ usage += val;
+
+ c->usage = usage;
+ if (usage > c->max_usage)
+  c->max_usage = usage;
+
+out:
+ spin_unlock(&c->lock);
+ return ret;
+
 }

 int res_counter_charge(struct res_counter *counter, unsigned long val,
@@ -45,9 +60,7 @@ int res_counter_charge(struct res_counter *counter, unsigned long val,
  *limit_fail_at = NULL;
  local_irq_save(flags);
  for (c = counter; c != NULL; c = c->parent) {
-  spin_lock(&c->lock);
-  ret = res_counter_charge_locked(c, val);
-  spin_unlock(&c->lock);
+  ret = __res_counter_add(c, val, true);
   if (ret < 0) {
    *limit_fail_at = c;
    goto undo;
```

```
@@ -57,9 +70,7 @@ int res_counter_charge(struct res_counter *counter, unsigned long val,
  goto done;
 undo:
  for (u = counter; u != c; u = u->parent) {
-  spin_lock(&u->lock);
-  res_counter_uncharge_locked(u, val);
-  spin_unlock(&u->lock);
+  __res_counter_add(u, -val, false);
  }
 done:
  local_irq_restore(flags);
@@ -77,11 +88,7 @@ int res_counter_charge_nofail(struct res_counter *counter, unsigned long
val,
  *limit_fail_at = NULL;
  local_irq_save(flags);
  for (c = counter; c != NULL; c = c->parent) {
-  spin_lock(&c->lock);
-  r = res_counter_charge_locked(c, val);
-  if (r)
-   c->usage += val;
-  spin_unlock(&c->lock);
+  r = __res_counter_add(c, val, false);
   if (r < 0 && ret == 0) {
    *limit_fail_at = c;
    ret = r;
@@ -91,13 +98,6 @@ int res_counter_charge_nofail(struct res_counter *counter, unsigned long
val,

  return ret;
 }
-void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
-{
- if (WARN_ON(counter->usage < val))
-  val = counter->usage;
-
- counter->usage -= val;
-}

 void res_counter_uncharge(struct res_counter *counter, unsigned long val)
 {
@@ -105,11 +105,8 @@ void res_counter_uncharge(struct res_counter *counter, unsigned long
val)
  struct res_counter *c;

  local_irq_save(flags);
- for (c = counter; c != NULL; c = c->parent) {
-  spin_lock(&c->lock);
-  res_counter_uncharge_locked(c, val);
```

```
-  spin_unlock(&c->lock);
- }
+ for (c = counter; c != NULL; c = c->parent)
+  __res_counter_add(c, -val, false);
  local_irq_restore(flags);
 }

--
1.7.4.1
```

Subject: [RFC 3/7] bundle a percpu counter into res_counters and use its lock
Posted by Glauber Costa on Fri, 30 Mar 2012 08:04:41 GMT
View Forum Message <> Reply to Message

This is a preparation patch.
It bundles a percpu_counter into the resource counter. But it doesn't
do accounting with it just yet.

Instead. this preparation patch removes the res_counter spinlock,
and rely on the percpu_counter own lock for that.

Over time, this need to be done with acessors if we really plan to merge
it. But right now it can be used to give an idea about how it might be.

Signed-off-by: Glauber Costa <glommer@parallels.com>

```
---
 include/linux/res_counter.h |   30 ++++++++++++++----------------
 kernel/res_counter.c        |   15 ++++++++-------
 2 files changed, 21 insertions(+), 24 deletions(-)

diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
index a860183..d4f3674 100644
--- a/include/linux/res_counter.h
+++ b/include/linux/res_counter.h
@@ -26,6 +26,7 @@ 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
   */
@@ -43,11 +44,6 @@ struct res_counter {
   */
  unsigned long long failcnt;
  /*
-  * the lock to protect all of the above.
```

```
-  * the routines below consider this to be IRQ-safe
-  */
- spinlock_t lock;
- /*
   * Parent counter, used for hierarchial resource accounting
   */
  struct res_counter *parent;
@@ -143,12 +139,12 @@ static inline unsigned long long res_counter_margin(struct res_counter
*cnt)
  unsigned long long margin;
  unsigned long flags;

- spin_lock_irqsave(&cnt->lock, flags);
+ raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
  if (cnt->limit > cnt->usage)
   margin = cnt->limit - cnt->usage;
  else
   margin = 0;
- spin_unlock_irqrestore(&cnt->lock, flags);
+ raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
  return margin;
 }

@@ -165,12 +161,12 @@ res_counter_soft_limit_excess(struct res_counter *cnt)
  unsigned long long excess;
  unsigned long flags;

- spin_lock_irqsave(&cnt->lock, flags);
+ raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
  if (cnt->usage <= cnt->soft_limit)
   excess = 0;
  else
   excess = cnt->usage - cnt->soft_limit;
- spin_unlock_irqrestore(&cnt->lock, flags);
+ raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
  return excess;
 }

@@ -178,18 +174,18 @@ static inline void res_counter_reset_max(struct res_counter *cnt)
 {
  unsigned long flags;

- spin_lock_irqsave(&cnt->lock, flags);
+ raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
  cnt->max_usage = cnt->usage;
- spin_unlock_irqrestore(&cnt->lock, flags);
+ raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
 }
```

```
 static inline void res_counter_reset_failcnt(struct res_counter *cnt)
 {
  unsigned long flags;

- spin_lock_irqsave(&cnt->lock, flags);
+ raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
  cnt->failcnt = 0;
- spin_unlock_irqrestore(&cnt->lock, flags);
+ raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
 }

 static inline int res_counter_set_limit(struct res_counter *cnt,
@@ -198,12 +194,12 @@ static inline int res_counter_set_limit(struct res_counter *cnt,
  unsigned long flags;
  int ret = -EBUSY;

- spin_lock_irqsave(&cnt->lock, flags);
+ raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
  if (cnt->usage <= limit) {
   cnt->limit = limit;
   ret = 0;
  }
- spin_unlock_irqrestore(&cnt->lock, flags);
+ raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
  return ret;
 }

@@ -213,9 +209,9 @@ res_counter_set_soft_limit(struct res_counter *cnt,
 {
  unsigned long flags;

- spin_lock_irqsave(&cnt->lock, flags);
+ raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
  cnt->soft_limit = soft_limit;
- spin_unlock_irqrestore(&cnt->lock, flags);
+ raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
  return 0;
 }

diff --git a/kernel/res_counter.c b/kernel/res_counter.c
index ecb4aad..70c46c9 100644
--- a/kernel/res_counter.c
+++ b/kernel/res_counter.c
@@ -11,15 +11,16 @@
 #include <linux/parser.h>
 #include <linux/fs.h>
 #include <linux/res_counter.h>
```

```
+#include <linux/percpu_counter.h>
 #include <linux/uaccess.h>
 #include <linux/mm.h>

 void res_counter_init(struct res_counter *counter, struct res_counter *parent)
 {
- spin_lock_init(&counter->lock);
  counter->limit = RESOURCE_MAX;
  counter->soft_limit = RESOURCE_MAX;
  counter->parent = parent;
+ percpu_counter_init(&counter->usage_pcp, 0);
 }

 int __res_counter_add(struct res_counter *c, long val, bool fail)
@@ -27,7 +28,7 @@ int __res_counter_add(struct res_counter *c, long val, bool fail)
  int ret = 0;
  u64 usage;

- spin_lock(&c->lock);
+ raw_spin_lock(&c->usage_pcp.lock);

  usage = c->usage;

@@ -45,7 +46,7 @@ int __res_counter_add(struct res_counter *c, long val, bool fail)
   c->max_usage = usage;

 out:
- spin_unlock(&c->lock);
+ raw_spin_unlock(&c->usage_pcp.lock);
  return ret;

 }
@@ -137,9 +138,9 @@ u64 res_counter_read_u64(struct res_counter *counter, int member)
  unsigned long flags;
  u64 ret;

- spin_lock_irqsave(&counter->lock, flags);
+ raw_spin_lock_irqsave(&counter->usage_pcp.lock, flags);
  ret = *res_counter_member(counter, member);
- spin_unlock_irqrestore(&counter->lock, flags);
+ raw_spin_unlock_irqrestore(&counter->usage_pcp.lock, flags);

  return ret;
 }
@@ -187,9 +188,9 @@ int res_counter_write(struct res_counter *counter, int member,
   if (*end != '\0')
    return -EINVAL;
  }
```

```
- spin_lock_irqsave(&counter->lock, flags);
+ raw_spin_lock_irqsave(&counter->usage_pcp.lock, flags);
  val = res_counter_member(counter, member);
  *val = tmp;
- spin_unlock_irqrestore(&counter->lock, flags);
+ raw_spin_unlock_irqrestore(&counter->usage_pcp.lock, flags);
  return 0;
 }
--
1.7.4.1
```

Subject: [RFC 4/7] move res_counter_set limit to res_counter.c
Posted by Glauber Costa on Fri, 30 Mar 2012 08:04:42 GMT
View Forum Message <> Reply to Message

Preparation patch. Function is about to get complication to be
inline. Move it to the main file for consistency.

Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 include/linux/res_counter.h | 17 ++----------
 kernel/res_counter.c | 14 +++++++++++
 2 files changed, 16 insertions(+), 15 deletions(-)

```
diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
index d4f3674..53b271c 100644
--- a/include/linux/res_counter.h
+++ b/include/linux/res_counter.h
@@ -188,21 +188,8 @@ static inline void res_counter_reset_failcnt(struct res_counter *cnt)
  raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
 }

-static inline int res_counter_set_limit(struct res_counter *cnt,
-  unsigned long long limit)
-{
- unsigned long flags;
- int ret = -EBUSY;
-
- raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
- if (cnt->usage <= limit) {
-  cnt->limit = limit;
-  ret = 0;
- }
- raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
- return ret;
-}
-
```

```
+int res_counter_set_limit(struct res_counter *cnt,
+     unsigned long long limit);
 static inline int
 res_counter_set_soft_limit(struct res_counter *cnt,
     unsigned long long soft_limit)
diff --git a/kernel/res_counter.c b/kernel/res_counter.c
index 70c46c9..052efaf 100644
--- a/kernel/res_counter.c
+++ b/kernel/res_counter.c
@@ -111,6 +111,20 @@ void res_counter_uncharge(struct res_counter *counter, unsigned long
val)
  local_irq_restore(flags);
 }

+int res_counter_set_limit(struct res_counter *cnt,
+     unsigned long long limit)
+{
+ unsigned long flags;
+ int ret = -EBUSY;
+
+ raw_spin_lock_irqsave(&cnt->usage_pcp.lock, flags);
+ if (cnt->usage <= limit) {
+  cnt->limit = limit;
+  ret = 0;
+ }
+ raw_spin_unlock_irqrestore(&cnt->usage_pcp.lock, flags);
+ return ret;
+}

 static inline unsigned long long *
 res_counter_member(struct res_counter *counter, int member)
--
1.7.4.1
```

Subject: [RFC 5/7] use percpu_counters for res_counter usage
Posted by Glauber Costa on Fri, 30 Mar 2012 08:04:43 GMT
View Forum Message <> Reply to Message

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();
+
+ if (val < 0) {
+  percpu_counter_add(&c->usage_pcp, val);
+  rcu_read_unlock();
+  return 0;
+ }
+
+ usage = percpu_counter_read(&c->usage_pcp);
+
+ if (percpu_counter_read(&c->usage_pcp) + val <
+     (c->limit + num_online_cpus() * percpu_counter_batch)) {
```

```
+  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);

  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();
+  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();
+  ret = percpu_counter_sum(&counter->usage_pcp);
+  return ret;
+ }
  return *res_counter_member(counter, member);
 }
 #endif
--
1.7.4.1
```

Subject: [RFC 6/7] Add min and max statistics to percpu_counter
Posted by Glauber Costa on Fri, 30 Mar 2012 08:04:44 GMT
View Forum Message <> Reply to Message

Because percpu counters can accumulate their per-cpu sums
over time outside of the control of the callers, we need
this patch that updates the maximum values when that
happens.

I am adding a mininum value as well for conistency, due to
the signed nature of the percpu_counters.

However, I am not sure this will be of general use, and might
be yet another indication that we need to duplicate those
structures...

Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 include/linux/percpu_counter.h | 2 ++
 include/linux/res_counter.h | 6 +-----
 kernel/res_counter.c | 6 +++---
 lib/percpu_counter.c | 4 ++++
 4 files changed, 10 insertions(+), 8 deletions(-)

diff --git a/include/linux/percpu_counter.h b/include/linux/percpu_counter.h
index 8310548..639d2d5 100644
--- a/include/linux/percpu_counter.h
+++ b/include/linux/percpu_counter.h
@@ -18,6 +18,8 @@
 struct percpu_counter {
  raw_spinlock_t lock;
  s64 count;
+ s64 max;
+ s64 min;
 #ifdef CONFIG_HOTPLUG_CPU
  struct list_head list; /* All percpu_counters are on a list */
 #endif
diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
index 8c1c20e..3527827 100644
--- a/include/linux/res_counter.h
+++ b/include/linux/res_counter.h
@@ -27,10 +27,6 @@ struct res_counter {
   */
  struct percpu_counter usage_pcp;
  /*
- * the maximal value of the usage from the counter creation
- */
- unsigned long long max_usage;
- /*

```
   * the limit that usage cannot exceed
   */
  unsigned long long limit;
@@ -178,7 +174,7 @@ 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 = __percpu_counter_sum_locked(&cnt->usage_pcp);
+ cnt->usage_pcp.max = __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 8a99943..7b05208 100644
--- a/kernel/res_counter.c
+++ b/kernel/res_counter.c
@@ -61,8 +61,8 @@ int __res_counter_add(struct res_counter *c, long val, bool fail)

  c->usage_pcp.count += val;

- if (usage > c->max_usage)
-  c->max_usage = usage;
+ if (usage > c->usage_pcp.max)
+  c->usage_pcp.max = usage;

 out:
  raw_spin_unlock(&c->usage_pcp.lock);
@@ -164,7 +164,7 @@ res_counter_member(struct res_counter *counter, int member)
 {
  switch (member) {
  case RES_MAX_USAGE:
-  return &counter->max_usage;
+  return &counter->usage_pcp.max;
  case RES_LIMIT:
   return &counter->limit;
  case RES_FAILCNT:
diff --git a/lib/percpu_counter.c b/lib/percpu_counter.c
index 0b6a672..6dff70b 100644
--- a/lib/percpu_counter.c
+++ b/lib/percpu_counter.c
@@ -81,6 +81,10 @@ void __percpu_counter_add(struct percpu_counter *fbc, s64 amount, s32
batch)
   raw_spin_lock(&fbc->lock);
   fbc->count += count;
   __this_cpu_write(*fbc->counters, 0);
+  if (fbc->count > fbc->max)
+   fbc->max = fbc->count;
+  if (fbc->count < fbc->min)
```

```
+   fbc->min = fbc->count;
   raw_spin_unlock(&fbc->lock);
  } else {
   __this_cpu_write(*fbc->counters, count);
--
1.7.4.1
```

Subject: [RFC 7/7] Global optimization
Posted by Glauber Costa on Fri, 30 Mar 2012 08:04:45 GMT
View Forum Message <> Reply to Message

When we are close to the limit, doing percpu_counter_add and its
equivalent tests is a waste of time.

This patch introduce a "global" state flag to the res_counter.
When we are close to the limit, this flag is set and we skip directly
to the locked part. The flag is unset when we are far enough away from
the limit.

In this mode, we function very much like the original resource counter

The main difference right now is that we still scan all the cpus.
This should however be very easy to avoid, with a flusher function
that empties the per-cpu areas, and then updating usage_pcp directly.

This should be doable because once we get the global flag, we know
no one else would be adding to the percpu areas any longer.

Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 include/linux/res_counter.h | 1 +
 kernel/res_counter.c | 18 ++++++++++++++++++
 2 files changed, 19 insertions(+), 0 deletions(-)

```
diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
index 3527827..a8e4646 100644
--- a/include/linux/res_counter.h
+++ b/include/linux/res_counter.h
@@ -30,6 +30,7 @@ struct res_counter {
   * the limit that usage cannot exceed
   */
  unsigned long long limit;
+ bool global;
  /*
   * the limit that usage can be exceed
   */
diff --git a/kernel/res_counter.c b/kernel/res_counter.c
```

```
index 7b05208..859a27d 100644
--- a/kernel/res_counter.c
+++ b/kernel/res_counter.c
@@ -29,6 +29,8 @@ int __res_counter_add(struct res_counter *c, long val, bool fail)
  u64 usage;

  rcu_read_lock();
+ if (c->global)
+  goto global;

  if (val < 0) {
   percpu_counter_add(&c->usage_pcp, val);
@@ -45,9 +47,25 @@ int __res_counter_add(struct res_counter *c, long val, bool fail)
   return 0;
  }

+global:
  rcu_read_unlock();

  raw_spin_lock(&c->usage_pcp.lock);
+ usage = __percpu_counter_sum_locked(&c->usage_pcp);
+
+ /* everyone that could update global is under lock
+  * reader could miss a transition, but that is not a problem,
+  * since we are always using percpu_counter_sum anyway
+  */
+
+ if (!c->global && val > 0 && usage + val >
+     (c->limit + num_online_cpus() * percpu_counter_batch))
+  c->global = true;
+
+ if (c->global && val < 0 && usage + val <
+     (c->limit + num_online_cpus() * percpu_counter_batch))
+  c->global = false;
+

  usage = __percpu_counter_sum_locked(&c->usage_pcp);

--
1.7.4.1
```

Subject: Re: [RFC 0/7] Initial proposal for faster res_counter updates
Posted by KAMEZAWA Hiroyuki on Fri, 30 Mar 2012 08:32:06 GMT
View Forum Message <> Reply to Message

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

> Hi,
>
> Here is my take about how we can make res_counter updates faster.
> Keep in mind this is a bit of a hack intended as a proof of concept.
>
> The pros I see with this:
>
> * free updates in non-constrained paths. non-constrained paths includes
> unlimited scenarios, but also ones in which we are far from the limit.
>
> * No need to have a special cache mechanism in memcg. The problem with
> the caching is my opinion, is that we will forward-account pages, meaning
> that we'll consider accounted pages we never used. I am not sure
> anyone actually ran into this, but in theory, this can fire events
> much earlier than it should.
>

Note: Assume a big system which has many cpus, and user wants to devide
the system into containers. Current memcg's percpu caching is done
only when a task in memcg is on the cpu, running. So, it's not so dangerous
as it looks.

But yes, if we can drop memcg's code, it's good. Then, we can remove some
amount of codes.

> But the cons:
>
> * percpu counters have signed quantities, so this would limit us 4G.
> We can add a shift and then count pages instead of bytes, but we
> are still in the 16T area here. Maybe we really need more than that.
>

....
struct percpu_counter {
	raw_spinlock_t lock;
	s64 count;

s64 limtes us 4G ?

> * some of the additions here may slow down the percpu_counters for
> users that don't care about our usage. Things about min/max tracking
> enter in this category.
>

I think it's not very good to increase size of percpu counter. It's already

very big...Hm. How about

```
 struct percpu_counter_lazy {
  struct percpu_counter pcp;
  extra information
  s64 margin;
 }
```
?

> * growth of the percpu memory.
>

This may be a concern.

I'll look into patches.

Thanks,
-Kame

> It is still not clear for me if we should use percpu_counters as this
> patch implies, or if we should just replicate its functionality.
>
> I need to go through at least one more full round of auditing before
> making sure the locking is safe, specially my use of synchronize_rcu().
>
> As for measurements, the cache we have in memcg kind of distort things.
> I need to either disable it, or find the cases in which it is likely
> to lose and benchmark them, such as deep hierarchy concurrent updates
> with common parents.
>
> I also included a possible optimization that can be done when we
> are close to the limit to avoid the initial tests altogether, but
> it needs to be extended to avoid scanning the percpu areas as well.
>
> In summary, if this is to be carried forward, it definitely needs
> some love. It should be, however, more than enough to make the
> proposal clear.
>
> Comments are appreciated.
>
> Glauber Costa (7):
>   split percpu_counter_sum
>   consolidate all res_counter manipulation
>   bundle a percpu counter into res_counters and use its lock
>   move res_counter_set limit to res_counter.c
>   use percpu_counters for res_counter usage

> Add min and max statistics to percpu_counter
> Global optimization
>
> include/linux/percpu_counter.h | 3 +
> include/linux/res_counter.h | 63 ++++++-----------
> kernel/res_counter.c | 151 ++++++++++++++++++++++++++++-----------
> lib/percpu_counter.c | 16 ++++-
> 4 files changed, 151 insertions(+), 82 deletions(-)
>

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

Subject: Re: [RFC 5/7] use percpu_counters for res_counter usage
Posted by KAMEZAWA Hiroyuki on Fri, 30 Mar 2012 09:58:03 GMT

(2012/03/30 18:33), KAMEZAWA Hiroyuki wrote:

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

>
> 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.
>

Ah....one more point. please see this memcg's code.
==

```
        if (nr_pages == 1 && consume_stock(memcg)) {
                /*
                 * It seems dagerous to access memcg without css_get().
                 * But considering how consume_stok works, it's not
                 * necessary. If consume_stock success, some charges
                 * from this memcg are cached on this cpu. So, we
                 * don't need to call css_get()/css_tryget() before
                 * calling consume_stock().
                 */
                rcu_read_unlock();
                goto done;
        }
        /* after here, we may be blocked. we need to get refcnt */
        if (!css_tryget(&memcg->css)) {
                rcu_read_unlock();
```

```
                goto again;
        }
==
```

Now, we do consume 'reserved' usage, we can avoid css_get(), an heavy atomic ops. You may need to move this code as

```
 rcu_read_lock()
 ....
 res_counter_charge()
 if (failure) {
  css_tryget()
  rcu_read_unlock()
 } else {
  rcu_read_unlock()
  return success;
 }
```

to compare performance. This css_get() affects performance very very much.

Thanks,
-Kame

Subject: Re: [RFC 0/7] Initial proposal for faster res_counter updates
Posted by Glauber Costa on Fri, 30 Mar 2012 10:46:17 GMT
View Forum Message <> Reply to Message

> Note: Assume a big system which has many cpus, and user wants to devide
> the system into containers. Current memcg's percpu caching is done
> only when a task in memcg is on the cpu, running. So, it's not so dangerous
> as it looks.

Agree. I actually think it is pretty
> But yes, if we can drop memcg's code, it's good. Then, we can remove some
> amount of codes.
>
>> But the cons:
>>
>> * percpu counters have signed quantities, so this would limit us 4G.
>> We can add a shift and then count pages instead of bytes, but we
>> are still in the 16T area here. Maybe we really need more than that.
>>
>
> ....
> struct percpu_counter {
> raw_spinlock_t lock;
> s64 count;

>
> s64 limtes us 4G ?
>
Yes, I actually explicitly mentioned that. We can go to 16T if we track
pages
instead of bytes (I considered having the res_counter initialization code to
specify a shift, so we could be generic).

But I believe that if we go this route, we'll need to either:
1) Have our own internal implementation of what percpu counters does
2) create u64 acessors that would cast that to u64 in the operations.
Since it
    is a 64 bit field anyway it should be doable. But being doable
doesn't mean we
    should do it....
3) Have a different percpu_counter structure, something like struct
percpu_positive_counter.

>
>> * some of the additions here may slow down the percpu_counters for
>>   users that don't care about our usage. Things about min/max tracking
>>   enter in this category.
>>
>
>
> I think it's not very good to increase size of percpu counter. It's already
> very big...Hm. How about
>
> struct percpu_counter_lazy {
>   struct percpu_counter pcp;
>   extra information
>   s64 margin;
> }
> ?

Can work, but we need something that also solves the signedness problem.
Maybe we can use a union for that, and then stuff things in the end of a
different
structure just for the users that want it.

Subject: Re: [RFC 5/7] use percpu_counters for res_counter usage
Posted by Glauber Costa on Fri, 30 Mar 2012 12:59:32 GMT
View Forum Message <> Reply to Message

> 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 ?

Yes. I think it should be done after global mode is set.
My idea is to flush all the percpu data, and then start treating that
essentially as a res_counter is today.

>
> 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.)
>

I don't get what you mean.
It is batched, because charge is batched. If we un-batch one, we
un-batch another. And if we don't we don't do for any.

>> +
>> + 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..

Beautiful. Haven't thought about that.

Thanks.

>
>> + 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...

A mask should work, yes.

Flipping a bit when a cpu update its data shouldn't hurt that much.
There is cache sharing and everything, but in most cases we won't be
really making it dirty.

> 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 ?

It would be nice to see an implementation of that as well to see how it
will turn up.

Meanwhile, points to consider over the things you raised:

1) I think if we use something like the global flag as I described, we
can pretty much guarantee hard limits in the memcg code.

2) Specially because it is a tree with usage propagated to the
ancestors, is that I went with a percpu approach.

See, We can reserve as many pages as we want. This only happens in the
level we are reserving.
If two different cgroups that share an ancestor reserve at the same
time, in different cpus, we would expect to see a more parallel
behavior. Instead, we'll have contention in the ancestor.

It gets even worse if the ancestor is not under pressure, because there
is no reason for the contention. You will cache the leaves, but that
won't help with the intermediate levels.

However, there are some points I admit:

The pressure-behavior with a pure per-cpu proposal is worse, because
then when you are under pressure, you are billing page-by-page, instead
of in bulks.

So maybe what we need is to make the res_counter batched by default - or
provided a res_counter_charge_batched() and
convert memcg for that. Then we can have a cache per res_counter. Two
children of the same res_counter will then both be able to consume their
parent stock, and we can maybe move forward without contention in this case.

> 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;
>>   }
>>
>
>
> 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.

I am starting to think it is not needed as well, specially here.
My goal was to make sure we don't have other per-cpu updaters
when we are trying to grab the final value. But guess that the only
place in which it really
matters is when we do limit testing.

Subject: Re: [RFC 5/7] use percpu_counters for res_counter usage
Posted by Glauber Costa on Fri, 30 Mar 2012 13:53:16 GMT
View Forum Message <> Reply to Message

On 03/30/2012 11:58 AM, KAMEZAWA Hiroyuki wrote:
> ==
>
> Now, we do consume 'reserved' usage, we can avoid css_get(), an heavy atomic
> ops. You may need to move this code as
>
> rcu_read_lock()
> ....
> res_counter_charge()
> if (failure) {
>   css_tryget()
>   rcu_read_unlock()
> } else {
>   rcu_read_unlock()
>   return success;
> }
>
> to compare performance. This css_get() affects performance very very much.

thanks for the tip.

But one thing:

To be sure: it effectively mean that we are drawing from a dead memcg
(because we pre-allocated, right?

Subject: Re: [RFC 5/7] use percpu_counters for res_counter usage
Posted by KAMEZAWA Hiroyuki on Mon, 09 Apr 2012 01:48:42 GMT
View Forum Message <> Reply to Message

(2012/03/30 22:53), Glauber Costa wrote:

> On 03/30/2012 11:58 AM, KAMEZAWA Hiroyuki wrote:
>> ==
>>

>> Now, we do consume 'reserved' usage, we can avoid css_get(), an heavy atomic
>> ops. You may need to move this code as
>>
>> rcu_read_lock()
>> ....
>> res_counter_charge()
>> if (failure) {
>>   css_tryget()
>>   rcu_read_unlock()
>> } else {
>>   rcu_read_unlock()
>>   return success;
>> }
>>
>> to compare performance. This css_get() affects performance very very much.
>
> thanks for the tip.
>
> But one thing:
>
> To be sure: it effectively mean that we are drawing from a dead memcg
> (because we pre-allocated, right?

Cached stock is consumed by the current task. It blocks removal of memcg.
It's not dead.

Thanks,
-Kame
