Subject: [PATCH v4 08/19] Allocate memory for memcg caches whenever a new memcg appears
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:02 GMT
View Forum Message <> Reply to Message

Every cache that is considered a root cache (basically the "original" caches,
tied to the root memcg/no-memcg) will have an array that should be large enough
to store a cache pointer per each memcg in the system.

Theoreticaly, this is as high as 1 << sizeof(css_id), which is currently in the
64k pointers range. Most of the time, we won't be using that much.

What goes in this patch, is a simple scheme to dynamically allocate such an
array, in order to minimize memory usage for memcg caches. Because we would
also like to avoid allocations all the time, at least for now, the array will
only grow. It will tend to be big enough to hold the maximum number of
kmem-limited memcgs ever achieved.

We'll allocate it to be a minimum of 64 kmem-limited memcgs. When we have more
than that, we'll start doubling the size of this array every time the limit is
reached.

Because we are only considering kmem limited memcgs, a natural point for this
to happen is when we write to the limit. At that point, we already have
set_limit_mutex held, so that will become our natural synchronization
mechanism.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>

```
---
 include/linux/memcontrol.h |   2 +
 mm/memcontrol.c            | 171 +++++++++++++++++++++++++++++++++++++++++++++-----
 mm/slab_common.c           |  25 +++++++
 3 files changed, 181 insertions(+), 17 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 957da60..e8d0571 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -420,6 +420,8 @@ int memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *s);
 void memcg_release_cache(struct kmem_cache *cachep);
```

```
 void memcg_cache_list_add(struct mem_cgroup *memcg, struct kmem_cache *cachep);

+int memcg_update_cache_size(struct kmem_cache *s, int num_groups);
+void memcg_update_array_size(int num_groups);
 /**
  * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
  * @gfp: the gfp allocation flags.
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index fff089e..8c5c570 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -371,9 +371,15 @@ static bool memcg_kmem_is_active(struct mem_cgroup *memcg)
  return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
 }

-static void memcg_kmem_set_activated(struct mem_cgroup *memcg)
+static bool memcg_kmem_set_activated(struct mem_cgroup *memcg)
 {
- set_bit(KMEM_ACCOUNTED_ACTIVATED, &memcg->kmem_accounted);
+ return !test_and_set_bit(KMEM_ACCOUNTED_ACTIVATED,
+     &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_clear_activated(struct mem_cgroup *memcg)
+{
+ clear_bit(KMEM_ACCOUNTED_ACTIVATED, &memcg->kmem_accounted);
 }

 static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
@@ -546,6 +552,17 @@ static void disarm_sock_keys(struct mem_cgroup *memcg)
 #endif

 #ifdef CONFIG_MEMCG_KMEM
+static int memcg_limited_groups_array_size;
+#define MEMCG_CACHES_MIN_SIZE 64
+/*
+ * MAX_SIZE should be as large as the number of css_ids. Ideally, we could get
+ * this constant directly from cgroup, but it is understandable that this is
+ * better kept as an internal representation in cgroup.c
+ *
+ * As of right now, this should be enough.
+ */
+#define MEMCG_CACHES_MAX_SIZE 65535
+
 struct static_key memcg_kmem_enabled_key;

 static void disarm_kmem_keys(struct mem_cgroup *memcg)
@@ -2782,6 +2799,115 @@ int memcg_css_id(struct mem_cgroup *memcg)
```

```
  return id;
 }

+/*
+ * This ends up being protected by the set_limit mutex, during normal
+ * operation, because that is its main call site.
+ *
+ * But when we create a new cache, we can call this as well if its parent
+ * is kmem-limited. That will have to hold set_limit_mutex as well.
+ */
+int memcg_update_cache_sizes(struct mem_cgroup *memcg)
+{
+ int num, ret;
+ /*
+  * After this point, kmem_accounted (that we test atomically in
+  * the beginning of this conditional), is no longer 0. This
+  * guarantees only one process will set the following boolean
+  * to true. We don't need test_and_set because we're protected
+  * by the set_limit_mutex anyway.
+  */
+ if (!memcg_kmem_set_activated(memcg))
+  return 0;
+
+ num = memcg_css_id(memcg);
+ ret = memcg_update_all_caches(num);
+ if (ret) {
+  memcg_kmem_clear_activated(memcg);
+  return ret;
+ }
+
+ INIT_LIST_HEAD(&memcg->memcg_slab_caches);
+ mutex_init(&memcg->slab_caches_mutex);
+ return 0;
+}
+
+static size_t memcg_caches_array_size(int num_groups)
+{
+ ssize_t size;
+ if (num_groups <= 0)
+  return 0;
+
+ size = 2 * num_groups;
+ if (size < MEMCG_CACHES_MIN_SIZE)
+  size = MEMCG_CACHES_MIN_SIZE;
+ else if (size > MEMCG_CACHES_MAX_SIZE)
+  size = MEMCG_CACHES_MAX_SIZE;
+
+ return size;
```

```
+}
+
+/*
+ * We should update the current array size iff all caches updates succeed. This
+ * can only be done from the slab side. The slab mutex needs to be held when
+ * calling this.
+ */
+void memcg_update_array_size(int num)
+{
+ if (num > memcg_limited_groups_array_size)
+  memcg_limited_groups_array_size = memcg_caches_array_size(num);
+}
+
+int memcg_update_cache_size(struct kmem_cache *s, int num_groups)
+{
+ struct memcg_cache_params *cur_params = s->memcg_params;
+
+ VM_BUG_ON(s->memcg_params && !s->memcg_params->is_root_cache);
+
+ if (num_groups > memcg_limited_groups_array_size) {
+  int i;
+  ssize_t size = memcg_caches_array_size(num_groups);
+
+  size *= sizeof(void *);
+  size += sizeof(struct memcg_cache_params);
+
+  s->memcg_params = kzalloc(size ,GFP_KERNEL);
+  if (!s->memcg_params) {
+   s->memcg_params = cur_params;
+   return -ENOMEM;
+  }
+
+  s->memcg_params->is_root_cache = true;
+
+  /*
+   * There is the chance it will be bigger than
+   * memcg_limited_groups_array_size, if we failed an allocation
+   * in a cache, in which case all caches updated before it, will
+   * have a bigger array.
+   *
+   * But if that is the case, the data after
+   * memcg_limited_groups_array_size is certainly unused
+   */
+  for (i = 0; memcg_limited_groups_array_size; i++) {
+   if (!cur_params->memcg_caches[i])
+    continue;
+   s->memcg_params->memcg_caches[i] =
+      cur_params->memcg_caches[i];
```

```
+  }
+
+  /*
+   * Ideally, we would wait until all caches succeed, and only
+   * then free the old one. But this is not worth the extra
+   * pointer per-cache we'd have to have for this.
+   *
+   * It is not a big deal if some caches are left with a size
+   * bigger than the others. And all updates will reset this
+   * anyway.
+   */
+  kfree(cur_params);
+ }
+ return 0;
+}
+
 int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s)
 {
  size_t size = sizeof(struct memcg_cache_params);
@@ -2789,6 +2915,9 @@ int memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *s)
  if (!memcg_kmem_enabled())
   return 0;

+ if (!memcg)
+  size =+ memcg_limited_groups_array_size * sizeof(void *);
+
  s->memcg_params = kzalloc(size, GFP_KERNEL);
  if (!s->memcg_params)
   return -ENOMEM;
@@ -4291,14 +4420,11 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
   ret = res_counter_set_limit(&memcg->kmem, val);
   VM_BUG_ON(ret);

-  /*
-   * After this point, kmem_accounted (that we test atomically in
-   * the beginning of this conditional), is no longer 0. This
-   * guarantees only one process will set the following boolean
-   * to true. We don't need test_and_set because we're protected
-   * by the set_limit_mutex anyway.
-   */
-  memcg_kmem_set_activated(memcg);
+  ret = memcg_update_cache_sizes(memcg);
+  if (ret) {
+   res_counter_set_limit(&memcg->kmem, RESOURCE_MAX);
+   goto out;
+  }
   must_inc_static_branch = true;
```

```
  /*
   * kmem charges can outlive the cgroup. In the case of slab
@@ -4337,9 +4463,10 @@ out:
 	return ret;
 }
 
-static void memcg_propagate_kmem(struct mem_cgroup *memcg,
-				 struct mem_cgroup *parent)
+static int memcg_propagate_kmem(struct mem_cgroup *memcg,
+				struct mem_cgroup *parent)
 {
+	int ret = 0;
 	memcg->kmem_accounted = parent->kmem_accounted;
 #ifdef CONFIG_MEMCG_KMEM
 	/*
@@ -4352,11 +4479,19 @@ static void memcg_propagate_kmem(struct mem_cgroup *memcg,
 	 * It is a lot simpler just to do static_key_slow_inc() on every child
 	 * that is accounted.
 	 */
-	if (memcg_kmem_is_active(memcg)) {
-		mem_cgroup_get(memcg);
-		static_key_slow_inc(&memcg_kmem_enabled_key);
-	}
+	if (!memcg_kmem_is_active(memcg))
+		return ret;
+
+	mutex_lock(&set_limit_mutex);
+	ret = memcg_update_cache_sizes(memcg);
+	mutex_unlock(&set_limit_mutex);
+	if (ret)
+		return ret;
+
+	mem_cgroup_get(memcg);
+	static_key_slow_inc(&memcg_kmem_enabled_key);
 #endif
+	return ret;
 }
 
 /*
@@ -5441,8 +5576,10 @@ mem_cgroup_create(struct cgroup *cont)
 		 * This refcnt will be decremented when freeing this
 		 * mem_cgroup(see mem_cgroup_put).
 		 */
+		error = memcg_propagate_kmem(memcg, parent);
+		if (error)
+			goto free_out;
 		mem_cgroup_get(parent);
-		memcg_propagate_kmem(memcg, parent);
```

```
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
diff --git a/mm/slab_common.c b/mm/slab_common.c
index f97f7b8..fcf59d7 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -75,6 +75,31 @@ static inline int kmem_cache_sanity_check(struct mem_cgroup *memcg,
 }
 #endif

+#ifdef CONFIG_MEMCG_KMEM
+int memcg_update_all_caches(int num_memcgs)
+{
+ struct kmem_cache *s;
+ int ret = 0;
+ mutex_lock(&slab_mutex);
+
+ list_for_each_entry(s, &slab_caches, list) {
+  ret = memcg_update_cache_size(s, num_memcgs);
+  /*
+   * See comment in memcontrol.c, memcg_update_cache_size:
+   * Instead of freeing the memory, we'll just leave the caches
+   * up to this point in an updated state.
+   */
+  if (ret)
+   goto out;
+ }
+
+ memcg_update_array_size(num_memcgs);
+out:
+ mutex_unlock(&slab_mutex);
+ return ret;
+}
+#endif
+
 /*
  * kmem_cache_create - Create a cache.
  * @name: A string which is used in /proc/slabinfo to identify this cache.
--
1.7.11.4
```