Subject: [PATCH v5 08/18] memcg: infrastructure to match an allocation to the right cache
Posted by Glauber Costa on Fri, 19 Oct 2012 14:20:32 GMT
View Forum Message <> Reply to Message

The page allocator is able to bind a page to a memcg when it is allocated. But for the caches, we'd like to have as many objects as possible in a page belonging to the same cache.

This is done in this patch by calling memcg_kmem_get_cache in the beginning of every allocation function. This routing is patched out by static branches when kernel memory controller is not being used.

It assumes that the task allocating, which determines the memcg in the page allocator, belongs to the same cgroup throughout the whole process. Misacounting can happen if the task calls memcg_kmem_get_cache() while belonging to a cgroup, and later on changes. This is considered acceptable, and should only happen upon task migration.

Before the cache is created by the memcg core, there is also a possible imbalance: the task belongs to a memcg, but the cache being allocated from is the global cache, since the child cache is not yet guaranteed to be ready. This case is also fine, since in this case the GFP_KMEMCG will not be passed and the page allocator will not attempt any cgroup accounting.

[v4: use a standard workqueue mechanism, create right away if
  possible, index from cache side]

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 |  41 ++++++++++
 init/Kconfig               |   2 +-
 mm/memcontrol.c            | 186 +++++++++++++++++++++++++++++++++++++++++++++++++
 3 files changed, 228 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 491d96c..92fc47a 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -422,6 +422,10 @@ 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);
+
+struct kmem_cache *
+__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
+
 /**
  * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
  * @gfp: the gfp allocation flags.
@@ -491,6 +495,37 @@ memcg_kmem_commit_charge(struct page *page, struct mem_cgroup
*memcg, int order)
   __memcg_kmem_commit_charge(page, memcg, order);
 }

+/**
+ * memcg_kmem_get_cache: selects the correct per-memcg cache for allocation
+ * @cachep: the original global kmem cache
+ * @gfp: allocation flags.
+ *
+ * This function assumes that the task allocating, which determines the memcg
+ * in the page allocator, belongs to the same cgroup throughout the whole
+ * process.  Misacounting can happen if the task calls memcg_kmem_get_cache()
+ * while belonging to a cgroup, and later on changes. This is considered
+ * acceptable, and should only happen upon task migration.
+ *
+ * Before the cache is created by the memcg core, there is also a possible
+ * imbalance: the task belongs to a memcg, but the cache being allocated from
+ * is the global cache, since the child cache is not yet guaranteed to be
+ * ready. This case is also fine, since in this case the GFP_KMEMCG will not be
+ * passed and the page allocator will not attempt any cgroup accounting.
+ */
+static __always_inline struct kmem_cache *
+memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ if (!memcg_kmem_enabled())
+  return cachep;
+ if (gfp & __GFP_NOFAIL)
+  return cachep;
+ if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
+  return cachep;
+ if (unlikely(fatal_signal_pending(current)))
+  return cachep;
+
+ return __memcg_kmem_get_cache(cachep, gfp);
+}
 #else
```

```
 static inline void sock_update_memcg(struct sock *sk)
 {
@@ -529,6 +564,12 @@ static inline void memcg_cache_list_add(struct mem_cgroup *memcg,
 {
  BUG();
 }
+
+static inline struct kmem_cache *
+memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ return cachep;
+}
 #endif /* CONFIG_MEMCG_KMEM */
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/init/Kconfig b/init/Kconfig
index af6c7f8..62b1f28 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -741,7 +741,7 @@ config MEMCG_SWAP_ENABLED
    then swapaccount=0 does the trick).
 config MEMCG_KMEM
  bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
- depends on MEMCG && EXPERIMENTAL
+ depends on MEMCG && EXPERIMENTAL && !SLOB
  default n
  help
    The Kernel Memory extension for Memory Resource Controller can limit
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index dd6ac6a..ac2e621 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -562,7 +562,14 @@ static int memcg_limited_groups_array_size;
  */
 #define MEMCG_CACHES_MAX_SIZE 65535

+/*
+ * A lot of the calls to the cache allocation functions are expected to be
+ * inlined by the compiler. Since the calls to memcg_kmem_get_cache are
+ * conditional to this static branch, we'll have to allow modules that does
+ * kmem_cache_alloc and the such to see this symbol as well
+ */
 struct static_key memcg_kmem_enabled_key;
+EXPORT_SYMBOL(memcg_kmem_enabled_key);

 static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
@@ -2930,9 +2937,188 @@ int memcg_register_cache(struct mem_cgroup *memcg, struct
```

```
kmem_cache *s)

 void memcg_release_cache(struct kmem_cache *s)
 {
+ struct kmem_cache *root;
+ int id = memcg_css_id(s->memcg_params->memcg);
+
+ if (s->memcg_params->is_root_cache)
+  goto out;
+
+ root = s->memcg_params->root_cache;
+ root->memcg_params->memcg_caches[id] = NULL;
+ mem_cgroup_put(s->memcg_params->memcg);
+out:
  kfree(s->memcg_params);
 }

+static char *memcg_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+ char *name;
+ struct dentry *dentry;
+
+ rcu_read_lock();
+ dentry = rcu_dereference(memcg->css.cgroup->dentry);
+ rcu_read_unlock();
+
+ BUG_ON(dentry == NULL);
+
+ name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
+     cachep->name, css_id(&memcg->css), dentry->d_name.name);
+
+ return name;
+}
+
+static struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
+      struct kmem_cache *s)
+{
+ char *name;
+ struct kmem_cache *new;
+
+ name = memcg_cache_name(memcg, s);
+ if (!name)
+  return NULL;
+
+ new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
+          (s->flags & ~SLAB_PANIC), s->ctor);
+
+ kfree(name);
```

```
+ return new;
+}
+
+/*
+ * This lock protects updaters, not readers. We want readers to be as fast as
+ * they can, and they will either see NULL or a valid cache value. Our model
+ * allow them to see NULL, in which case the root memcg will be selected.
+ *
+ * We need this lock because multiple allocations to the same cache from a non
+ * GFP_WAIT area will span more than one worker. Only one of them can create
+ * the cache.
+ */
+static DEFINE_MUTEX(memcg_cache_mutex);
+static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
+        struct kmem_cache *cachep)
+{
+ struct kmem_cache *new_cachep;
+ int idx;
+
+ BUG_ON(!memcg_can_account_kmem(memcg));
+
+ idx = memcg_css_id(memcg);
+
+ mutex_lock(&memcg_cache_mutex);
+ new_cachep = cachep->memcg_params->memcg_caches[idx];
+ if (new_cachep)
+  goto out;
+
+ new_cachep = kmem_cache_dup(memcg, cachep);
+
+ if (new_cachep == NULL) {
+  new_cachep = cachep;
+  goto out;
+ }
+
+ mem_cgroup_get(memcg);
+ cachep->memcg_params->memcg_caches[idx] = new_cachep;
+ wmb(); /* the readers won't lock, make sure everybody sees it */
+ new_cachep->memcg_params->memcg = memcg;
+ new_cachep->memcg_params->root_cache = cachep;
+out:
+ mutex_unlock(&memcg_cache_mutex);
+ return new_cachep;
+}
+
+struct create_work {
+ struct mem_cgroup *memcg;
+ struct kmem_cache *cachep;
```

```
+ struct work_struct work;
+};
+
+static void memcg_create_cache_work_func(struct work_struct *w)
+{
+ struct create_work *cw;
+
+ cw = container_of(w, struct create_work, work);
+ memcg_create_kmem_cache(cw->memcg, cw->cachep);
+ /* Drop the reference gotten when we enqueued. */
+ css_put(&cw->memcg->css);
+ kfree(cw);
+}
+
+/*
+ * Enqueue the creation of a per-memcg kmem_cache.
+ * Called with rcu_read_lock.
+ */
+static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+          struct kmem_cache *cachep)
+{
+ struct create_work *cw;
+
+ cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ if (cw == NULL)
+  return;
+
+ /* The corresponding put will be done in the workqueue. */
+ if (!css_tryget(&memcg->css))
+  return;
+
+ cw->memcg = memcg;
+ cw->cachep = cachep;
+
+ INIT_WORK(&cw->work, memcg_create_cache_work_func);
+ schedule_work(&cw->work);
+}
+
+/*
+ * Return the kmem_cache we're supposed to use for a slab allocation.
+ * We try to use the current memcg's version of the cache.
+ *
+ * If the cache does not exist yet, if we are the first user of it,
+ * we either create it immediately, if possible, or create it asynchronously
+ * in a workqueue.
+ * In the latter case, we will let the current allocation go through with
+ * the original cache.
+ *
```

```
+ * Can't be called in interrupt context or from kernel threads.
+ * This function needs to be called with rcu_read_lock() held.
+ */
+struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep,
+       gfp_t gfp)
+{
+ struct mem_cgroup *memcg;
+ int idx;
+
+ if (cachep->memcg_params && cachep->memcg_params->memcg)
+  return cachep;
+
+ rcu_read_lock();
+ memcg = mem_cgroup_from_task(rcu_dereference(current->mm->owner));
+ rcu_read_unlock();
+
+ if (!memcg_can_account_kmem(memcg))
+  return cachep;
+
+ idx = memcg_css_id(memcg);
+ VM_BUG_ON(idx == -1);
+
+ if (cachep->memcg_params->memcg_caches[idx] == NULL) {
+  /*
+   * If we are in a safe context, better to be predictable and
+   * return right away. This guarantees that the allocation being
+   * performed already belongs in the new cache. When we enqueue,
+   * we can't do that, and at least the current allocation will
+   * be relayed to the root cache while our cache is being
+   * created.
+   */
+  if ((gfp & __GFP_WAIT) && !in_interrupt())
+   return memcg_create_kmem_cache(memcg, cachep);
+
+  memcg_create_cache_enqueue(memcg, cachep);
+  return cachep;
+ }
+
+ return cachep->memcg_params->memcg_caches[idx];
+}
+EXPORT_SYMBOL(__memcg_kmem_get_cache);
+
 /*
  * We need to verify if the allocation against current->mm->owner's memcg is
  * possible for the given order. But the page is not allocated yet, so we'll
--
1.7.11.7
```