Subject: [RFC 0/4] bypass charges if memcg is not used
Posted by Glauber Costa on Tue, 25 Sep 2012 08:52:49 GMT
View Forum Message <> Reply to Message

For the special case that memcg is compiled in (a quite common case) but not in use (also a common case), we are still seeing a huge performance impact. Mel Gorman demonstrated in the Kernel Summit, mm mini summit, that depending on the hardware, this can vary between 6 and 15 %.

I am proposing in this initial patch, that we take the strategy of bypassing the memcg code with static branches. We can flip it on when the first memcg gets created. Up to that moment, we'll mostly have a bunch of nops.

This patch also defers the call to page_cgroup_init() to that moment. This means that the memory used by page_cgroup structure won't be wasted until we really need it. We can do the same with the memory used for swap, if needed.

There are many edges to be trimmed, but I wanted to send this early to collect feedback. I coded this enough to get numbers out of it. I tested it in a 24-way 2-socket Intel box, 24 Gb mem. I used Mel Gorman's pft test, that he used to demonstrate this problem back in the Kernel Summit. There are three kernels:

nomemcg : memcg compile disabled.
base : memcg enabled, patch not applied.
bypassed : memcg enabled, with patch applied.

	base	bypassed
User	109.12	105.64
System	1646.84	1597.98
Elapsed	229.56	215.76

	nomemcg	bypassed
User	104.35	105.64
System	1578.19	1597.98
Elapsed	212.33	215.76

So as one can see, the difference between base and nomemcg in terms of both system time and elapsed time is quite drastic, and consistent with the figures shown by Mel Gorman in the Kernel summit. This is a ~ 7 % drop in performance, just by having memcg enabled. memcg functions appear heavily in the profiles, even if all tasks lives in the root memcg.

With bypassed kernel, we drop this down to 1.5 %, which starts to fall in the acceptable range. More investigation is needed to see if we can claim that last percent back, but I believe at last part of it should be.

Glauber Costa (4):
  memcg: provide root figures from system totals
  memcg: make it suck faster
  memcg: do not call page_cgroup_init at system_boot
  memcg: do not walk all the way to the root for memcg

 include/linux/memcontrol.h | 56 ++++++++++++++++++++++++++------
 include/linux/page_cgroup.h | 20 +++++++---
 init/main.c | 1 -
 mm/memcontrol.c | 78 ++-----
 mm/page_cgroup.c | 44 ++++++++++++++++++++++++++
 5 files changed, 165 insertions(+), 34 deletions(-)

--
1.7.11.4

Subject: [RFC 1/4] memcg: provide root figures from system totals
Posted by Glauber Costa on Tue, 25 Sep 2012 08:52:50 GMT
View Forum Message <> Reply to Message

For the root memcg, there is no need to rely on the res_counters.
The sum of all mem cgroups plus the tasks in root itself, is necessarily
the amount of memory used for the whole system. Since those figures are
already kept somewhere anyway, we can just return them here, without too
much hassle.

The limit can't be set for the root cgroup, so it is left at 0. Same is
true for failcnt, because its actual meaning is how many times we failed
allocations due to the limit being hit. We will fail allocations in the
root cgroup, but the limit will never the reason.

TODO includes figuring out what to do with the soft limit and max_usage.
Comments and suggestions appreciated.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Mel Gorman <mgorman@suse.de>
CC: Andrew Morton <akpm@linux-foundation.org>
---
 mm/memcontrol.c | 45 +++
 1 file changed, 45 insertions(+)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 82c5b8f..bac398b 100644

```
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -4506,6 +4506,45 @@ static inline u64 mem_cgroup_usage(struct mem_cgroup *memcg,
bool swap)
  return val << PAGE_SHIFT;
 }

+static u64 mem_cgroup_read_root(struct mem_cgroup *memcg, enum res_type type, int name)
+{
+ struct sysinfo i;
+        unsigned long pages[NR_LRU_LISTS];
+        int lru;
+
+ si_meminfo(&i);
+ si_swapinfo(&i);
+
+ if (name == RES_LIMIT)
+  return RESOURCE_MAX;
+ if (name == RES_SOFT_LIMIT)
+  return 0;
+ if (name == RES_FAILCNT)
+  return 0;
+ if (name == RES_MAX_USAGE)
+  return 0;
+
+ if (WARN_ON_ONCE(name != RES_USAGE))
+  return 0;
+
+ for (lru = LRU_BASE; lru < NR_LRU_LISTS; lru++)
+  pages[lru] = global_page_state(NR_LRU_BASE + lru);
+
+ switch (type) {
+ case _MEM:
+                return pages[LRU_ACTIVE_ANON]   + pages[LRU_ACTIVE_FILE] +
+   pages[LRU_INACTIVE_ANON] + pages[LRU_INACTIVE_FILE];
+ case _MEMSWAP:
+                return pages[LRU_ACTIVE_ANON]   + pages[LRU_ACTIVE_FILE] +
+   pages[LRU_INACTIVE_ANON] + pages[LRU_INACTIVE_FILE] +
+   i.totalswap - i.freeswap;
+ case _KMEM:
+  return 0;
+ default:
+  BUG();
+ };
+}
+
 static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
         struct file *file, char __user *buf,
```

```
          size_t nbytes, loff_t *ppos)
@@ -4522,6 +4561,11 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
  if (!do_swap_account && type == _MEMSWAP)
   return -EOPNOTSUPP;

+ if (mem_cgroup_is_root(memcg)) {
+  val = mem_cgroup_read_root(memcg, type, name);
+  goto root_bypass;
+ }
+
  switch (type) {
  case _MEM:
   if (name == RES_USAGE)
@@ -4542,6 +4586,7 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
   BUG();
  }

+root_bypass:
  len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
  return simple_read_from_buffer(buf, nbytes, ppos, str, len);
 }
--
1.7.11.4
```

Subject: [RFC 2/4] memcg: make it suck faster
Posted by Glauber Costa on Tue, 25 Sep 2012 08:52:51 GMT
View Forum Message <> Reply to Message

It is an accepted fact that memcg sucks. But can it suck faster? Or in
a more fair statement, can it at least stop draining everyone's
performance when it is not in use?

This experimental and slightly crude patch demonstrates that we can do
that by using static branches to patch it out until the first memcg
comes to life. There are edges to be trimmed, and I appreciate comments
for direction. In particular, the events in the root are not fired, but
I believe this can be done without further problems by calling a
specialized event check from mem_cgroup_newpage_charge().

My goal was to have enough numbers to demonstrate the performance gain
that can come from it. I tested it in a 24-way 2-socket Intel box, 24 Gb
mem. I used Mel Gorman's pft test, that he used to demonstrate this
problem back in the Kernel Summit. There are three kernels:

nomemcg : memcg compile disabled.
base : memcg enabled, patch not applied.

bypassed : memcg enabled, with patch applied.

	base	bypassed
User	109.12	105.64
System	1646.84	1597.98
Elapsed	229.56	215.76

	nomemcg	bypassed
User	104.35	105.64
System	1578.19	1597.98
Elapsed	212.33	215.76

So as one can see, the difference between base and nomemcg in terms
of both system time and elapsed time is quite drastic, and consistent
with the figures shown by Mel Gorman in the Kernel summit. This is a
~ 7 % drop in performance, just by having memcg enabled. memcg functions
appear heavily in the profiles, even if all tasks lives in the root
memcg.

With bypassed kernel, we drop this down to 1.5 %, which starts to fall
in the acceptable range. More investigation is needed to see if we can
claim that last percent back, but I believe at last part of it should
be.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Mel Gorman <mgorman@suse.de>
CC: Andrew Morton <akpm@linux-foundation.org>
---
 include/linux/memcontrol.h | 56 ++++++++++++++++++++++++++++++++---------
 mm/memcontrol.c | 26 ++++++++++++++++++++++++------
 mm/page_cgroup.c | 4 ++--
 3 files changed, 67 insertions(+), 19 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 6d5e212..9dabe61 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -42,6 +42,23 @@ struct mem_cgroup_reclaim_cookie {
 };

 #ifdef CONFIG_MEMCG
+extern struct static_key memcg_in_use_key;
+
+static inline bool mem_cgroup_subsys_disabled(void)
+{

```
+ return !!mem_cgroup_subsys.disabled;
+}
+
+static inline bool mem_cgroup_disabled(void)
+{
+ if (!static_key_false(&memcg_in_use_key))
+  return true;
+ if (mem_cgroup_subsys_disabled())
+  return true;
+ return false;
+}
+
+
 /*
  * All "charge" functions with gfp_mask should use GFP_KERNEL or
  * (gfp_mask & GFP_RECLAIM_MASK). In current implementatin, memcg doesn't
@@ -53,8 +70,18 @@ struct mem_cgroup_reclaim_cookie {
  * (Of course, if memcg does memory allocation in future, GFP_KERNEL is sane.)
  */

-extern int mem_cgroup_newpage_charge(struct page *page, struct mm_struct *mm,
+extern int __mem_cgroup_newpage_charge(struct page *page, struct mm_struct *mm,
     gfp_t gfp_mask);
+
+static inline int
+mem_cgroup_newpage_charge(struct page *page, struct mm_struct *mm,
+     gfp_t gfp_mask)
+{
+ if (mem_cgroup_disabled())
+  return 0;
+ return __mem_cgroup_newpage_charge(page, mm, gfp_mask);
+}
+
 /* for swap handling */
 extern int mem_cgroup_try_charge_swapin(struct mm_struct *mm,
   struct page *page, gfp_t mask, struct mem_cgroup **memcgp);
@@ -66,7 +93,15 @@ extern int mem_cgroup_cache_charge(struct page *page, struct
mm_struct *mm,
      gfp_t gfp_mask);

 struct lruvec *mem_cgroup_zone_lruvec(struct zone *, struct mem_cgroup *);
-struct lruvec *mem_cgroup_page_lruvec(struct page *, struct zone *);
+struct lruvec *__mem_cgroup_page_lruvec(struct page *, struct zone *);
+
+static inline struct lruvec *
+mem_cgroup_page_lruvec(struct page *page, struct zone *zone)
+{
+ if (mem_cgroup_disabled())
```

```
+  return &zone->lruvec;
+ return __mem_cgroup_page_lruvec(page, zone);
+}

 /* For coalescing uncharge for reducing memcg' overhead*/
 extern void mem_cgroup_uncharge_start(void);
@@ -129,13 +164,6 @@ extern void mem_cgroup_replace_page_cache(struct page *oldpage,
 extern int do_swap_account;
 #endif

-static inline bool mem_cgroup_disabled(void)
-{
- if (mem_cgroup_subsys.disabled)
-  return true;
- return false;
-}
-
 void __mem_cgroup_begin_update_page_stat(struct page *page, bool *locked,
       unsigned long *flags);

@@ -184,7 +212,15 @@ unsigned long mem_cgroup_soft_limit_reclaim(struct zone *zone, int
order,
       gfp_t gfp_mask,
       unsigned long *total_scanned);

-void mem_cgroup_count_vm_event(struct mm_struct *mm, enum vm_event_item idx);
+void __mem_cgroup_count_vm_event(struct mm_struct *mm, enum vm_event_item idx);
+
+static inline void mem_cgroup_count_vm_event(struct mm_struct *mm, enum vm_event_item
idx)
+{
+ if (mem_cgroup_disabled())
+  return;
+ __mem_cgroup_count_vm_event(mm, idx);
+}
+
 #ifdef CONFIG_TRANSPARENT_HUGEPAGE
 void mem_cgroup_split_huge_fixup(struct page *head);
 #endif
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index bac398b..a2c88c4 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -472,6 +472,8 @@ struct mem_cgroup *mem_cgroup_from_css(struct cgroup_subsys_state
*s)
  return container_of(s, struct mem_cgroup, css);
 }
```

```
+struct static_key memcg_in_use_key;
+
 /* Writing them here to avoid exposing memcg's inner layout */
 #ifdef CONFIG_MEMCG_KMEM
 #include <net/sock.h>
@@ -1446,12 +1448,19 @@ static void memcg_check_events(struct mem_cgroup *memcg,
struct page *page)

 struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
 {
+ if (mem_cgroup_disabled())
+  return root_mem_cgroup;
+
  return mem_cgroup_from_css(
   cgroup_subsys_state(cont, mem_cgroup_subsys_id));
 }

 struct mem_cgroup *mem_cgroup_from_task(struct task_struct *p)
 {
+
+ if (mem_cgroup_disabled())
+  return root_mem_cgroup;
+
  /*
   * mm_update_next_owner() may clear mm->owner to NULL
   * if it races with swapoff, page migration, etc.
@@ -1599,7 +1608,7 @@ static inline bool mem_cgroup_is_root(struct mem_cgroup *memcg)
  return (memcg == root_mem_cgroup);
 }

-void mem_cgroup_count_vm_event(struct mm_struct *mm, enum vm_event_item idx)
+void __mem_cgroup_count_vm_event(struct mm_struct *mm, enum vm_event_item idx)
 {
  struct mem_cgroup *memcg;

@@ -1666,15 +1675,12 @@ struct lruvec *mem_cgroup_zone_lruvec(struct zone *zone,
  * @page: the page
  * @zone: zone of the page
  */
-struct lruvec *mem_cgroup_page_lruvec(struct page *page, struct zone *zone)
+struct lruvec *__mem_cgroup_page_lruvec(struct page *page, struct zone *zone)
 {
  struct mem_cgroup_per_zone *mz;
  struct mem_cgroup *memcg;
  struct page_cgroup *pc;

- if (mem_cgroup_disabled())
-  return &zone->lruvec;
```

```
-
  pc = lookup_page_cgroup(page);
  memcg = pc->mem_cgroup;

@@ -1686,8 +1692,11 @@ struct lruvec *mem_cgroup_page_lruvec(struct page *page, struct
zone *zone)
   * Our caller holds lru_lock, and PageCgroupUsed is updated
   * under page_cgroup lock: between them, they make all uses
   * of pc->mem_cgroup safe.
+  *
+  * XXX: We can now be LRU and memcg == 0. Scanning all pages on cgroup init
+  * is too expensive. We can ultimately pay,  but prefer to do it here.
   */
- if (!PageLRU(page) && !PageCgroupUsed(pc) && memcg != root_mem_cgroup)
+ if (!PageCgroupUsed(pc) && memcg != root_mem_cgroup)
   pc->mem_cgroup = memcg = root_mem_cgroup;

  mz = page_cgroup_zoneinfo(memcg, page);
@@ -3374,7 +3383,7 @@ static int mem_cgroup_charge_common(struct page *page, struct
mm_struct *mm,
  return 0;
 }

-int mem_cgroup_newpage_charge(struct page *page,
+int __mem_cgroup_newpage_charge(struct page *page,
         struct mm_struct *mm, gfp_t gfp_mask)
 {
  if (mem_cgroup_disabled())
@@ -5703,6 +5712,8 @@ mem_cgroup_create(struct cgroup *cont)
   parent = mem_cgroup_from_cont(cont->parent);
   memcg->use_hierarchy = parent->use_hierarchy;
   memcg->oom_kill_disable = parent->oom_kill_disable;
+
+  static_key_slow_inc(&memcg_in_use_key);
  }

  if (parent && parent->use_hierarchy) {
@@ -5777,6 +5788,7 @@ static void mem_cgroup_destroy(struct cgroup *cont)

  kmem_cgroup_destroy(memcg);

+ static_key_slow_dec(&memcg_in_use_key);
  mem_cgroup_put(memcg);
 }

diff --git a/mm/page_cgroup.c b/mm/page_cgroup.c
index 5ddad0c..5699e9f 100644
--- a/mm/page_cgroup.c
```

```
+++ b/mm/page_cgroup.c
@@ -68,7 +68,7 @@ void __init page_cgroup_init_flatmem(void)

  int nid, fail;

- if (mem_cgroup_disabled())
+ if (mem_cgroup_subsys_disabled())
    return;

  for_each_online_node(nid)  {
@@ -268,7 +268,7 @@ void __init page_cgroup_init(void)
  unsigned long pfn;
  int nid;

- if (mem_cgroup_disabled())
+ if (mem_cgroup_subsys_disabled())
    return;

  for_each_node_state(nid, N_HIGH_MEMORY) {
--
1.7.11.4
```

Subject: [RFC 3/4] memcg: do not call page_cgroup_init at system_boot
Posted by Glauber Costa on Tue, 25 Sep 2012 08:52:52 GMT
View Forum Message <> Reply to Message

If we are not using memcg, there is no reason why we should allocate
this structure, that will be a memory waste at best. We can do better
at least in the sparsemem case, and allocate it when the first cgroup
is requested. It should now not panic on failure, and we have to handle
this right.

flatmem case is a bit more complicated, so that one is left out for
the moment.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Mel Gorman <mgorman@suse.de>
CC: Andrew Morton <akpm@linux-foundation.org>
---
 include/linux/page_cgroup.h | 20 ++++++++++++++-----
 init/main.c | 1 -
 mm/memcontrol.c | 5 +++++
 mm/page_cgroup.c | 40 ++++++++++++++++++++++++++++++++--------
 4 files changed, 52 insertions(+), 14 deletions(-)

```
diff --git a/include/linux/page_cgroup.h b/include/linux/page_cgroup.h
index 777a524..46dd0e7 100644
--- a/include/linux/page_cgroup.h
+++ b/include/linux/page_cgroup.h
@@ -27,16 +27,21 @@ struct page_cgroup {
  struct mem_cgroup *mem_cgroup;
 };

-void __meminit pgdat_page_cgroup_init(struct pglist_data *pgdat);
+void pgdat_page_cgroup_init(struct pglist_data *pgdat);

 #ifdef CONFIG_SPARSEMEM
 static inline void __init page_cgroup_init_flatmem(void)
 {
 }
-extern void __init page_cgroup_init(void);
+extern int page_cgroup_init(void);
+extern void page_cgroup_destroy(void);
 #else
 void __init page_cgroup_init_flatmem(void);
-static inline void __init page_cgroup_init(void)
+static inline int page_cgroup_init(void)
+{
+}
+
+static inline void page_cgroup_destroy(void)
 {
 }
 #endif
@@ -85,7 +90,7 @@ static inline void unlock_page_cgroup(struct page_cgroup *pc)
 #else /* CONFIG_MEMCG */
 struct page_cgroup;

-static inline void __meminit pgdat_page_cgroup_init(struct pglist_data *pgdat)
+static inline void pgdat_page_cgroup_init(struct pglist_data *pgdat)
 {
 }

@@ -94,7 +99,12 @@ static inline struct page_cgroup *lookup_page_cgroup(struct page *page)
  return NULL;
 }

-static inline void page_cgroup_init(void)
+static inline int page_cgroup_init(void)
+{
+ return 0;
+}
```

```
+
+static inline void page_cgroup_destroy(void)
 {
 }

diff --git a/init/main.c b/init/main.c
index 09cf339..9c48d1c 100644
--- a/init/main.c
+++ b/init/main.c
@@ -588,7 +588,6 @@ asmlinkage void __init start_kernel(void)
   initrd_start = 0;
  }
 #endif
- page_cgroup_init();
  debug_objects_mem_init();
  kmemleak_init();
  setup_per_cpu_pageset();
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index a2c88c4..f8115f0 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5709,6 +5709,10 @@ mem_cgroup_create(struct cgroup *cont)
   }
   hotcpu_notifier(memcg_cpu_hotplug_callback, 0);
  } else {
+  /* FIXME: Need to clean this up properly if fails */
+  if (!page_cgroup_init())
+   goto free_out;
+
   parent = mem_cgroup_from_cont(cont->parent);
   memcg->use_hierarchy = parent->use_hierarchy;
   memcg->oom_kill_disable = parent->oom_kill_disable;
@@ -5789,6 +5793,7 @@ static void mem_cgroup_destroy(struct cgroup *cont)
  kmem_cgroup_destroy(memcg);

  static_key_slow_dec(&memcg_in_use_key);
+ page_cgroup_destroy();
  mem_cgroup_put(memcg);
 }

diff --git a/mm/page_cgroup.c b/mm/page_cgroup.c
index 5699e9f..ee5738f 100644
--- a/mm/page_cgroup.c
+++ b/mm/page_cgroup.c
@@ -16,7 +16,7 @@ static unsigned long total_usage;
 #if !defined(CONFIG_SPARSEMEM)
```

```
-void __meminit pgdat_page_cgroup_init(struct pglist_data *pgdat)
+void pgdat_page_cgroup_init(struct pglist_data *pgdat)
 {
  pgdat->node_page_cgroup = NULL;
 }
@@ -42,7 +42,7 @@ struct page_cgroup *lookup_page_cgroup(struct page *page)
  return base + offset;
 }

-static int __init alloc_node_page_cgroup(int nid)
+static int alloc_node_page_cgroup(int nid)
 {
  struct page_cgroup *base;
  unsigned long table_size;
@@ -105,7 +105,7 @@ struct page_cgroup *lookup_page_cgroup(struct page *page)
  return section->page_cgroup + pfn;
 }

-static void *__meminit alloc_page_cgroup(size_t size, int nid)
+static void *alloc_page_cgroup(size_t size, int nid)
 {
  gfp_t flags = GFP_KERNEL | __GFP_ZERO | __GFP_NOWARN;
  void *addr = NULL;
@@ -124,7 +124,7 @@ static void *__meminit alloc_page_cgroup(size_t size, int nid)
  return addr;
 }

-static int __meminit init_section_page_cgroup(unsigned long pfn, int nid)
+static int init_section_page_cgroup(unsigned long pfn, int nid)
 {
  struct mem_section *section;
  struct page_cgroup *base;
@@ -263,7 +263,9 @@ static int __meminit page_cgroup_callback(struct notifier_block *self,

 #endif

-void __init page_cgroup_init(void)
+static atomic_t page_cgroup_initialized = ATOMIC_INIT(0);
+
+int page_cgroup_init(void)
 {
  unsigned long pfn;
  int nid;
@@ -271,6 +273,12 @@ void __init page_cgroup_init(void)
  if (mem_cgroup_subsys_disabled())
   return;

+ if (atomic_add_return(1, &page_cgroup_initialized) != 1)
```

```
+  return 0;
+
+ /* We can arrive here multiple times, if memcgs come and go. */
+ total_usage = 0;
+
  for_each_node_state(nid, N_HIGH_MEMORY) {
   unsigned long start_pfn, end_pfn;

@@ -306,16 +314,32 @@ void __init page_cgroup_init(void)
  return;
 oom:
  printk(KERN_CRIT "try 'cgroup_disable=memory' boot option\n");
- panic("Out of memory");
+ return -ENOMEM;
 }

-void __meminit pgdat_page_cgroup_init(struct pglist_data *pgdat)
+void pgdat_page_cgroup_init(struct pglist_data *pgdat)
 {
  return;
 }

-#endif
+void page_cgroup_destroy(void)
+{
+ int nid;

+ if (atomic_sub_return(1, &page_cgroup_initialized))
+  return;
+
+ for_each_node_state(nid, N_HIGH_MEMORY) {
+  unsigned long start, end, pfn;
+
+  start = node_start_pfn(nid);
+  end = node_end_pfn(nid);
+
+  for (pfn = start; pfn < end; pfn += PAGES_PER_SECTION)
+   __free_page_cgroup(pfn);
+ }
+}
+#endif

 #ifdef CONFIG_MEMCG_SWAP

--
1.7.11.4
```

Subject: [RFC 4/4] memcg: do not walk all the way to the root for memcg
Posted by Glauber Costa on Tue, 25 Sep 2012 08:52:53 GMT
View Forum Message <> Reply to Message

Since the root is special anyway, and we always get its figures from
global counters anyway, there is no make all cgroups its descendants,
wrt res_counters. The sad effect of doing that is that we need to lock
the root for all allocations, since it is a common ancestor of
everybody.

Not having the root as a common ancestor should lead to better
scalability for not-uncommon case of tasks in the cgroup being
node-bound to different nodes in NUMA systems.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Mel Gorman <mgorman@suse.de>
CC: Andrew Morton <akpm@linux-foundation.org>
---
 mm/memcontrol.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index f8115f0..829ea9e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5720,7 +5720,7 @@ mem_cgroup_create(struct cgroup *cont)
   static_key_slow_inc(&memcg_in_use_key);
  }

- if (parent && parent->use_hierarchy) {
+ if (parent && !mem_cgroup_is_root(parent) && parent->use_hierarchy) {
   struct mem_cgroup __maybe_unused *p;

   res_counter_init(&memcg->res, &parent->res);
--
1.7.11.4

Subject: Re: [RFC 2/4] memcg: make it suck faster
Posted by akpm on Tue, 25 Sep 2012 21:02:36 GMT
View Forum Message <> Reply to Message

On Tue, 25 Sep 2012 12:52:51 +0400
Glauber Costa <glommer@parallels.com> wrote:

> It is an accepted fact that memcg sucks. But can it suck faster? Or in
> a more fair statement, can it at least stop draining everyone's
> performance when it is not in use?
>
> This experimental and slightly crude patch demonstrates that we can do
> that by using static branches to patch it out until the first memcg
> comes to life. There are edges to be trimmed, and I appreciate comments
> for direction. In particular, the events in the root are not fired, but
> I believe this can be done without further problems by calling a
> specialized event check from mem_cgroup_newpage_charge().
>
> My goal was to have enough numbers to demonstrate the performance gain
> that can come from it. I tested it in a 24-way 2-socket Intel box, 24 Gb
> mem. I used Mel Gorman's pft test, that he used to demonstrate this
> problem back in the Kernel Summit. There are three kernels:
>
> nomemcg : memcg compile disabled.
> base   : memcg enabled, patch not applied.
> bypassed : memcg enabled, with patch applied.
>
>   base   bypassed
> User   109.12   105.64
> System   1646.84   1597.98
> Elapsed   229.56   215.76
>
>   nomemcg   bypassed
> User   104.35   105.64
> System   1578.19   1597.98
> Elapsed   212.33   215.76
>
> So as one can see, the difference between base and nomemcg in terms
> of both system time and elapsed time is quite drastic, and consistent
> with the figures shown by Mel Gorman in the Kernel summit. This is a
> ~ 7 % drop in performance, just by having memcg enabled. memcg functions
> appear heavily in the profiles, even if all tasks lives in the root
> memcg.
>
> With bypassed kernel, we drop this down to 1.5 %, which starts to fall
> in the acceptable range. More investigation is needed to see if we can
> claim that last percent back, but I believe at last part of it should
> be.

Well that's encouraging. I wonder how many users will actually benefit
from this - did I hear that major distros are now using memcg in some
system-infrastructure-style code?

iirc, the idea of disabling memcg operations until someone enables a
container had a couple of problems:

a) certain boot-time initialisation issn't performed and

b) when memcg starts running for real, it expects that certain stats
   gathering has been running since boot. If this is not the case,
   those stats are wrong and stuff breaks.

It would be helpful if you could summarise these and similar issues
and describe how they were addressed.

>
> ...
>
> struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
> {
> + if (mem_cgroup_disabled())
> + return root_mem_cgroup;

There would be some benefit in inlining the above instructions into the
caller.

>   return mem_cgroup_from_css(
>   cgroup_subsys_state(cont, mem_cgroup_subsys_id));
> }

In fact the entire mem_cgroup_from_cont() could be inlined.

> struct mem_cgroup *mem_cgroup_from_task(struct task_struct *p)
> {
> +
> + if (mem_cgroup_disabled())
> + return root_mem_cgroup;

Ditto.

>   /*
>   * mm_update_next_owner() may clear mm->owner to NULL
>   * if it races with swapoff, page migration, etc.
>
> ...
>

Subject: Re: [RFC 2/4] memcg: make it suck faster
Posted by Glauber Costa on Wed, 26 Sep 2012 08:53:21 GMT
View Forum Message <> Reply to Message

On 09/26/2012 01:02 AM, Andrew Morton wrote:

>> nomemcg : memcg compile disabled.
>> > base   : memcg enabled, patch not applied.
>> > bypassed : memcg enabled, with patch applied.
>> >
>> >   base   bypassed
>> > User   109.12   105.64
>> > System   1646.84   1597.98
>> > Elapsed   229.56   215.76
>> >
>> >   nomemcg   bypassed
>> > User   104.35   105.64
>> > System   1578.19   1597.98
>> > Elapsed   212.33   215.76
>> >
>> > So as one can see, the difference between base and nomemcg in terms
>> > of both system time and elapsed time is quite drastic, and consistent
>> > with the figures shown by Mel Gorman in the Kernel summit. This is a
>> > ~ 7 % drop in performance, just by having memcg enabled. memcg functions
>> > appear heavily in the profiles, even if all tasks lives in the root
>> > memcg.
>> >
>> > With bypassed kernel, we drop this down to 1.5 %, which starts to fall
>> > in the acceptable range. More investigation is needed to see if we can
>> > claim that last percent back, but I believe at last part of it should
>> > be.
> Well that's encouraging. I wonder how many users will actually benefit
> from this - did I hear that major distros are now using memcg in some
> system-infrastructure-style code?
>

If they do, they actually be come "users of memcg". This here is aimed
at non-users of memcg, which given all the whining about it, it seems to
be plenty.

Also, I noticed, for instance, that libvirt is now creating memcg
hierarchies for lxc and qemu as placeholders, before you actually create
any vm or container.

There are two ways around that:
1) Have userspace in general to defer the creation of those directories
until they are actually going to use it, given the costs associated with
this.

2) defer our creation of memcg structures until the first task joins the
group, which then is the most clear signal possible that this is being
used.

> iirc, the idea of disabling memcg operations until someone enables a
> container had a couple of problems:
>
I'll need other people to jump in here and make it specific, but in general:

> a) certain boot-time initialisation issn't performed and
>

I am calling page_cgroup_init() at 1st memcg creation time.
The problem still exist that we will have tasks that are in LRUs but
with page_cgroup not filled. I handled this is this series by just not
testing this case, and assuming that empty page_cgroup == root_cgroup.

This can make bugs harder to find should they arise, but I'll argue that
it is worth it, given the gains.

> b) when memcg starts running for real, it expects that certain stats
>   gathering has been running since boot. If this is not the case,
>   those stats are wrong and stuff breaks.
>

I need specifics as well, but in general, my strategy lays in the
observation that all those statistics are already gathered in a global
level. We just report the global statistics when someone asks for it for
the root cgroup.

> It would be helpful if you could summarise these and similar issues
> and describe how they were addressed.
>

See above.
I would appreciate if anyone with a more specific and directed concern
would raise it.

>> >
>> > ...
>> >
>> > struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
>> > {
>> > + if (mem_cgroup_disabled())
>> > + return root_mem_cgroup;
> There would be some benefit in inlining the above instructions into the
> caller.
>
>> >   return mem_cgroup_from_css(
>> >   cgroup_subsys_state(cont, mem_cgroup_subsys_id));
>> > }

> In fact the entire mem_cgroup_from_cont() could be inlined.
>

Indeed.

>> > struct mem_cgroup *mem_cgroup_from_task(struct task_struct *p)
>> > {
>> > +
>> > + if (mem_cgroup_disabled())
>> > + return root_mem_cgroup;
> Ditto.
>

Indeetto.

>> > /*
>> > * mm_update_next_owner() may clear mm->owner to NULL
>> > * if it races with swapoff, page migration, etc.
>> >
>> > ...
>> >

Subject: Re: [RFC 2/4] memcg: make it suck faster
Posted by Daniel P. Berrange on Wed, 26 Sep 2012 09:03:43 GMT
View Forum Message <> Reply to Message

On Wed, Sep 26, 2012 at 12:53:21PM +0400, Glauber Costa wrote:
> On 09/26/2012 01:02 AM, Andrew Morton wrote:
> >> nomemcg : memcg compile disabled.
> >> > base : memcg enabled, patch not applied.
> >> > bypassed : memcg enabled, with patch applied.
> >> >
> >> > base bypassed
> >> > User 109.12 105.64
> >> > System 1646.84 1597.98
> >> > Elapsed 229.56 215.76
> >> >
> >> > nomemcg bypassed
> >> > User 104.35 105.64
> >> > System 1578.19 1597.98
> >> > Elapsed 212.33 215.76
> >> >
> >> > So as one can see, the difference between base and nomemcg in terms
> >> > of both system time and elapsed time is quite drastic, and consistent
> >> > with the figures shown by Mel Gorman in the Kernel summit. This is a
> >> > ~ 7 % drop in performance, just by having memcg enabled. memcg functions
> >> > appear heavily in the profiles, even if all tasks lives in the root

> >> > memcg.
> >> >
> >> > With bypassed kernel, we drop this down to 1.5 %, which starts to fall
> >> > in the acceptable range. More investigation is needed to see if we can
> >> > claim that last percent back, but I believe at last part of it should
> >> > be.
> > Well that's encouraging. I wonder how many users will actually benefit
> > from this - did I hear that major distros are now using memcg in some
> > system-infrastructure-style code?
> >
>
> If they do, they actually be come "users of memcg". This here is aimed
> at non-users of memcg, which given all the whining about it, it seems to
> be plenty.
>
> Also, I noticed, for instance, that libvirt is now creating memcg
> hierarchies for lxc and qemu as placeholders, before you actually create
> any vm or container.

This is mostly just lazyness on our part. There's no technical reason
why we can't delay creating our intermediate cgroups until we actually
have a VM ready to start, it was just simpler to create them when we
started the main daemon.

Daniel
--
|: http://berrange.com -o- http://www.flickr.com/photos/dberrange/ :|
|: http://libvirt.org -o- http://virt-manager.org :|
|: http://autobuild.org -o- http://search.cpan.org/~danberr/ :|
|: http://entangle-photo.org -o- http://live.gnome.org/gtk-vnc :|

Subject: Re: [RFC 1/4] memcg: provide root figures from system totals
Posted by Michal Hocko on Mon, 01 Oct 2012 17:00:46 GMT
View Forum Message <> Reply to Message

On Tue 25-09-12 12:52:50, Glauber Costa wrote:
> For the root memcg, there is no need to rely on the res_counters.

This is true only if there are no children groups but once there is at
least one we have to move global statistics into root res_counter and
start using it since then. This is a tricky part because it has to be
done atomically so that we do not miss anything.

> The sum of all mem cgroups plus the tasks in root itself, is necessarily
> the amount of memory used for the whole system. Since those figures are
> already kept somewhere anyway, we can just return them here, without too

> much hassle.
>
> The limit can't be set for the root cgroup, so it is left at 0.

You meant RESOURCE_MAX, didn't you?

> Same is
> true for failcnt, because its actual meaning is how many times we failed
> allocations due to the limit being hit. We will fail allocations in the
> root cgroup, but the limit will never the reason.
>
> TODO includes figuring out what to do with the soft limit and max_usage.
> Comments and suggestions appreciated.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Mel Gorman <mgorman@suse.de>
> CC: Andrew Morton <akpm@linux-foundation.org>
> ---
> mm/memcontrol.c | 45 ++
> 1 file changed, 45 insertions(+)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 82c5b8f..bac398b 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -4506,6 +4506,45 @@ static inline u64 mem_cgroup_usage(struct mem_cgroup *memcg,
bool swap)
>   return val << PAGE_SHIFT;
> }
>
> +static u64 mem_cgroup_read_root(struct mem_cgroup *memcg, enum res_type type, int
name)
> +{
> + struct sysinfo i;
> +   unsigned long pages[NR_LRU_LISTS];
> +   int lru;
> +
> + si_meminfo(&i);
> + si_swapinfo(&i);
> +
> + if (name == RES_LIMIT)
> + return RESOURCE_MAX;
> + if (name == RES_SOFT_LIMIT)
> + return 0;
> + if (name == RES_FAILCNT)

> + return 0;
> + if (name == RES_MAX_USAGE)
> + return 0;
> +
> + if (WARN_ON_ONCE(name != RES_USAGE))
> + return 0;
> +
> + for (lru = LRU_BASE; lru < NR_LRU_LISTS; lru++)
> + pages[lru] = global_page_state(NR_LRU_BASE + lru);
> +
> + switch (type) {
> + case _MEM:
> + return pages[LRU_ACTIVE_ANON]   + pages[LRU_ACTIVE_FILE] +
> +   pages[LRU_INACTIVE_ANON] + pages[LRU_INACTIVE_FILE];
> + case _MEMSWAP:
> + return pages[LRU_ACTIVE_ANON]   + pages[LRU_ACTIVE_FILE] +
> +   pages[LRU_INACTIVE_ANON] + pages[LRU_INACTIVE_FILE] +
> +   i.totalswap - i.freeswap;
> + case _KMEM:
> + return 0;
> + default:
> + BUG();
> + };
> +}
> +
> static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
> struct file *file, char __user *buf,
> size_t nbytes, loff_t *ppos)
> @@ -4522,6 +4561,11 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   if (!do_swap_account && type == _MEMSWAP)
>   return -EOPNOTSUPP;
>
> + if (mem_cgroup_is_root(memcg)) {
> + val = mem_cgroup_read_root(memcg, type, name);
> + goto root_bypass;
> + }
> +
>   switch (type) {
>   case _MEM:
>   if (name == RES_USAGE)
> @@ -4542,6 +4586,7 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   BUG();
>   }
>
> +root_bypass:
>   len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);

> return simple_read_from_buffer(buf, nbytes, ppos, str, len);
> }
> --
> 1.7.11.4
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [RFC 1/4] memcg: provide root figures from system totals
Posted by Glauber Costa on Tue, 02 Oct 2012 09:15:43 GMT
View Forum Message <> Reply to Message

On 10/01/2012 09:00 PM, Michal Hocko wrote:
> On Tue 25-09-12 12:52:50, Glauber Costa wrote:
>> > For the root memcg, there is no need to rely on the res_counters.
> This is true only if there are no children groups but once there is at
> least one we have to move global statistics into root res_counter and
> start using it since then. This is a tricky part because it has to be
> done atomically so that we do not miss anything.
>
Why can't we shortcut it all the time?

It makes a lot of sense to use the root cgroup as the sum of everything,
IOW, global counters. Otherwise you are left in a situation where you
had global statistics, and all of a sudden, when a group is created, you
start having just a subset of that, excluding the tasks in root.

If we can always assume root will have the sum of *all* tasks, including
the ones in root, we should never need to rely on root res_counters.

Subject: Re: [RFC 1/4] memcg: provide root figures from system totals
Posted by Michal Hocko on Tue, 02 Oct 2012 09:34:33 GMT
View Forum Message <> Reply to Message

On Tue 02-10-12 13:15:43, Glauber Costa wrote:
> On 10/01/2012 09:00 PM, Michal Hocko wrote:
> > On Tue 25-09-12 12:52:50, Glauber Costa wrote:
> >> > For the root memcg, there is no need to rely on the res_counters.
> > This is true only if there are no children groups but once there is at

> > least one we have to move global statistics into root res_counter and
> > start using it since then. This is a tricky part because it has to be
> > done atomically so that we do not miss anything.
> >
> Why can't we shortcut it all the time?

Because it has its own tasks and we are still not at use_hierarchy := 1

> It makes a lot of sense to use the root cgroup as the sum of everything,
> IOW, global counters. Otherwise you are left in a situation where you
> had global statistics, and all of a sudden, when a group is created, you
> start having just a subset of that, excluding the tasks in root.

Yes because if there are no other tasks then, well, global == root. Once
you have more groups (with tasks of course) then it depends on our
favorite use_hierarchy buddy.

> If we can always assume root will have the sum of *all* tasks, including
> the ones in root, we should never need to rely on root res_counters.

but we are not there yet.

--
Michal Hocko
SUSE Labs