Subject: Re: [PATCH 1/4] Add notification about some major slab events
Posted by Christoph Lameter on Wed, 19 Sep 2007 17:45:32 GMT
View Forum Message <> Reply to Message

On Wed, 19 Sep 2007, Pavel Emelyanov wrote:

> so the fast path is still fast, and we have two ways:
> 1. we keep the checks on the fastpath and have 0 overhead for
>   unaccounted caches and some overhead for accounted;

This stuff accumulates. I have a bad experience from SLAB. We are counting
cycle counts and cachelines touched in the fastpath and this is going to
add to them.

> 2. we move the checks into the slow one and have 0 overhead for
>   unaccounted caches and huge overhead for accounted.

Huge? Its not that huge.

> I admit that I messed something, so shall I measure some
> other activity or use another HW?

You could use this module to test the cycles in the fastpath:

```
/* test-slub.c
 */


#include <linux/jiffies.h>
#include <linux/compiler.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/calc64.h>
#include <asm/timex.h>
#include <asm/system.h>

#define TEST_COUNT 10000

#define PARALLEL

#ifdef PARALLEL
#include <linux/completion.h>
#include <linux/sched.h>
#include <linux/workqueue.h>
#include <linux/kthread.h>

struct test_struct {
```

```
 struct task_struct *task;
 int cpu;
 int size;
 int count;
 void **v;
 void (*test_p1)(struct test_struct *);
 void (*test_p2)(struct test_struct *);
 unsigned long start;
 unsigned long stop1;
 unsigned long stop2;
} test[NR_CPUS];

/*
 * Allocate TEST_COUNT objects and later free them all again
 */
static void kmalloc_alloc_then_free_test_p1(struct test_struct *t)
{
 int i;

 for (i = 0; i < t->count; i++)
  t->v[i] = kmalloc(t->size, GFP_KERNEL);
}

static void kmalloc_alloc_then_free_test_p2(struct test_struct *t)
{
 int i;

 for (i = 0; i < t->count; i++)
  kfree(t->v[i]);
}

/*
 * Allocate TEST_COUNT objects. Free them immediately.
 */
static void kmalloc_alloc_free_test_p1(struct test_struct *t)
{
 int i;

 for (i = 0; i < TEST_COUNT; i++)
  kfree(kmalloc(t->size, GFP_KERNEL));
}

static atomic_t tests_running;
static DECLARE_COMPLETION(completion);
static int started;

static int test_func(void *private)
{
```

```
 struct test_struct *t = private;
 cpumask_t newmask = CPU_MASK_NONE;

        cpu_set(t->cpu, newmask);
        set_cpus_allowed(current, newmask);
 t->v = kmalloc(t->count * sizeof(void *), GFP_KERNEL);

 atomic_inc(&tests_running);
 wait_for_completion(&completion);
 t->start = get_cycles();
 t->test_p1(t);
 t->stop1 = get_cycles();
 if (t->test_p2)
  t->test_p2(t);
 t->stop2 = get_cycles();
 kfree(t->v);
 atomic_dec(&tests_running);
 set_current_state(TASK_UNINTERRUPTIBLE);
 schedule();
 return 0;
}

static void do_concurrent_test(void (*p1)(struct test_struct *),
  void (*p2)(struct test_struct *),
  int size, const char *name)
{
 int cpu;
 unsigned long time1 = 0;
 unsigned long time2 = 0;
 unsigned long sum1 = 0;
 unsigned long sum2 = 0;

 atomic_set(&tests_running, 0);
 started = 0;
 init_completion(&completion);

 for_each_online_cpu(cpu) {
  struct test_struct *t = &test[cpu];

  t->cpu = cpu;
  t->count = TEST_COUNT;
  t->test_p1 = p1;
  t->test_p2 = p2;
  t->size = size;
  t->task = kthread_run(test_func, t, "test%d", cpu);
  if (IS_ERR(t->task)) {
   printk("Failed to start test func\n");
   return;
```

```
 }
}

/* Wait till all processes are running */
while (atomic_read(&tests_running) < num_online_cpus()) {
 set_current_state(TASK_UNINTERRUPTIBLE);
 schedule_timeout(10);
}
complete_all(&completion);
while (atomic_read(&tests_running)) {
 set_current_state(TASK_UNINTERRUPTIBLE);
 schedule_timeout(10);
}

for_each_online_cpu(cpu)
 kthread_stop(test[cpu].task);

printk(KERN_ALERT "%s(%d):", name, size);
for_each_online_cpu(cpu) {
 struct test_struct *t = &test[cpu];

 time1 = t->stop1 - t->start;
 time2 = t->stop2 - t->stop1;
 sum1 += time1;
 sum2 += time2;
 printk(" %d=%lu", cpu, time1 / TEST_COUNT);
 if (p2)
  printk("/%lu", time2 / TEST_COUNT);
}
printk(" Average=%lu", sum1 / num_online_cpus() / TEST_COUNT);
if (p2)
 printk("/%lu", sum2 / num_online_cpus() / TEST_COUNT);
printk("\n");
schedule_timeout(200);
}
#endif

static int slub_test_init(void)
{
 void **v = kmalloc(TEST_COUNT * sizeof(void *), GFP_KERNEL);
 unsigned int i;
 cycles_t time1, time2, time;
 long rem;
 int size;

 printk(KERN_ALERT "test init\n");

 printk(KERN_ALERT "Single thread testing\n");
```

```
 printk(KERN_ALERT "====================\n");
 printk(KERN_ALERT "1. Kmalloc: Repeatedly allocate then free test\n");
 for (size = 8; size <= PAGE_SIZE << 2; size <<= 1) {
  time1 = get_cycles();
  for (i = 0; i < TEST_COUNT; i++) {
   v[i] = kmalloc(size, GFP_KERNEL);
  }
  time2 = get_cycles();
  time = time2 - time1;

  printk(KERN_ALERT "%i times kmalloc(%d) ", i, size);
  time = div_long_long_rem(time, TEST_COUNT, &rem);
  printk("-> %llu cycles ", time);

  time1 = get_cycles();
  for (i = 0; i < TEST_COUNT; i++) {
   kfree(v[i]);
  }
  time2 = get_cycles();
  time = time2 - time1;

  printk("kfree ");
  time = div_long_long_rem(time, TEST_COUNT, &rem);
  printk("-> %llu cycles\n", time);
 }

 printk(KERN_ALERT "2. Kmalloc: alloc/free test\n");
 for (size = 8; size <= PAGE_SIZE << 2; size <<= 1) {
  time1 = get_cycles();
  for (i = 0; i < TEST_COUNT; i++) {
   kfree(kmalloc(size, GFP_KERNEL));
  }
  time2 = get_cycles();
  time = time2 - time1;

  printk(KERN_ALERT "%i times kmalloc(%d)/kfree ", i, size);
  time = div_long_long_rem(time, TEST_COUNT, &rem);
  printk("-> %llu cycles\n", time);
 }
 kfree(v);
#ifdef PARALLEL
 printk(KERN_INFO "Concurrent allocs\n");
 printk(KERN_INFO "================\n");
 for (i = 3; i <= PAGE_SHIFT; i++) {
  do_concurrent_test(kmalloc_alloc_then_free_test_p1,
   kmalloc_alloc_then_free_test_p2,
   1 << i, "Kmalloc N*alloc N*free");
 }
```

```
 for (i = 3; i <= PAGE_SHIFT; i++) {
  do_concurrent_test(kmalloc_alloc_free_test_p1, NULL,
   1 << i, "Kmalloc N*(alloc free)");
 }
#endif

 return -EAGAIN; /* Fail will directly unload the module */
}

static void slub_test_exit(void)
{
 printk(KERN_ALERT "test exit\n");
}

module_init(slub_test_init)
module_exit(slub_test_exit)

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mathieu Desnoyers");
MODULE_DESCRIPTION("SLUB test");
```