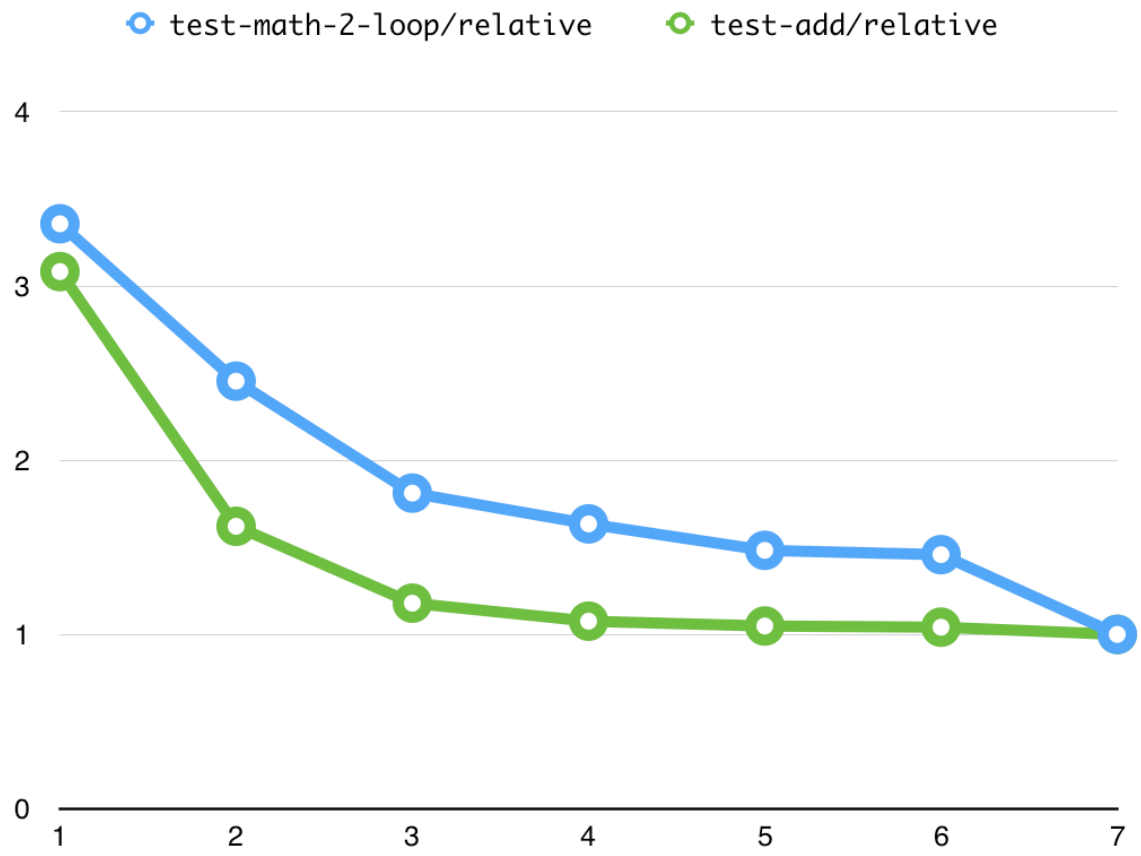
框架版本：　　new-sim-algorithm-with-simple-scheduler - Fri Jul 31 16:32:17 2015 +0800

模拟器版本：　　cache_core_mesh_1024cores

测试程序：　　test-math-2-loop



如上图所示，在框架线程数从 1 增加至 7 的过程中，系统运行时间从 1635 秒逐渐降至 487 秒，加速比约为 3.36。

框架版本: new-sim-algorithm-with-simple-scheduler - Fri Jul 31 16:32:17 2015 +0800

模拟器版本: cache_core_mesh_1024cores

测试程序: server208:SimIT/mmu_for_simu/test-math.c

采集事件: cycles

```
Samples: 21M of event 'cycles', Event count (approx.): 9842930815815
 58.18%  runtime  runtime              [.] worker
  7.92%  runtime  [kernel.kallsyms]   [k] change_protection_range
  3.22%  runtime  runtime              [.] main
  2.77%  runtime  libcore.so           [.] ss_decode
  2.75%  runtime  libc-2.17.so         [.] _int_malloc
  2.61%  runtime  runtime              [.] move_new_msgs.constprop.12
  1.86%  runtime  libc-2.17.so         [.] _int_free
  1.71%  runtime  libmemory.so         [.] mem_access
  1.66%  runtime  libcore.so           [.] ss_writeback
  1.42%  runtime  libcache.so          [.] simict_port_msg_proc
  1.08%  runtime  [kernel.kallsyms]   [k] vm_normal_page
  0.99%  runtime  libcore.so           [.] ss_issue
```

从 perf 的结果看到：main.c 中的 worker()函数占总体运行时间的 58.18%，

```
       |
       |     static inline u64 task_q_get(task_q_t *q, task_t *buf[], u64 n)
       |     {
       |             u64 i;
       |             u64 head = q->head;
  0.00 | 18:   mov    (%r15),%rax
  0.05 |       nop
       |             u64 tail = smp_load_acquire(&q->tail);
  0.01 | 20:   mov    0x8(%r15),%rdx
       |             while (head == tail)
 94.57 |       cmp    %rdx,%rax
  0.79 |     ↑ je     20
       |                     tail = smp_load_acquire(&q->tail);
       |             for (i = 0; i < n && head != tail; i++) {
       |                     buf[i] = ACCESS_ONCE(q->buf[head]);
       |       lea    0x2(%rax),%rcx
       |                     head = (head + 1) % BUF_SIZE;
  0.03 |       add    $0x1,%rax
  0.00 |       movzbl %al,%eax
```

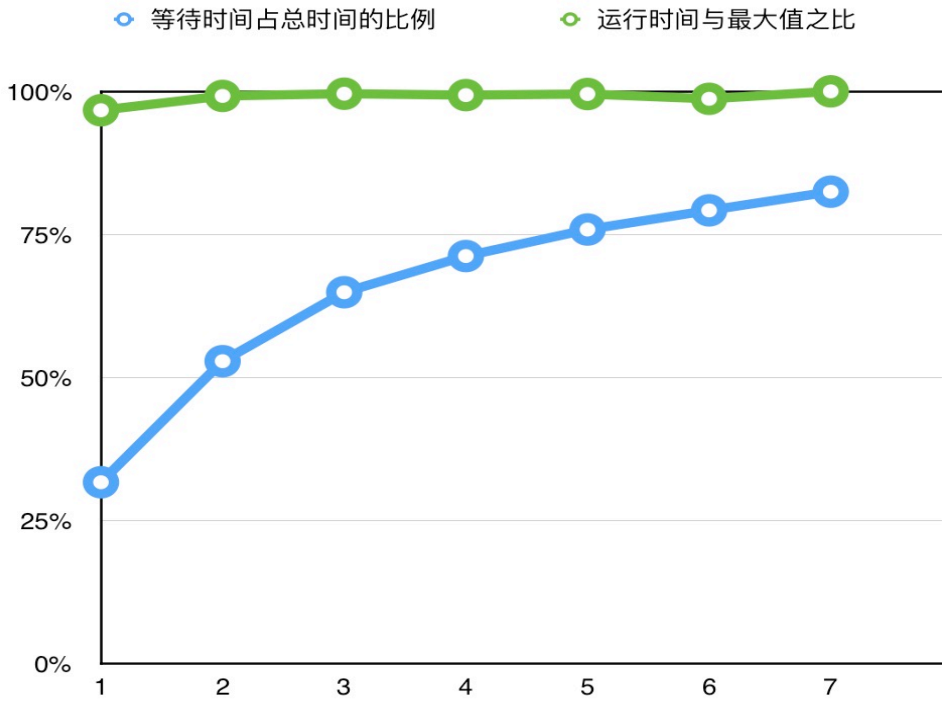
而该函数中占用时间 94.57%的函数是 msg_q_get()中

while (head == tail)

tail = mp_load_acquire(&q->tail);

从中可以的分析出 msg_q_get()占总体运行时间 55.02%的比例 。

而在多线程方式执行的模拟器中，这个比例更加夸张。



随着线程数从 1 增加至 7，整个系统的运行时间几乎不变，而其中等待时间占总运行时间的比例却呈近似线性的增加。

分析 task_q_get()，这个 while 循环执行的操作是从存储消息的循环队列中获取新的消息。在这一步上花费如此高比例的时间，说明模拟器没有向框架持续的提供新的消息，框架只得反复地到消息队列中询问是否提供了新消息。

在这样的情况下，现有的多个线程就已经不能在很短的时间里得到需要执行的消息，即使再增加线程数，也不能使整个系统执行的更快（没有那么多活可干）

从消息数量的角度看两种运行方式的可扩展性差异：

多线程方式中实际只是跑了一个应用程序，其事件消息非常有限，而多进程方式实际上是将同一个应用程序同时跑多次，产生的消息数量是原来的多倍。加之由于多个程序同时跑，一定程度上掩盖了消息的延迟，使得消息供给的速度也更快。

多线程方式中，如果从总的运行时间中减去因为等待消息而浪费的时间，则可以发现其实际执行的时间是具有很好的可扩展性的（从 1 线程到 7 线程有 4 倍加速）。

附：worker()函数及 task_q_get()函数的实现

```c
static void* worker(void *data)
{
        worker_t *w = data;
        runtime_info_t *info = &g_runtime_info;

        task_t *doing[TASK_Q_SIZE];
        while (1) {
                u64 doing_len = task_q_get(&w->todo_q, doing, TASK_Q_SIZE);
                for (u64 i = 0; i < doing_len; i++) {
                        do_callback_on_task(info, doing[i]);
                        u64 nr = task_q_put(&w->done_q, &doing[i], 1);
                        assert(nr == 1);
                }
        }
        return NULL;
}
```

```c
static inline u64 task_q_get(task_q_t *q, task_t *buf[], u64 n)
{
        u64 i;
        u64 head = q->head;
        u64 tail = smp_load_acquire(&q->tail);
        while (head == tail)
                tail = smp_load_acquire(&q->tail);
        for (i = 0; i < n && head != tail; i++) {
                buf[i] = ACCESS_ONCE(q->buf[head]);
                head = (head + 1) % BUF_SIZE;
        }
        smp_store_release(&q->head, head);
        return i;
}
```

主线程调度worker线程，worker就是加速线程。
主线程和每个workder线程都有一对消息队列，用于任务分发和结果回收。