Measuring memory throughputs with multiple virtual machines

Final Project Report for CS7600: Intensive Computer Systems, Spring 2012

Instructor: Prof. Peter Desnoyers

Authors: Rohan Garg, Qian Zhang, Jingzhi Yu

Contents

1. Introduction

Virtualization of hardware is being widely used these days to increase the profit margins, agile deployment, inherent security attributes and reducing the downtime in organizations and IT services industry. In this project we aim to run a series of experiments to understand the behavior and some of the performance parameters of an operating system (OS) running in a virtualized environment.

We are interested in measuring the impact of running multiple guest OS's on top of a para-virtualized hypervisor to memory throughputs. This is a common setup being employed at various organizations which provide cloud services. To get a better insight into this we run three experiments: 1) measure the typical switching times with multiple guest OS's running on a hypervisor, 2) measure the *throughput loss* which might come as an artifact of the switching, and 3) measure the *throughput* as a function of *switching time*. The benchmarks we use are 1) Dhrystone, 2) STREAM, 3) Stress.

We chose to run our experiments using Lguest, which is a para-virtualized virtual machine monitor (VMM) for Linux, mainly because of its relatively small code base (~6000 LOC), which makes it conducive to small modifications and tweaks which might have been required.

1.1 Lguest Overview

Lguest is a minimalistic 32-bit x86 para-virtualized (as opposed to full-virtualization) hypervisor for the Linux kernel. It became a part of the Linux kernel since version 2.6.23. Lguest allows running multiple copies of Linux kernel as separate processes on the host OS. The Linux kernel running as a guest needs to be *aware* of the virtualized environment, in order to run. This requires compiling the kernel (for both -- the guest as well as the host) with Lguest support. A brief on the internals of Lguest is described below.

The core of Lguest is a loadable module (*lg.ko*) in the host Linux kernel, which creates a character-device driver entry -- */dev/lguest* -- in the host OS space. Lguest has a user-space component, called *Launcher*, which 1) initializes the memory for the guest OS, 2) maps the kernel image to memory, 3) sets up the virtual devices for the guest and 4) does a *read()* on the */dev/lguest* to inform the kernel module about the guest and run it.

As mentioned earlier, the guest kernel is aware of the virtualized environment in the sense that (a) the guest kernel is runs in ring 1 (and not ring 0, which it would usually expect to) and as a consequence of which, (b) the privileged operations/instructions are replaced with "hypercalls" which land into the hypervisor. The guest requests host operations through a "hypercall", which similar to a system call used by normal processes. Figure 1 shows a high level architecture of the Lguest para-virtualization approach.

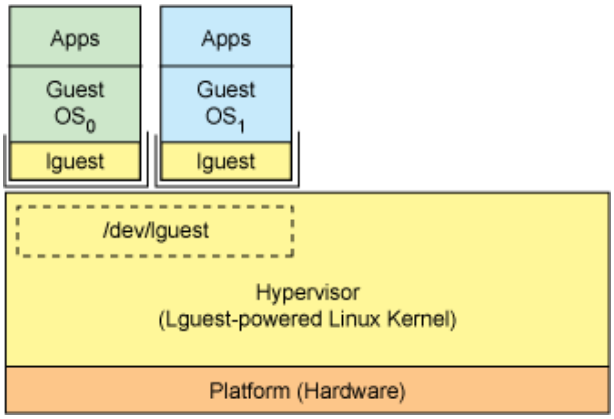

Figure 1 Lguest approach to para-virtualization (source: http://www.ibm.com/developerworks/linux/library/l-hypervisor/index.html?ca=dgr-lnxw06Lnx-Hypervisor&S_TACT=105AGX59&S_CMP=grlnxw06)

The other mechanism which Lguest uses for communication between the host and the guest is a shared structure, *struct lguest_data*. This provides a faster method to communicate things like the interrupt vector. This struct is mapped into the memory of process containing the guest OS and the host kernel.

The booting procedure for Lguest is presented in Figure 2.

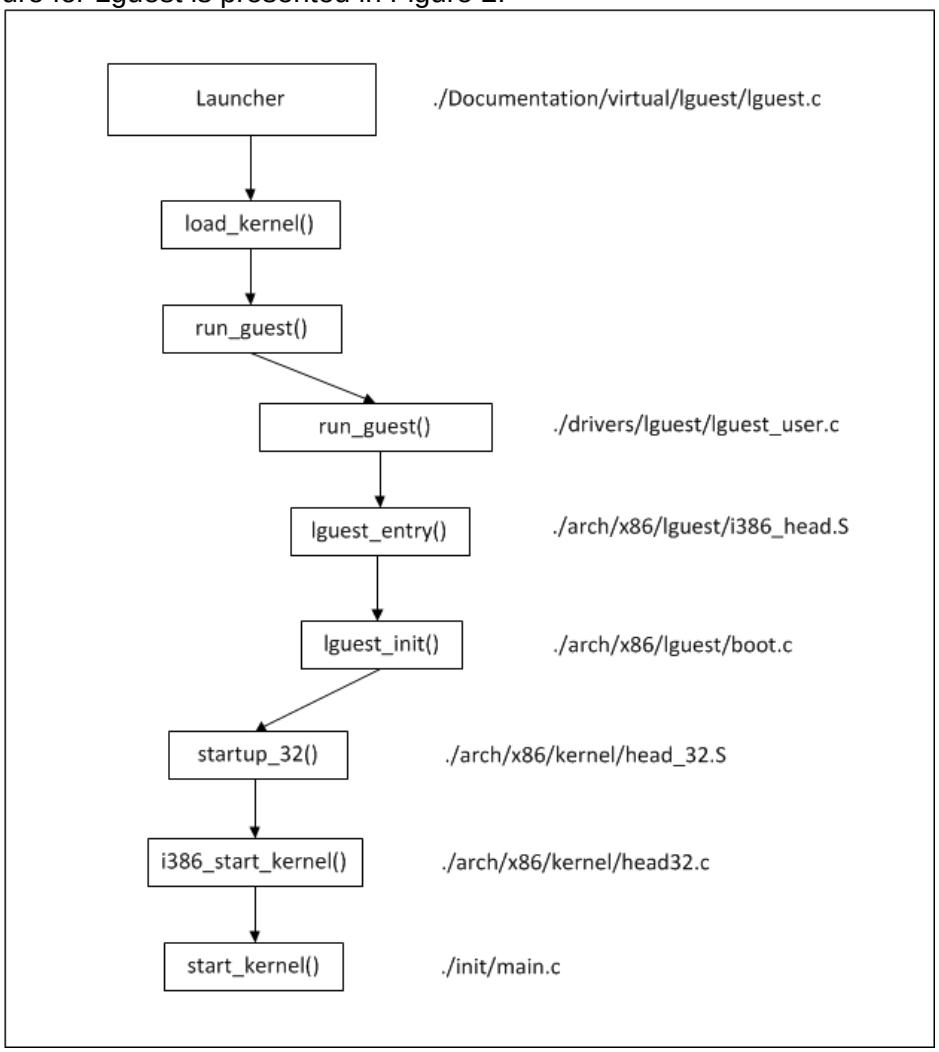

Figure 2 Booting Sequence of Guest with Lguest

1.2 Benchmark Tools

In this project, we mainly used Stream benchmark to measure memory throughputs for guest OSs sitting in lguest hypervisor. Stream benchmark [6] aims to provide a sustainable tool to measure memory bandwidth. It uses four application functions to measure the memory throughputs by summing the amount of data that an application code explicitly reads plus the amount of data that the application code explicitly writes. Using the previous 1 million byte copy example, the Stream bandwidth would be counted as 1 million bytes read plus 1 million bytes written in one second, for a total of 2 million bytes per second. In addition to Stream benchmark, we also employed another two benchmarks: Stress and Dhrystone. Stress benchmark [3] is not a 'standard' benchmark tool; however, it started out as a very simple way to generate work on a computer and provides a series of testing on memory bandwidth, cpu performance, etc. But in our experiment, we just modified the memory bandwidth measurement to fit our requirements. Dhrystone [2] gathers meta-data from a broad range of applications and measuring the processor performance of some common set of programs.

1.3 Related Works

The addition of a hypervisor to the software stack can have important performance implications for a computer system. XenoProf [7] is one tool devised to address the challenges of profiling the whole system taking into account the existence of a hypervisor and its related side-effects on performance and an extra layer of indirection. It leverages the system-level statistical profiling capabilities of OProfile [1] and adds support for a more “informed” profiling. Du et. al. in [4] provide another tool utilizing hardware performance counters to implement both guest-wide and system-wide profiling on KVM hypervisor.
To the the best of our knowledge, LgDb [5] is only tool which uses Lguest to provide a framework for kernel profiling, code coverage using Lguest. It adds support for hypercalls in Lguest which, once, instrumented in the code can be used for profiling and code-coverage analysis.

3. Design of Experiments

For all the experiments, we use the number of CPU cycles between the points of interest as a measure of the time difference. Getting the value of CPU cycles is done using the x86 *rdtsc* instruction. The conversion from cycle count to actual time is just a matter of scaling; the nature of the graphs will remain unaffected by this.

Experiment 1: Measuring the switching time between multiple running guests
The *switcher_32* module, which is just a piece of assembly code, handles the switching into the guest kernel part and switching the CR3 to point to the new page tables. We add the code in Lguest to keep track of the state and measure the cycle count just before switching. Then we launch two guests simultaneously and let them run without any interruptions for two minutes. We keep track of the minimum cycle counter difference encountered till now in the Lguest kernel module. Storing this value immediately on the disk is not done to avoid disruptions to the experiment. After that, the guests are killed and the value from the kernel module is read by a separate user space process.

Experiment 2: Measuring the throughput loss after switching
We add the capability to control the switching between guests at the hypervisor level. The switching can be done in two scenarios:

- *Scenario 1*: Switch into a guest once from another running guest and then keep running
- *Scenario 2*: Alternate running between two guests after each given time interval

These switching modes and timings are controlled by a separate user space program at run-time. For this experiment, we add the executable of the benchmarks on the guest root-fs and they are made to run at the bootup by adding entries in the init.d scripts. Two guest OS’s are started up simultaneously and after a fixed time into the run, the separate user-space process writes to the /dev/lguest kernel module to change the mode and switching time.

Experiment 3: Measuring the throughput as a function of switching times
The design of this experiment is almost similar to that of experiment-2, except for the fact that the user-space program chooses to write a different mode value to /dev/lguest.

3.1 Test Setup

The attributes of the various actors in our setup for running the experiments are described below.

- Linux Distribution -- Ubuntu 11.10
- Linux Kernel -- 3.0.22 - i686, i386
    - We run a custom compiled kernel with Lguest support
    - OLPC support is compiled out
    - The *.config* file used to compile the kernel can be found in Appendix A
- CPU -- Intel Core i7-2620M @2.7Ghz
    - Cache size -- 4MB

    - It is a quad-core CPU
    - We disable three cores while running our experiments
- Host Main Memory -- 4GB
- Virtual Machine Parameters
    - Maximum number of simultaneous running VMs -- 2
    - Memory -- 512MB
    - Mounted rootfs -- 1GB, ext2

3.2 Modifications in Lguest

For experiment-1, we needed to measure the switching times between different guests. We added code in the switcher module in Lguest to record the last switched-in guest and the value of the time stamp cycle-counter. If the new guest about to be switched-in is different from the last one, we take a difference of the cycle-counter values and store it.

In the file_ops defined for /dev/lguest, the option to query for this value is added so that when a user space application which is different from the Launcher does a write() on /dev/lguest, the buffer filled-in with this value is returned.

Lguest, currently, does not maintain a track of the multiple guest OS's running. The guests run in separate sandbox completely unaware of each other. To be able to control the switching and measure the various switching related parameters we added the code in the Lguest kernel module which allows it to keep track of the multiple guests. These involved keeping global data structures in the Lguest kernel module to point to the number of running guests, keeping a reference count, pointers to the running tasks, the last task which ran and other variables related to the guests.

For supporting controlled switching, a timer is added to the Lguest kernel module, the parameters (described earlier) of which can be controlled by doing writes to the /dev/lguest. These parameters are:

- Timer expiry interval in milli-seconds
- The numeric ID of the guest which to run initially
- The mode of the timer -- this extra parameter can be utilized to used run both experiments #2 and #3, with the same infrastructure in place. It, basically, denotes if the timer should be reset for continuous alternate switching or not.

The timer once started, expires and does a call-back to a newly defined function where the guest to-be-run next is brought back to the running queue and guest currently running is put to TASK_INTERRUPTIBLE state. Another variable, associated with the guest, ensures that even if the task that has been put to sleep is woken up (interrupted) by the kernel scheduler, it is not "run" unless it is its turn.

The modifications made in Lguest's source code are described in Appendix B.

3.3 Modifications in Benchmarks

We tried to avoid doing any big modifications to standard benchmarking tools but certain changes were warranted as we were running as a special case. The changes mainly included the changes in time measurement and modifications for controlling the number of runs.

As mentioned before, we use the cycle count as a more accurate measurement of the time difference in the execution time of the code between the points of interest. The use of rdtsc by a benchmark also makes us less susceptible to the errors in the clock of the guest OS. A reason for these errors is the modifications done to the Lguest code which allowed us to control the switching between the multiple guest OS's. While sending the IRQ-0 to disrupt a running guest, the clock on the guest OS goes awry, since IRQ-0 is the one used for clock interrupts.

The other modifications we did were to delay the printing or saving of values in the run till as late as possible. The purpose of this change was to avoid any interruptions since we want to measure the memory throughput and this interrupt would only increase the cycle count and skew our data.

4. Results and Discussion

4.1 Measuring switching time

In this part, we first measure the switching time by counting cpu cycles. As we estimated in Homework 1, the counter cycle frequency for the experiment machine is 75 cpu_cycles/ns. For each run of experiment, we calculate the minimum differences of cpu cycle counts between starting and ending of switching Δrdtsc. We choose the minimum value because of the same reason we explained in Homework 1, that due to possible interruptions in the host there could be extra cpu cycles for such interruptions, and therefore the minimum value is the closest to the real one.

Table 1 Statistical results for switching time in two scenarios.

	average switching time (ns)	median switching time (ns)	standard deviation	95% confidence interval	minimum switching time (ns)	maximum switching time (ns)
two idle guests	486.722311	297.28	504.544715	81.4036292	162.44	3280.68
one busy guest one idle guest	178.168311	155.88	51.6561076	8.2235327	102.64	400.72

We measured the time for the hypervisor to switch between two guest OSs in two different scenarios. In the first scenario, we just launched two guest OSs and they did not perform any other task and after 2 minutes both two guests were killed by the host in 2 minutes. In the second scenario, one guest OSs ran 10 loops of calculating factorial for integers from 1 to 50 at the end of system initialization, and another guest did nothing but regular system initialization. In this scenario, two guests were also killed by the host We run the experiment for 150 runs, and Table xx reports the results. The table shows very interesting results that both standard deviation and confidence interval is quite large for the scenario of two idle guests. The large difference between average switching time and the median value also shows the switching time varies significantly in this case. But we observe that the minimum values of switching time in two cases are closer. One possible explanation might be in the first scenario, since two guests are idle, the host kernel may decrease the priority of these two processes, and introduce more interruptions from the perspective of guest OSs.

4.2 Memory throughputs for guest OSs

We first ran Stream benchmark on a single guest OS. Figure 3 shows the median values of memory throughputs as a function of time steps for four testing functions in Stream benchmark. The results are very intuitive: overall the curve is flat with some tiny peaks and valleys because of possible interruptions. Figure 4 shows the memory throughputs as a function of time steps in the scenario of two running guest OSs, where the switching between two guests is controlled by the unmodified hypervisor. For all of four functions, one can observe there are more significant fluctuations for both guests. The reason is obvious: the switching to another guest results in interruptions from the perspective of the guest and reduces the speed to access memories. Note the x-axis is the time step, not the continuous time. Since when one guest is switched out, the

benchmarking temporarily stops working, there are some overlaps of peaks or valleys in the curves for two guests which in fact only represent relative temporal information for each guest. In the following experiments, the temporal information is with the same meaning.

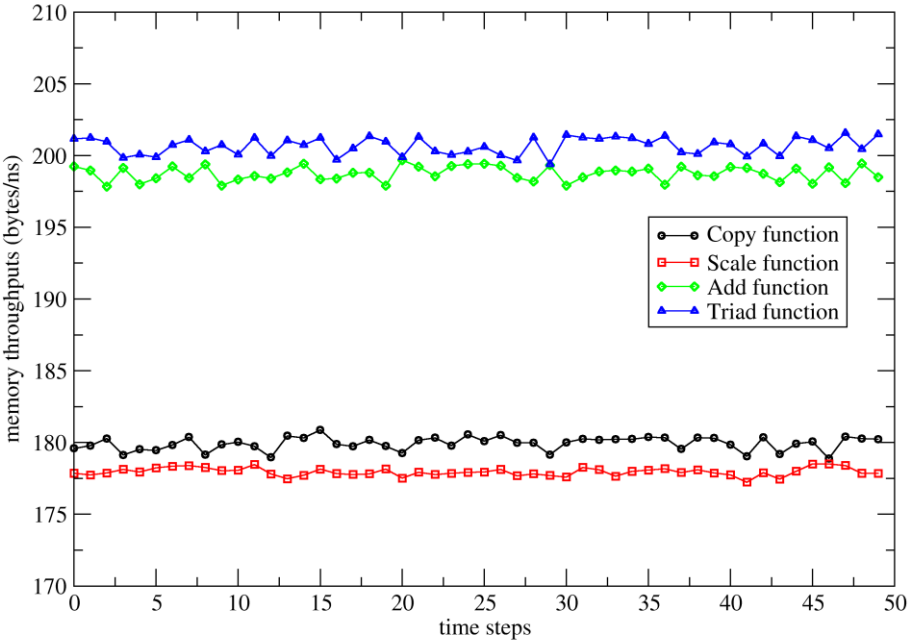

Figure 3 Memory throughputs from Stream benchmark for one guest running on lguest.

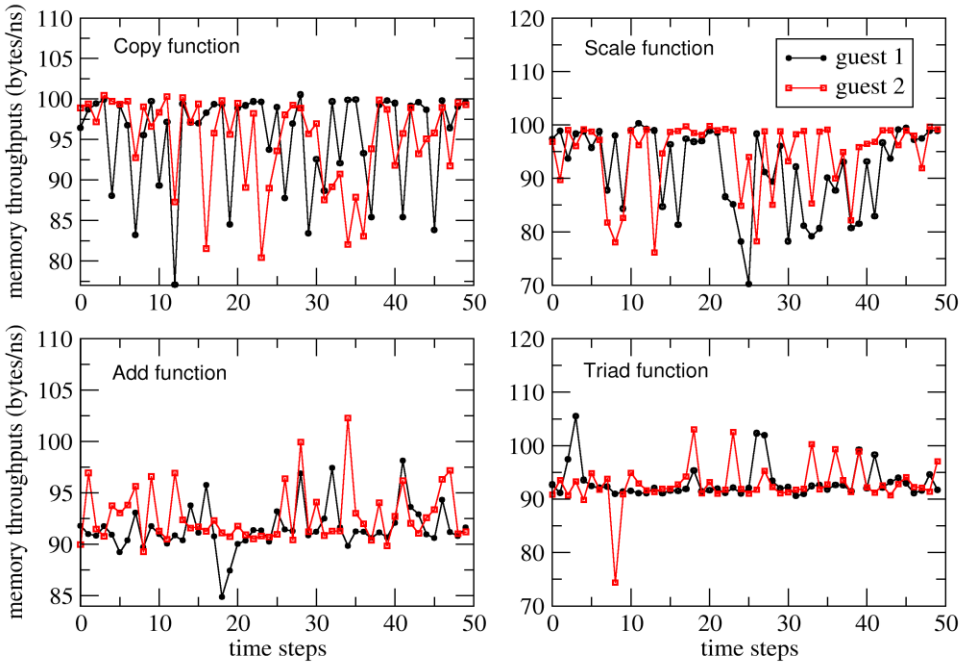

Figure 4 Memory throughputs from Stream benchmark for two guests running on lguest without switching control.

The above two experiments are baseline for the following experiments in which a user space program takes the role of controlling switching guest OSs. The first switching control experiment on memory throughputs is ‘one shot’ control, i.e., we started two guests simultaneously and halted the guest 1 first for 10 seconds and then resumed it and halted the guest 2 until both guests are killed by the host. We report the memory throughputs as a function of time step for guest 1 in Figure 5 The figure shows the average memory throughputs over 50 experimental runs for four testing function in Stream benchmark, and for all these function, the throughputs first decrease because the hypervisor switches to run guest 2 for a short period of time. After that, the curves jump up to peak values and become stable while guest 2 is halted in that period of time. The peak values are quite close to the values in Figure 3 in which only one guest ran.

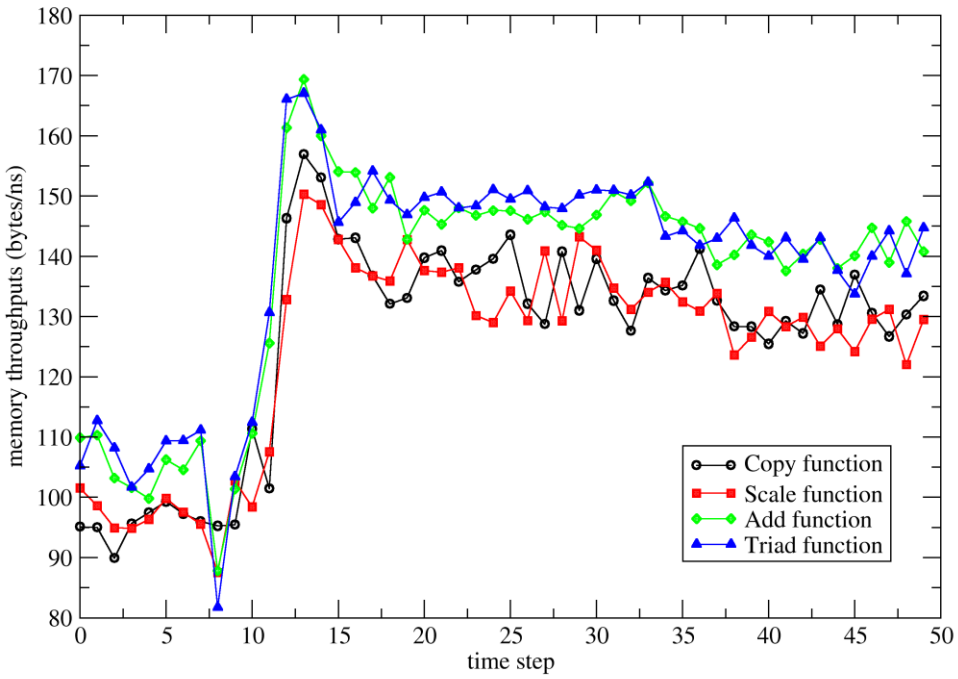

Figure 5 Memory throughputs from Stream benchmark for guest 1 when two guests run on lguest with one-shot control switching control. The guest 1 is switched off at first for short period of time, and switched back while the guest 2 is permanently switched out.

The second experiment we performed in the switching control case is to vary switching time between two guest and run Stream benchmark for different values of switching time. Figure 6 provides the average memory throughputs as a function of the controlled switching time for two guests for four testing functions in one experimental run. As the plots show, the average throughputs are increasing as the switching time grows. However, if we go into more details of the data, for instance Figure 7 depicts the boxplots of memory throughputs for guest 1, we can find in fact the value of memory throughputs varies significantly, especially the minimum value.

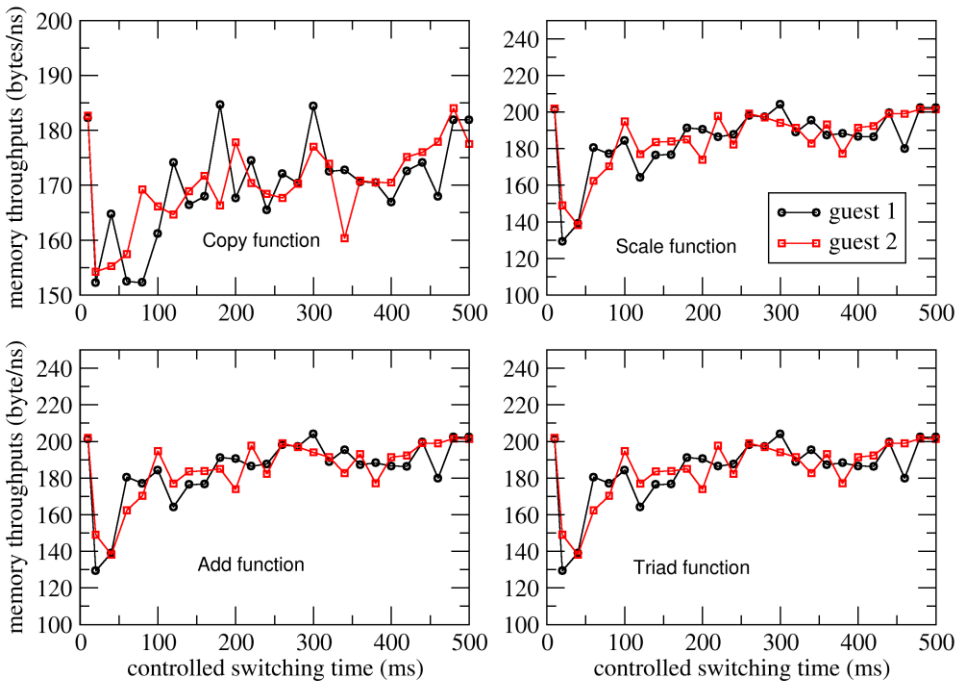

Figure 6 Memory throughputs from Stream benchmark for two guests running on lguest with different controlled switching time.

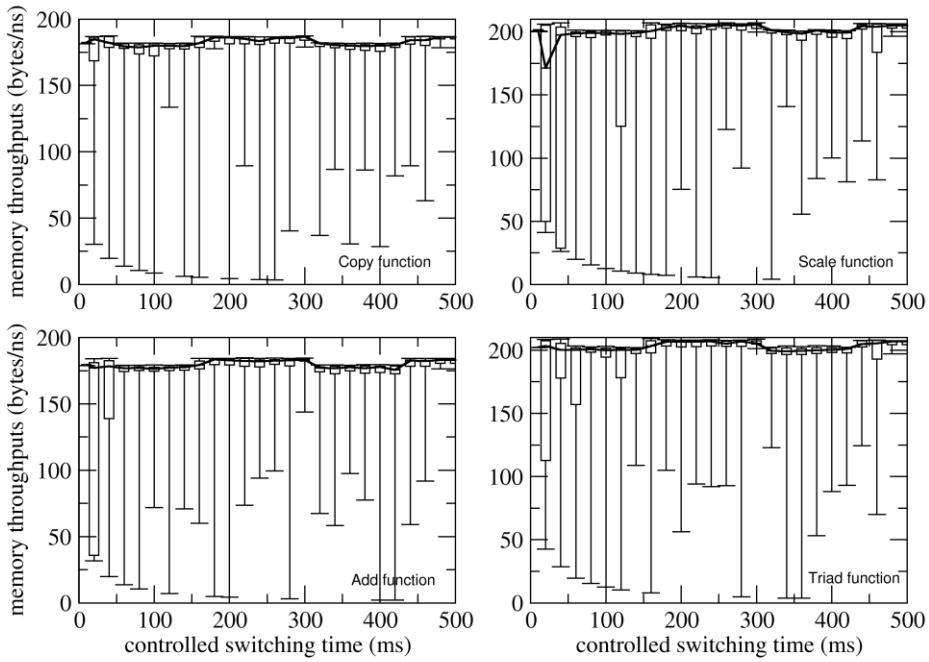

Figure 7 Boxplots for memory throughputs from Stream benchmark for guest 1 when two guests run on lguest with different controlled switching time (corresponding to Figure 6).

From the Figure 7, the general trend seems to convey that the throughput increases with decrease in the switching frequency (or increase in the time allocated to each guest), as a process gets more time to execute before being switched out, but upon a more minute analysis of the data, there are other interesting things worth noticing. The decrease in switching frequency makes the processes susceptible to much lower throughputs, because if a guest OS is switched out in the middle of a computation, it will be a much longer period before it is switched back in. This can actually result in more erratic throughputs and would be a bad idea to implement on any systems which is required to provide consistency guarantees (of throughput). Also, the maximum throughputs that are achieved may not be high enough to justify sacrificing a much more consistent behavior. It is also worth to note that since we have to output the results, such writing operations actually introduce many interruptions. All these results, though from many experimental runs, are just the best approximation we can achieve.

Besides Stream benchmark, we also employed Stress benchmark to measure memory throughputs for the first three experiments we performed above: one guest running, two guests running of whom switching is controlled by the hypervisor, and ‘one shot’ controlled switching for two guests. Not surprisingly, we got similar shapes of curves of memory throughputs as a function of time steps as in Figure 3, Figure 4 and Figure 5, shown in Figure 8. However, two interesting things should be noted in this set of results. First, the value of the memory throughputs measured in Stress benchmark is one order less than that from Stream benchmark experiments. Note that Stress benchmark does memory allocation and deallocation in every iteration of the measurements, while Stream benchmark uses a memory resident global array and does not rely on dynamic allocation. Since memory allocation operations are time consuming, so Stress benchmark actually uses more cpu cycles to finish each run of measurement. Second, the curves from Stress benchmark are less fluctuated than those from Stream benchmark, i.e., it seems there are fewer interruptions. However, so far we have no clue why it behaves like this.

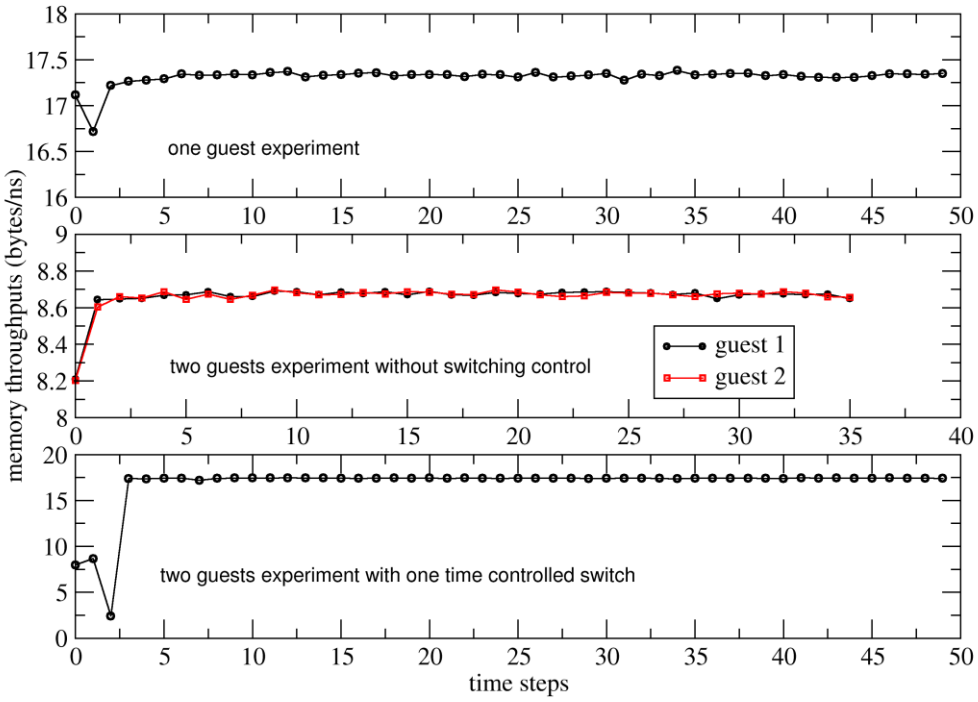

Figure 8 Memory throughputs from Stress benchmark for one guest running on lguest, two lguests running on lguest without switching control and two guests running on lguest with one-shot switching control.

Last but not least, we also ran Dhrystone benchmark to measure cpu frequency (or in terms of Dhrystone language, DMIPS). The results are quite intuitive, once a guest is switched back and the other is switched out, the DMIPS for the switched back guest immediately grows up close to the maximum value, so the curve is square-wave like (figures not provided).

5. Summary

In this project, we first modified Lguest hypervisor in linux kernel to (1) measure switching time when two guest OSs sitting in the hypervisor and (2) measure memory throughputs in perspective of guest OSs by adding extra functions into Lguest hypervisor to manually control the switching time between guest OSs and running different benchmark tools on guests. We performed various experiments for the above two targets and got a set of rough estimations on both switching time and memory throughputs. However, we found that because Lguest hypervisor actually does not entirely manage the scheduling of processes in Linux which is done by the host kernel scheduler,, in all our experiments we cannot effectively control interruptions introduced by the kernel. All the results we obtained, though they can be viewed as approximations of switching time and memory throughputs, are actually deviating from the real values. But we should point out that the idea and method we employed can be further investigated to measure memory performance for other virtual machine with fine grained control on scheduling, like Xen.

6. References

[1] http://oprofile.sourceforge.net/about/

[2] http://en.wikipedia.org/wiki/Dhrystone

[3] http://weather.ou.edu/~apw/projects/stress/

[4] J. Du, N. Sehrawat and W. Zwaenepoel, Performance Profiling of Virtual Machines . VEE 2011

[5] E. Khen*, et al.*. "Using virtualization for online kernel profiling, code coverage and instrumentation,". In *Performance Evaluation of Computer & Telecommunication Systems (SPECTS), 2011 International Symposium on*, 2011, pp. 104-110.

[6] McCalpin, John D., 1995: "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

[7] A. Menon, J. R. Santos, Y. Turner, G. (John) Janakiraman, W. Zwaenepoel, *Diagnosing Performance Overheads in the Xen Virtual Machine Environment*, First ACM/Usenix Conference on Virtual Execution Environments (VEE'05), 2005

Appendix A: Linux *CONFIG* File

Here we provide a listing of the changes made to the Linux kernel *config* file for compiling with Lguest support

CONFIG_EXPERIMENTAL=y
CONFIG_PARAVIRT=y
CONFIG_LGUEST_GUEST=y
CONFIG_HIGHMEM64G=n
CONFIG_PHYSICAL_START=0x400000
CONFIG_PHYSICAL_ALIGN=0x100000
CONFIG_VIRTIO_BLK=m
CONFIG_VIRTIO_NET=m
CONFIG_TUN=m
CONFIG_LGUEST=m
CONFIG_OLPC = n # this results in ``error: lguest: unhandled trap 13 at 0x100062 (0x0)”

Linux distributions on which we were unable run Lguest:
- CentOS 5 / 6
- Debian 6 (2.6.31 or above)
- Fedora 16

Appendix B: Modifications to Lguest Code

Here we provide a listing of the changes made to the Lguest code to measure the switching times
The modifications of the code for measuring the switching time in Lguest is as follows.

```
/driver/lguest/lg.h
/*DEFINE GLOBAL VARIABLES AND FUNCTIONS FOR MEASURING SWITCHING TIME*/
#define u64_INF 0xfffffffffffUL
extern unsigned int LG_COUNTER; //global counter for guest os, April 20 2012
extern volatile u64 MIN_RDTSC;  //minimum value for rtdsc
extern unsigned int LAST_ID; //to record the last running guest id
extern u64 LAST_RDTSC; //last rdtsc value
u64 rdtsc(void);
/*END  @author QZ @date 04/21/2012 */

struct lguest
+ unsigned int id; //add an id for each guest April 20 2012 Q

/driver/lguest/lguest_user.c
+
unsigned int LG_COUNTER = 0;
volatile u64 MIN_RDTSC = 0;
unsigned int LAST_ID = 0;
u64 LAST_RDTSC = 0;
/*DEFINITION ON rtdsc FUNCTION TO COUNT MINIMUM CPU CYCLES FOR SWITCHING BETWEEN TWO GUEST OS @author QZ
@date 04/21/2012 */
u64 rdtsc(void)
{
    //the x86 counter is 64 bits and is accessed via the RDTSC instruction
    //to count the number of cycles since reset
    u32 eax, edx;
    __asm volatile("rdtsc" : "=a" (eax), "=d" (edx));
    return ((u64)edx << 32) | eax;
}
/*END DEFINITION of rdtsc*/

+ write()
       case LHREQ_BREAK:
       {
               size_t len;

               /* We can only return as much as the buffer they read with. */
               len = sizeof(u64);
               printk("Sending the response, %lld, len: %d", MIN_RDTSC, len);
               if (copy_to_user(input, &MIN_RDTSC, len) != 0)
                       return -EFAULT;
```

```
                printk("Sent the response");
                return len;
        }
```

function: static int initialize(struct file *file, const unsigned long __user *input)
within mutex_lock and mutex_unlock

```
+/*START MODIFICATIONS FOR SWITCHING MEASUREMENT*/
+        lg->id = LG_COUNTER++; //assign id to the guest OS
+        //printk("guest id: %d¥n", lg->id);
+        if (LG_COUNTER == 2)
+        {//when there are two guest os, reset the LAST_ID, MIN_RDTSC and LAST_RDTSC
+         LAST_ID = lg->id;
+                MIN_RDTSC = u64_INF;
+         LAST_RDTSC = rdtsc();
+        }
+/*END @author QZ @date 04/21/2012*/
```

function: static int close(struct inode *inode, struct file *file)
within mutex_lock and mutex_unlock

```
+/*MODIFICATION: WHEN A GUEST IS CLOSED LG_COUNTER SHOULD BE GOES DOWN*/
+LG_COUNTER--;
+/*END OF MODIFICATION @author QZ @date 04/21/2012 */
```

/driver/x86/core.c
in function: static void run_guest_once(struct lg_cpu *cpu, struct lguest_pages *pages)
before "asm volatile" where real switching occurs

```
/*MODIFICATIONS: IF CURRENT GUEST ID IS DIFFERENT FROM LAST_ID, A SWITCHING OCCURS
 AND GET CURRENT RDTSC AND THE DIFFERENCE TO PREVIOUS ONE. KEEP THE MINIMUM DIFFERENCE
 OF RDTSC AND SET LAST_ID TO BE THE CURRENT ID UNTIL NEXT SWITCHING OCCURS*/
if (LG_COUNTER == 2 && cpu->lg->id != LAST_ID)
{
   u64 new_rdtsc = rdtsc();
   u64 delta_rdtsc = new_rdtsc - LAST_RDTSC;
   //printk("new_rdtsc: %lld ; delta_rdtsc: %lld¥n", new_rdtsc, delta_rdtsc);
   LAST_ID = cpu->lg->id;
   LAST_RDTSC = new_rdtsc;
   if (delta_rdtsc < MIN_RDTSC)
   {
       MIN_RDTSC = delta_rdtsc;
   }
}
/*END OF MODIFICATIONS @author QZ @date 04/21/2012 */
```

The diff of the code for controlling the switching in Lguest is as follows.

```
diff -r lguest//core.c linux-source-3.0.0/drivers/lguest//core.c
266,267c266
<         if (cpu->halted || !cpu->canRun) {
---
>         if (cpu->halted) {
316d314
diff -r lguest//lg.h linux-source-3.0.0/drivers/lguest//lg.h
73d72
<     int canRun;
93d91
<     u32 id;
121,123d118
< extern struct mutex vms_lock;
< extern struct lg_cpu *guests[2];
< extern u32 nr_guests;
diff -r lguest//lguest_user.c linux-source-3.0.0/drivers/lguest//lguest_user.c
14d13
< #include <linux/timer.h>
17,136d15
< u32 nrGuests = 0;
< struct lg_cpu *guests[2] = {NULL, NULL};
< DEFINE_MUTEX(vms_lock);
<
< u32 lastId = 1;
< u32 timerStarted = 0;
< unsigned long switchOnce = 0;
<
< u32 millisecs = -1;
<
< static struct timer_list my_timer;
<
< static void changeStateOfVM (struct lg_cpu *vm, int canRun)
< {
<     if (!vm || !vm->tsk)
<         return;
<     vm->canRun = canRun;
<     set_interrupt(vm, 0);
< }
<
< void my_timer_callback( unsigned long guest_id)
< {
<     int ret;
<     if (!switchOnce)
<     { // only reset the timer if switchOnce is false
<         ret = mod_timer( &my_timer, jiffies + msecs_to_jiffies(millisecs) );
```

```
<         if (ret)
<             printk("%s, Error in mod_timer, ret: %d¥n", __FUNCTION__, ret);
<     }
<     mutex_lock(&vms_lock);
<     // only do the switching if there are two guests
<     if (nrGuests == 2)
<     {
<
<         // stop the last running VM
<         changeStateOfVM(guests[lastId], (guests[lastId]->canRun)^1);
<
<         // toggle the last id
<         lastId ^= 1;
<
<         // interrupt the other VM; force it to run
<         changeStateOfVM(guests[lastId], (guests[lastId]->canRun)^1);
<     }
<     mutex_unlock(&vms_lock);
<
< }
<
361d239
<
363,368c241
<     if ((cpu->canRun) && (millisecs > 0))
<     {
<         return run_guest(cpu, (unsigned long __user *)user);
<     }
<     else
<         return 0;
---
>     return run_guest(cpu, (unsigned long __user *)user);
385d257
<     cpu->canRun = 1;
470,471d341
<     lg->id = nrGuests++;
489,493d358
<     if (lg->id < 2 && lg->id >= 0)
<     {
<         printk("Setting task for guest: %d¥n", lg->id);
<         guests[lg->id] = &lg->cpus[0];
<     }
542c407
<     unsigned long req, vmId, timerDuration;
---
>     unsigned long req;
545d409
```

```
<     int ret;
553c417
<     if (req != LHREQ_INITIALIZE && req != LHREQ_BREAK) {
---
>     if (req != LHREQ_INITIALIZE) {
566,616d429
<     case LHREQ_BREAK:
<         {
<             if (get_user(timerDuration, input) != 0)
<                 return -EFAULT;
<             input++;
<             if (get_user(vmId, input) != 0)
<                 return -EFAULT;
<             input++;
<             if (get_user(switchOnce, input) != 0)
<                 return -EFAULT;
<
<             if ( (nrGuests == 0) || // If there are no running VMs, OR
<                     !(vmId == 0 || vmId == 1) || // VM ID is not {0,1}, OR
<                     !(switchOnce == 0 || switchOnce == 1) ) // switchOnce is not {0,1}
<             return -EFAULT; // return invalid
<
<             // duration of timer
<             millisecs = timerDuration;
<             // start the timer once only if there is atleast one running VM
<             if (!timerStarted)
<             {
<                 ret = mod_timer( &my_timer, jiffies + msecs_to_jiffies(millisecs) );
<                 if (ret)
<                 {
<                     printk("Error in mod_timer¥n");
<                     return -EFAULT;
<                 }
<                 //timerStarted = 1;
<             }
< #if 0
<             else if (nrGuests == 1)
<             { // for user-controlled start and stop of one VM
<                 // toggle the canRun bit
<                 changeStateOfVM(guests[vmId], (guests[vmId]->canRun)^1);
<             }
< #endif
<             if (nrGuests == 2)
<             {
<                 // Keep running the VM whose id is vmId
<                 // stop the other VM
<                 lastId = vmId;
```

```
<                 changeStateOfVM(guests[lastId^1], 0);
<             }
<             return 0;
<         }
658d470
<         //lg->cpus[i] = NULL;
665,672d476
<     mutex_lock(&vms_lock);
<     nrGuests--;
<     if (guests[0] && !guests[0]->tsk)
<         guests[0] = NULL;
<     if (guests[1] && !guests[1]->tsk)
<         guests[1] = NULL;
<     mutex_unlock(&vms_lock);
728,732d531
<
<     // my_timer.function, my_timer.data
<     setup_timer( &my_timer, my_timer_callback, 0 );
<
738,743d536
<     int ret;
<
<     ret = del_timer( &my_timer );
<     if (ret) printk("Error in deleting timer.¥n");
```