École polytechnique fédérale de Lausanne
Operating Systems Laboratory (LABOS)

Analysis of live virtual machine migrations in VMware vSphere

Lukáš Frélich

M.S. Semester project
Supervised by: Dr. Jean-Philippe Martin-Flatin
Co-supervised by: Jiaqing Du
17 June 2011

Abstract

The objective of this M.S. semester project is to evaluate the behavior of vMotion technology used for live migrations of virtual machines in VMware vSphere, under different load types, in an environment that tries to a mimic system architecture being used in the real world. To meet this goal, we have built a testbed and a general-purpose monitoring and management tool, which collects periodically statistics from VMware vCenter, manages all migrations and clients and produces easy-to-analyze output. We have done extensive evaluations of performance statistics collected with VMware vCenter, and we generate charts of interesting metrics. We measured the migration time and the performance of applications running in virtual machines, during and after migration, and uncovered a fundamental limitation in VMware software: the monitoring granularity has a lower bound of 20 seconds.

1 Introduction

Since 2007–2008 server virtualization has been widely used in the enterprise world. One of the reasons for its success is the ability to do things that are simply not possible with physical servers. The typical example is the live migration of virtual machines, which essentially means relocating a virtual machine being run on one physical host to another with virtually zero downtime. This capability usually requires the source and target hosts involved in the live migration, to use shared storage, which most of the time does not come for free. But nevertheless it is widely used in practice. Thanks to it, planned hardware maintenance can be done without any disruption to services. Migrations can also be done automatically depending on the load of the hosts. This load balancing is very useful when running applications not designed with scalability in mind. Moreover, when used in conjunction with other technologies live migrations can save energy by shutting down servers at slack hours and restarting them when needed.

One of the companies that brought the idea of live migrations to reality was VMware [9], which is now the leader of the virtualization industry. Since its introduction in 2003, VMware's virtualization technology is called vMotion. Apart from ESX or ESXi hypervisors, which must be installed on all hosts, a customer who wants to take advantage of vMotion must also run vCenter, which provides centralized management and monitoring of virtualized datacenters. This software only runs on Microsoft Windows and needs to be installed on a dedicated machine.

In 2005 VMware published a short paper [1] briefly explaining the main idea behind vMotion and giving some basic performance charts. Another implementation of this feature was developed in the academia on top of the XEN hypervisor and published in 2005 [2]. This highly cited paper not only explains in detail how the solution works but also gives a deep look into its performance. By comparing [1] and [2], it seems that the basic idea behind the two implementations is the same. However, details of the VMware technology have not been published.

The goal of this project is to evaluate the behavior of the vMotion technology under different load types, in real world conditions. To do that we built a testbed and developed a general-purpose monitoring and management tool, which collects periodically some statistics from vCenter, manages all migrations and clients, and produces easy-to-analyze output.

The testbed consists of several servers (which are in our case virtual machines) processing requests from clients. Since there are many requests and the application itself was not designed to be used in a cluster, there is a load balancer between clients and servers. We further assume that sending request is very cheap whereas generating response is costly.

During the development process, we found out that vCenter gathers performance statistics only once every twenty seconds and that it is not possible to change this setting [3] [4]. Moreover, hosts themselves are also gathering performance statistics once every twenty seconds. This metric, officially called *real-time performance monitoring* by VMware, is in our opinion not real-time enough. We therefore decided to lengthen time needed to generate response and eventually shifted a bit to explore the behavior of vMotion more broadly.

The rest of this report is organized as follows. In the Section 2, we present the test bed and the software developed for performance evaluation. In section 3, we describe the experiments we used, and analyze data we gathered. Finally, we conclude in Section 4.

2 Architecture

All tests were performed on Dell SC1425 servers. We used up to 10 servers, each of them featuring:

- 2x Intel Xeon 3.8 GHz processor (Hyper-threading enabled)
- 4 GB RAM
- 2x Intel Pro 1000/MT Gigabit Ethernet NIC

The overall architecture of the testing environment is depicted in Figure 1.

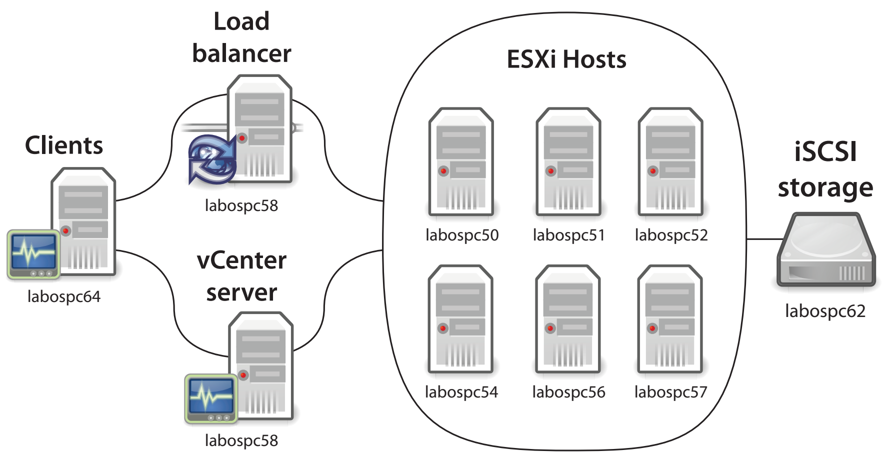

Figure 1: Testbed.

2.1 Clients and monitoring

Perhaps the most important part of the project was developing software for running semi-automatic tests in the datacenter. Our tool, called *Vm-monitoring*, is responsible for carrying out actions according to a given script file and for monitoring performance. Details about this software and the format of the script file can be found in Appendix A. In the first line of the script, *Vm-monitoring* expects to find a list of hosts and virtual machines running on them. It automatically migrates the virtual machines accordingly. Every other line of the script constitutes an action to be performed at a time specified in the script. There are four commands, whose names are self-explanatory: *add-client*, *remove-client*, *migrate* and *end*.

Clients are of different load type (see Section 2.6), and each runs in a separate thread. A client does not send a request directly to a specific server, but rather to the load balancer, which forwards the request. In response, the server returns how much time the request took to process and the server identifier; *Vm-monitoring* collects those responses. This makes it easier to correlate host performance to client behavior than if monitoring and clients were managed by two different applications.

Live migration means sending vCenter a request for migration and then waiting for the result. This is always done in a separate thread in order to not affect other parts of the application, which communicate with vCenter. It also allows supporting multiple migrations simultaneously. VCenter 4.1 Update 1 (the latest version when this work was performed), supports up to four concurrent migrations per host via 1 Gbit network [5]. If a migration is triggered when the maximum number of migrations is already reached, *Vm-monitoring* postpones this migration as well as all other outstanding actions.

Finally, *Vm-monitoring* polls vCenter every 20 seconds to get statistics about all hosts. Those statistics are printed, as well as information about all actions, in CSV format to standard output.

Vm-monitoring was used on a dedicated server running Ubuntu 11.04 and Java 1.6.0_24.

2.2 Load balancer

Load balancing was done using Apache HTTP Server 2.2 and mod_proxy_balancer on Ubuntu 10.10. The same machine also served as a DNS server for look-ups within the cluster. Using DNS made it easier to deal with a rather large number of virtual machines and facilitated some last-minute changes.

2.3 vCenter

VCenter coordinates multiple hosts within a datacenter. It is required for most of advanced features, including vMotion. We used version 4.1 Update 1 on Windows Server 2008, Service pack 2.

2.4 Hosts and storage

Our testbed also included six servers running ESXi, again version 4.1 Update 1. As required by vMotion, a common storage was used in a form of an iSCSI target (another server running Ubuntu 10.10 with iSCSI Enterprise Target). Additionally, we used NFS to share server applications among all VMs.

2.5 Virtual machines

All virtual machines (VMs) were configured to use one virtual CPU and 512 MB RAM. From a software perspective, we used Ubuntu 11.04 without VMware Tools.

2.6 Load types

From the very beginning, we wanted to test various load types to assess the differences between CPU-bound, memory-bound and IO-bound applications. We had to drop the IO-bound scenario because our iSCSI target had two old disks and was connected to all hosts using a single 1 Gbit Ethernet NIC.

There was also a question if we should use a standard benchmark or program simple 'server applications' from scratch. In the end, we opted for the latter because in most of the cases, we needed the request processing time to be quite long (to circumvent the problem of low refresh rate), but sometimes we actually needed it to be short. We implemented one CPU-bound and two memory-bound tests:

- *cpu* – high-precision approximation of π, using very little memory. The duration can be easily changed by setting the desired precision.
- *memory* – memory allocation using malloc function and filling the allocated memory with numbers. This test was designed to use a lot of memory whose content is not changing, and little CPU time. After filling the allocated memory, the program is put on sleep for a time given as a command-line parameter. The size of allocated memory is also given as a parameter (in all tests this value was set to 100 MB).
- *memory2* – this test is almost the same as *memory*, but instead of waiting the program keeps filling the allocated memory (adjustable via command-line parameter, in our tests always 100 MB) in a loop. Designed to use a lot of memory whose content is changing all the time, which makes the migration as hard as possible.

3 Results

3.1 Interesting metrics

The VMware API supports for hosts 129 different performance metrics (internally called *counters*). Because each server has more than one processor and network card, each query for performance counters returns about 560 {key, value} pairs. However, *Vm-monitoring* automatically makes an average of multiple instances of the same counter.

To find the most interesting counters for the load types we used, the following scenario has been designed. Twenty VMs are serving 20 (in *cpu* load) or 40 clients (in *memory* and *memory2* load). One host (let us call it target) is chosen for performance monitoring and starts with zero VM. Every 2000 seconds, one VM is migrated to target until we have six VMs running there. The process is then reversed and goes at the same pace as before.

The basic processing time for one client request was set to about 200 seconds. This way we wanted reduce the negative effect of low counters refresh rate. Even longer time turned out to be very impractical: load-balancer occasionally assigns more requests to one server then it should. In that situation VM gets quickly overloaded and the processing time soars quickly up. The test ran separately for each type of load and all hosts were restarted before every test.

All charts (Figures 2 through 6) plot four variables: the first three are the load types; the last is the number of VMs running at the moment on the host.

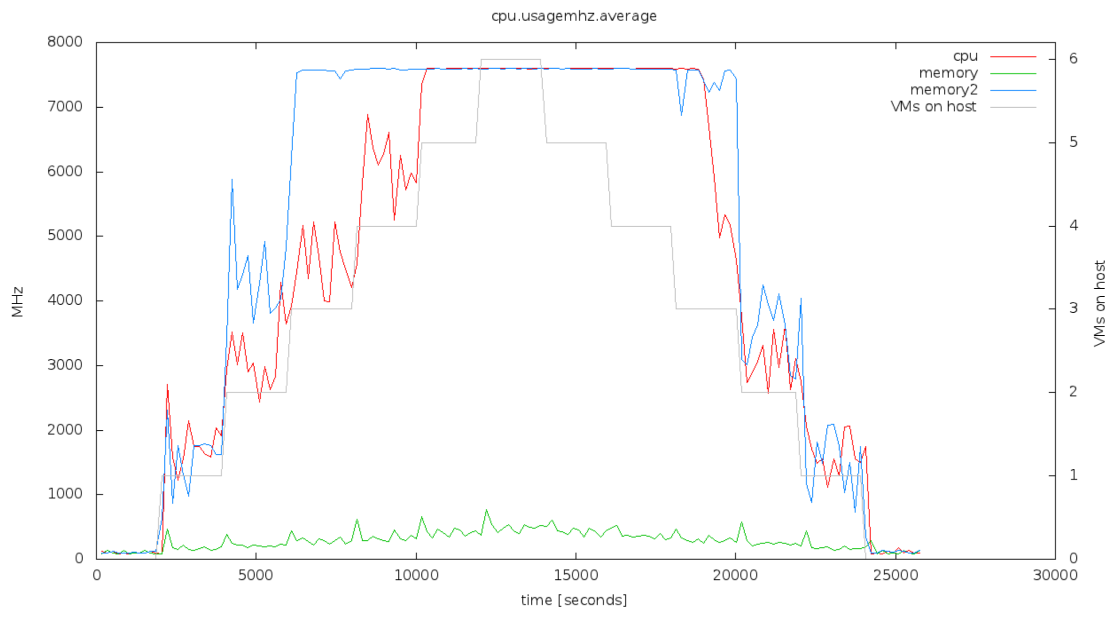

Figure 2: CPU utilization.

Figure 2 shows CPU utilization. The maximum value is equal to the sum of all CPUs frequencies, in our case 2×3800 MHz = 7200 MHz. As expected, both *cpu* and *memory2* highly utilize CPU, whereas *memory* uses very little CPU. In fact, *memory2* utilizes CPU even more, but that is only caused by the number of clients (40 vs. 20). In addition, it took quite long until the utilization finally dropped after a heavy load. The explanation is that the VMs were simply overloaded and it took a while to process all requests in the queue.

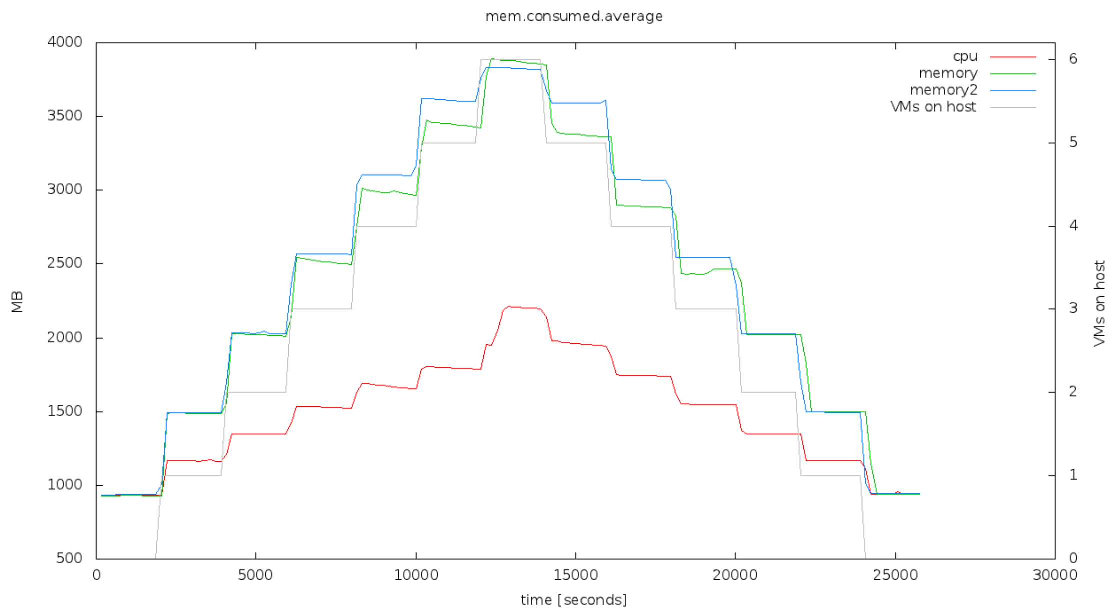

Figure 3: Memory being really allocated.

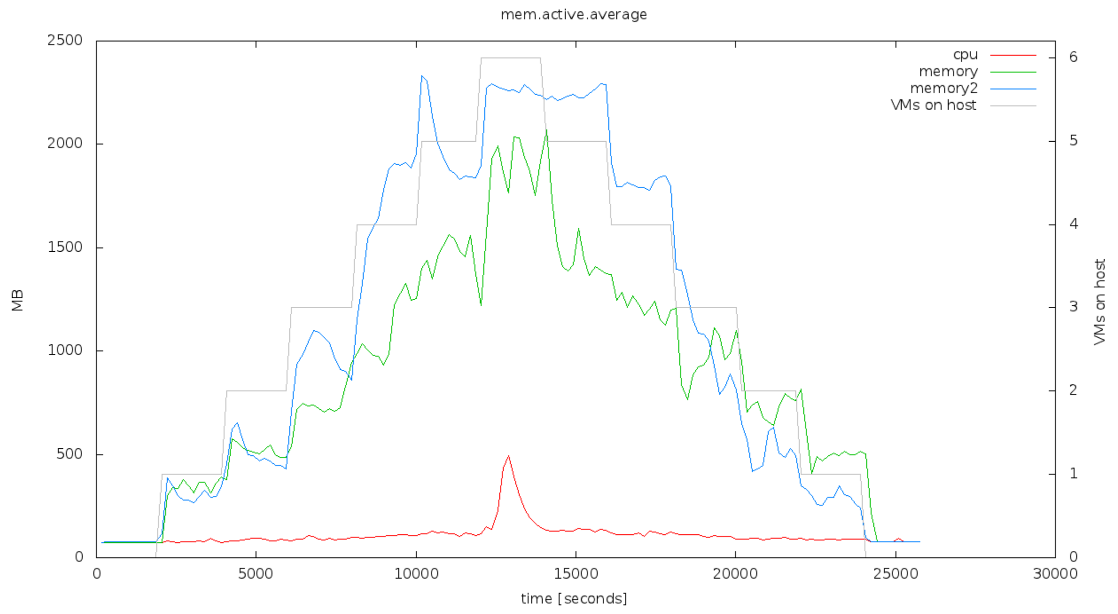

Figure 4: Memory actively in use.

The next two charts (Figures 3 and 4) are analyzed together, because they are closely related. *Mem.consumed.average* is memory in use by VMs, VMware kernel and so forth [6]. It should be noted that this number is not a simple summation of memory assigned to the VMs: some savings are possible thanks to memory sharing (see Figure 5). This counter for all scenarios precisely follows the migrations schedule: after a migration is done, we can immediately see the change in the counter. In *mem.active.average* the host estimates how much memory has been actively used in the last time interval [7]. By comparing Figures 3 and 4 we can see how much memory was used without being touched, especially in the CPU test.

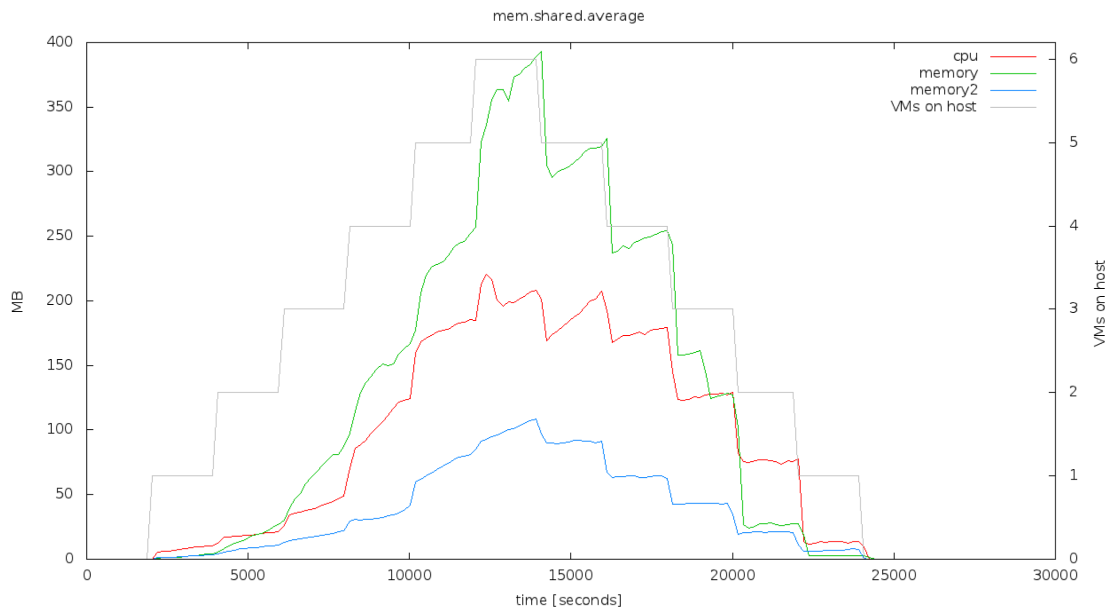

Fig. 5: Memory savings.

Mem.shared.average shows how VMkernel saves memory by storing parts of memory being used by several VMs only once. This works particularly well in our environment, where all the VMs are essentially the same. Details are beyond scope of this project (basics are well explained in [8]), but the chart makes a lot of sense – for *memory2*, not much can be shared, since the memory is changing all the time. The *cpu* test is better, but there is not much to be shared anyway, and *memory* is the best for obvious reasons.

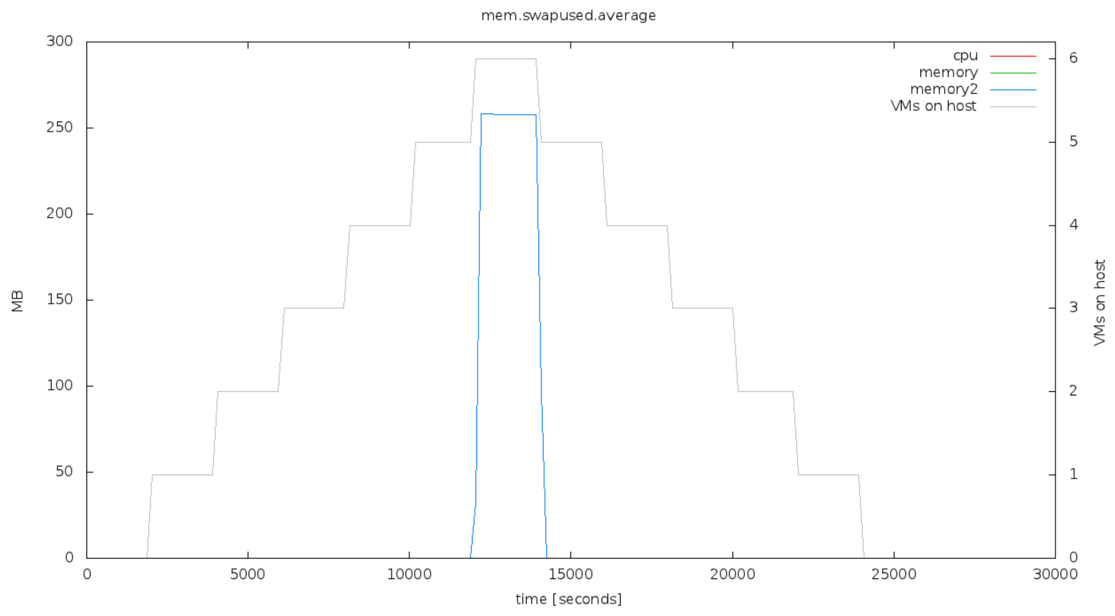

Fig. 6: If possible, we should avoid using swap.

Finally, *mem.swapused.average* (Figure 6) shows in the middle what should not happen. In the *memory2* test the host was so overloaded that it had to resort to swapping. Fortunately, VMkernel freed the swap as soon as possible, making the performance drop not as noticeable.

3.2 Migration time

In our second set of experiments we measured what is the migration time (i.e. the time between triggering migration and successful completion). We again used long-time load (about 200 seconds to finish one request), but this time we had two test runs for each load type.

In the 'light' case, five VMs were being migrated among six hosts in a circle, so at all time there was at most one VM per host. In total 50 migrations were performed. In the 'heavy' case, we used 20 VMs and 5 hosts and did 60 migrations again in a circle manner. For the 'light' case we used 5 clients, for the 'heavy' case we used 20.

load	scenario	min [ms]		max [ms]		average [ms]		std. dev. [ms]
cpu	light	7547		18219		12582		64
cpu	heavy	9823	+30 %	19523	+7 %	14171	+13 %	3
memory	light	8960		17727		13532		106
memory	heavy	9870	+10 %	17120	−3 %	13511	0 %	18
memory2	light	10592		22263		15086		123
memory2	heavy	13495	+27 %	30956	+39 %	20045	+33 %	19

Table 1: Measuring migration time under different conditions.

The differences between the light and the heavy scenario are worth a deeper look. For the *memory* test, there is no difference at all and it makes sense – this load only allocates memory and then sleeps, so there is no such thing as a 'heavy' case here.

With the *cpu* test we can certainly notice an increase in all statistics when we hit the 'heavy' case. For *memory2* the difference between 'heavy' and 'light' is very noticeable.

Comparison of the load types among themselves fits what was described for the XEN hypervisor in [2] – the more memory we need to transfer and the more it is changing, the longer the live migration takes.

3.3 Server performance

Finally, we wanted to know if there is any performance drop perceived by a client during a live migration. To find out the answer, we drastically changed the testbed. We had only one client and one VM migrating in cycle across all six hosts. Server applications were changed so that response-processing time went down to 100–200 ms depending on the load type. We measured how many responses we got per second and how much time the processing took during live migration and during regular operation. We performed 100 migrations for each type of load.

load	phase	responses · s^{-1}		min [ms]		max [ms]		average [ms]		std. dev. [ms]
cpu	migration	5,00		116		621		132		64
cpu	normal	6,21	+24%	115	−1%	150	−76%	122	−8%	3
memory	migration	1,40		197		606		281		106
memory	normal	2,07	+48%	186	−6%	271	−55%	227	−19%	18
memory2	migration	1,30		212		665		303		123
memory2	normal	2,01	+55%	196	−8%	293	−56%	240	−21%	19

Table 2: Server performance during and after migration.

We can see in Table 2 that the differences are not negligible. We did not have any means to monitor downtime during the migration, so we can only speculate that the reason lies really in the downtime.

Though the differences are not insignificant, they should not be observable in real-life applications. Live migrations seem to be quick enough so that no one would notice the performance drop.

4 Conclusion

Over the last four months, we have successfully designed and deployed a testbed which served very well for all the experiments we used. We have also developed software, which enabled us to assess the behavior of vMotion technology under three different types of load. Using it we found which performance counters provided by vCenter are worth deeper look and analysed them. We measured the live migration time in 'light' and 'heavy' cases for each of used load type, as well as how migration affects the server performance really perceived by a client. Last but not least, the software we developed can be easily used in other projects for performing different tests in similar testbed.

There is number of directions for further research and development. First, it would be useful to add support for measuring the downtime experienced during the live migration and refine our results in Section 3.3.

We would also like to better understand memory management. There are many performance counters related to the memory subsystem and differences between them are often subtle and require a deep understanding of how the VMkernel works.

Third, the Distributed Resource Scheduler (DRS) technology, allows VMs to be automatically migrated, depending on the load of the system. With Distributed Power Management (DPM) it is also possible to shut down servers when they are not needed and start them when the load soars up. Assessing the impact of DRS and DPM would be interesting. An aggressive DRS policy might entail that migrations are performed very often, which would directly impact the performance of applications.

References

[1] M. Nelson, B. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX 2005*, 2005.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/ USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[3] VMware Communities: PerfProviderSummary - refreshRate always 20 secs? http://communities.vmware.com/message/494013?tstart=225

[4] VMware Communities: Change refreshRate http://communities.vmware.com/thread/316257

[5] VMware: VMware vCenter Server Performance and Best Practices http://www.vmware.com/files/pdf/techpaper/vsp_41_perf_VC_Best_Practices.pdf

[6] Marie McGarry, VMware vSphere Host Metrics http://support.hyperic.com/display/EVO/VMware+vSphere+Host+Metrics

[7] VMware Communities: vCenter Performance Counters http://communities.vmware.com/docs/DOC-5600

[8] VMware: The Role of Memory in VMware ESX Server 3 http://www.vmware.com/pdf/esx3_memory.pdf

[9] VMware: VMware Introduces the VMware VirtualCenter Suite for Blade Servers http://www.vmware.com/company/news/releases/vc_blades.html

[10] VMware VI (vSphere) Java API http://vijava.sourceforge.net/

Appendix A: Description of Vm-monitoring

Dependencies

The API provided by VMware is in fact a web service. This solution is truly multiplatform, but for most popular programming languages, it makes things more complicated. Steve Jin encapsulated web services in his VI Java API [10] and allowed Java developers to work as they are used to. Using this library simplified our work greatly.

For simulating clients we used the Apache HttpClient library.

Basic concepts

After parsing the script file Vm-monitoring first starts the monitoring thread, which keeps running until the program is finished. For performing an action, we use the TimerTask object, which is scheduled to be called every two seconds. Actions might be therefore slightly delayed compared to the time stated in the script. When the TimerTask object is woken up, it triggers all actions whose time has already come. Adding or removing clients means only starting or stopping ClientsThreads. Migration is done in a separate thread, and the application needs to watch if the maximum number of concurrent migrations has been exceeded.

Script file format

Vm-monitoring expects one command line parameter: path to the script file. All blank lines and lines beginning with character '#' are ignored.

First line contains the list of hosts and VMs running on them, in the following format:

```
[host]:[list-of-VMs]?
```

where both host and VM names must be known by vCenter. The list of VMs is comma-separated and is not required. This entry may repeat, in which case entries are separated by white space.

Other lines contain actions:

```
[time] [action] [parameters]
```

where time is given in seconds and is relative to previous action. Table 3 lists actions and their parameters.

action	parameter1	parameter2
add-client	type (cpu/memory/memory2)	number of clients
remove-client	type (cpu/memory/memory2)	number of clients
migrate	VM name	target host
end		

Table 3: Actions and their parameters.

Sample script files can be found in the /sample directory of the project.