Live Migration of Virtual Machine Based on Full System Trace and Replay

Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, Chen Yu
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@hust.edu.cn

ABSTRACT
Live migration of *virtual machines* (VM) across distinct physical hosts provides a significant new benefit for administrators of data centers and clusters. Previous migration schemes focused on transferring the run-time memory state of the VM. Those approaches employed memory pre-copy algorithm to synchronize the migrating VM states, which make VM live migration cost much network traffic and application downtime, especially for memory-intensive workloads. This paper describes the design and implementation of a novel approach CR/TR-Motion that adopts checkpointing/recovery and trace/replay technology to provide fast, transparent VM migration. With execution trace logged on the source host, a synchronization algorithm is performed to orchestrate the running source and target VM until they get a consistent state. We also give a formalized characterization about the migration evaluation metrics and make a mathematical analysis about our algorithm. Our scheme can greatly reduce the migration downtime and network bandwidth consumption. Experimental measurements show that our approach can drastically reduce migration overheads compared with pre-copy algorithm: up to 72.4% on application observed downtime, up to 31.5% on total migration time and up to 95.9% on the data to synchronize the VM state, while the application performance overhead due to migration is less than 8.54% on average.

Categories and Subject Descriptors
D.4.7 [**Operating Systems**]: Organization and Design—*Distributed System*; C.4 [**Performance of Systems**]: Design studies

General Terms
Algorithm, Design, Measurement, Management, Performance

Keywords
Virtual Machine, Live Migration, Checkpoint, Trace, Replay

1. INTRODUCTION
The use of *virtual machine* (VM) migration technology for data centers management has attracted significant attention in the recent years [6, 7, 17, 23, 27]. The capability of migrating live virtual machine among distinct physical hosts provides a significant new benefit for multiple VM-based environments. Live migration of virtual machines is an extremely powerful tool for cluster administrators in many key scenarios:

1) Load balancing, VMs may be rearranged across physical machines in a cluster to relieve load on congested hosts.

2) Online maintenance and proactive fault tolerance [16], sometimes, a physical machine may need upgrade or servicing for upcoming system faults, an administrator should migrate the running VMs to alternative machine(s), freeing the original machine for maintenance. So live VM migration improves system serviceability and availability.

3) Power management [18], usually, the load and throughput of servers are uneven but statistically regular at different periods. When some VMs resided in distributed hosts are running light-load jobs, which can be consolidated into fewer hosts, the off-loaded hosts may be decommissioned once migration is completed. This strategy helps corporations to reduce IT operation expenses and benefit the natural environment.

In such situations, the combination of virtualization and migration significantly improves manageability of data centers and clusters. For the above requirements, there are many literatures that related to implementing high performance migration methods [6, 7, 10, 17]. The most influential approaches are VMotion [17] and XenMotion [7] which were shipped by VMware and XenSource as parts of their products respectively. Their implementation mechanism is similar, because they have the same applying scenarios (in a LAN) and analogical scheme for migrating physical memory and network connections.

In each solution, there are mainly three kinds of states that should be migrated: the VM's physical memory; the network connections and virtual device state; the SCSI storage. The most intractable issue is migrating physical memory, because it is the main factor that affects the migration *downtime*, *i.e.*, the time during which the services on the VM are entirely unavailable. VMotion and XenMotion adopted pre-copy algorithm [17, 7] to address this issue. Although the memory pre-coping algorithm is able to decrease the best case downtime to the magnitude of millisecond, there are still some unsolved issues which should be considered further. First, when the rate that memory pages dirtied is faster than the replication rate of pre-copy procedure, all pre-copy work will be ineffectual and one should immediately stop the VM and

HPDC'09, June 11–13, 2009, Munich, Germany.

copy the entire memory pages to the target host. Some memory-intensive workloads would get no benefit from pre-copy algorithm and the downtime may rise to several seconds. This limitation also makes the algorithm only applicable in high speed LANs. Second, some para-virtualized optimization schemes, such as stunning rogue processes and freeing unallocated pages that are mentioned in XenMotion [7] may cause some negative effect to users' experience, especially for some latency sensitive interactive services. At last, pre-copy algorithm does not recover the CPU's cache data. Although it may not lead to any mistake on the target host, massive cache and TLB missing may cause performance degradation once the VM takes over the service.

In this paper, we propose a novel live VM migration approach – CR/TR-Motion. We implement our prototype based on a full system trace and replay system – ReVirt [9]. Checkpointing/recovery and trace/replay technology are adopted to provide fast, transparent VM migration in a LAN. A trace daemon continuously logs the non-deterministic events of the VM while sacrificing very little performance. The execution trace file logged at the source host is iteratively transferred to the target host and used to synchronize the migrated VM's execution state. Experimental measurements show that our approach can drastically reduce migration time and network traffic compared to pre-copy algorithm.

The contributions of this paper are mainly listed as follows: 1) we design and implement a novel approach that uses checkpointing/recovery and trace/replay technology to minimize the VM migration downtime and network traffic; 2) we implement a transparent VM checkpoint with copy-on-write mechanism; 3) we make a formalized characterization about the migration metrics and give a mathematical analysis of the algorithm's performance.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction about our groundwork. Section 3 presents the design of our VM migration algorithm and gives a theoretical analysis about our approach's performance. Section 4 describes the detail implementation of our approach. Section 5 presents the experiments undertaken and results obtained. Section 6 introduces the related work about VM migration. Finally, we conclude our work in section 7.

2. DETERMINISTIC REPLAY WITH EXECUTION TRACE

Checkpoint/recovery [4，8] and trace/replay [9, 11, 19, 26] technology are used widely for recovering system state. The basic concept is straightforward: starting from a checkpoint of a prior state, and then rolling forward using the log to reach the desired state. Replaying a system requires logging the non-deterministic events that affect the system's computation. These log records guide the system as it re-executes (rolls forward) from a checkpoint. Most events are deterministic (e.g. arithmetic, memory, branch instructions) and do not need to be logged; the process will re-execute these events in the same way during replay as it did during logging. Non-deterministic events fall into two categories: time and external input. Time refers to the exact point in the execution stream at which an event takes place. External input refers to data received from a user or another computer via a peripheral device, such as a keyboard, mouse, or network interface card. However, output to peripherals will be reconstructed during replay and hence need not be saved.

ReVirt is a typical full system logging and replay tool ported on UMLinux [5, 12] and designed for intrusion detection. It logs enough information to replay a long term execution of the virtual machine instruction-by-instruction. ReVirt only logs asynchronous virtual interrupts and sufficient information to re-deliver the signal at the same point during replay. It plays back the original asynchronous virtual interrupts using the combination of the hardware counters and host kernel hooks that are used during logging. Replay can be executed on any host with the same type of processors as the source host. There is not any deviation generated during the replay process.

Moreover, ReVirt adds reasonable time and space overhead. Overheads due to virtualization are imperceptible for interactive use and CPU-bound workloads, and 13-58% for kernel-intensive workloads. Logging adds only up to 8% performance overhead. Workloads with little non-determinism (e.g. kernel-build) generate very little log traffic. Even the I/O intensive workloads (dynamic web applications) generate feasible log traffic at a rate of hundreds of kilobyte per second. Log growth rates range from 0.04GB per day to 1.2GB per day for the related workloads [9].

The slow speed of log traffic growth and VM mobility inspires us to achieve live VM migration with checkpoint and execution log files. The strategy is straightforward: we first make a transparent checkpoint of the running VM in a copy-on-write fashion; second, the source VM continues running while the checkpoint is pushed across the network to the target host, the generated log files during the transmission interval must be iteratively sent to the target host to ensure the consistency of the both side; finally, the source VM is stopped, and the last log file is copied to the target VM, then the migrated VM replays with this log file and takes over the source VM's service.

3. ALGORITHM DESIGN

This section describes our design of live VM migration scheme based on checkpointing/recovery and trace/replay technology, we name this scheme as CR/TR-Motion for short. A synchronization protocol is detailedly introduced to show how we use execution trace to orchestrate the migrating source VM and target VM until they get a consistent state. We also give a formalized characterization about the evaluation metrics and make theoretic analysis about the migration performance. It will help us to optimize the migration algorithm in system implementation.

3.1 Design Objectives

Live virtual machine migration takes a running VM and moves it from one physical machine to another. This process must be transparent to the guest operating system, applications running on the operating system, and remote clients of the virtual machine. We address this issue and make tradeoffs among the following performance evaluation metrics involved in local-area migration.

1) Downtime: the time when no CPU cycle is devoted to any of the VM-resident applications, neither at the source nor at the target system, it consists of the time necessary to suspend the VM on the source, transfer the VM state to the destination, load the device state, and activate the migrated VM on the remote host.

2) Total Migration Time: the duration between the time migration is initiated and the time the migrated VM gets a consistent state with the original one, i.e., during which the state of two VMs is synchronized.

3) Total Data Transmitted: the total data is transferred while synchronizing the both VMs' state.

When a VM is running a live service, it is necessary to make a tradeoff to ensure that the migration occurs in a manner that may minimize all the three metrics. Our motivation is to design a live VM migration scheme with negligible downtime, lowest network bandwidth consumption and reasonable total migration time. Furthermore, we should ensure that the migration would not disrupt other active services residing in the same host through resource contention (e.g., CPU, network bandwidth).

3.2 Live Migration Process

This section describes the design of our live VM migration approach combining with instructions execution trace and replay. Unlike memory pre-coping algorithms, our method employs the target host's computation capability to synchronize the migrated VM's state. What we copied is the execution log of the source VM but not the dirty memory pages, and this may greatly decrease the amount of data transferred while synchronizing the two VM's running state. Our approach reduces the downtime by combining a bounded iterative log transferring phase with a typically short stop-and-copy phase. By iterative we mean that synchronization occurs in rounds, in which the log files to be transferred during round *n* are those generated during round *n*-1 (the checkpoint file is transferred in the first round). After several rounds of iteration, the last log file transferred in the stop-and-copy phase is reduced to a negligible size so that the downtime can be decreased to an unperceived degree.

Like the limitation of pre-copy algorithm, the dirty memory pages must be transferred faster than that they are dirtied, there are also some prerequisites for our approach. It is obvious that the log transfer rate should be faster than the log growth rate. Otherwise, our algorithm would be useless because log files will quickly accumulate on the source host. Fortunately, the log data grows far more slowly than it is transferred even when the source VM is running an OS-intensive or I/O-intensive workload [9]. The same outcome has also been presented in other analogical works [19, 26]. Our experiments also show the same conclusion for typical server workloads presented in table 1. For most workloads, the log growth rate is not more than 1MB/sec, which is much less than the network transfer rate in a Gbit/s network. Another requirement of our approach is that the log replay rate must be faster than the log growth rate. If this condition is not satisfied, the downtime may be even much longer than the elapsed time transferring the checkpoint file from the source host to the destination. Generally, the replay speed can be faster than the original execution with logging, because during normal execution a process may block waiting for I/O events, while during replay all events can be immediately replayed due to the ability of skipping over the idle time of HLT instructions [9, 26]. We assume the two above prerequisites are satisfied in the following description and discussion.

Figure 1 shows the whole process migrating a running VM from host *A* to host *B*. We view the migration process as a transactional interaction between the two hosts involved the following phases:

1) Initialization: a target host with sufficient resources is selected to guarantee the requirement of receiving migration. A good choice may speed the upcoming migration and boost up the server's QoS.

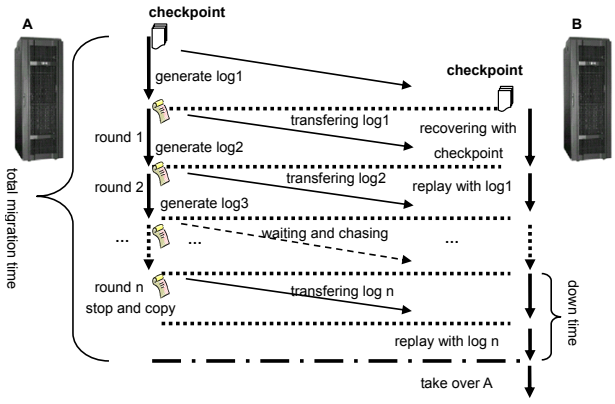

Figure 1. Process of live VM migration

2) Reservation: host *A* makes a request of migrating a VM to host *B*. A VM container of the source VM's size should be reserved to guarantee the necessary resources are available on host *B*.

3) Checkpointing: the VM on top of host *A* freezes, the system state (virtual main memory, CPU registers, memory from external devices and virtual disk) at the current instant are saved to an image file in a copy-on-write fashion. After checkpointing, the source VM continues to run as though nothing had happened.

4) Iterative Log Transferring: during the first round of transferring, the checkpoint file is copied from host *A* to *B*, while the VM on host *A* is continuously running and non-deterministic system events are recorded in a log file. Subsequent iterations copy the log file generated during the previous transfer round. At the same time, host *B* is replaying with the received log files once it had recovered from the checkpoint. As the log is transferred much faster than the log generated, this iterative process is convergent.

5) Waiting-and-Chasing: after several rounds of iteration, when the log file generated during the previous transfer round is reduced to a specified size (we defined this threshold value as V_{thd} in section 3.3 and set its default value as 1KB in our experiments), host *A* inquires *B* whether the stop-and-copy phase can be executed soon, if the resumed VM on host *B* does not replay fast enough, *i.e.*, the cumulative unused log file size on host *B* is still larger than V_{thd} at this time, host *B* should inform host *A* to postpone the stop-and-copy phase until the log is used up on host *B*. The iterative log transferring should be continuously performed till the size of unconsumed log at host *B* is reduced to V_{thd}. As the log replay speed on host *B* is faster than log generated speed on host *A*, the migrating VM on host *B* would chase up the running state of the source VM finally.

6) Stop-and-Copy: the source VM is suspended and the remaining log file is transferred to host *B*. After the last log file is replayed, there is a consistent replica of the VM at both *A* and *B*. The VM at *A* is still considered to be primary and may be resumed in case of failure.

7) Commitment: host *B* informs *A* that it has successfully synchronized their running states. Host *A* acknowledges this message as commitment of the migration transaction, and then all its network traffic is redirected to host *B*. Now the source VM may be discarded.

8) Service Taking Over: the migrated VM on host *B* is activated now, and the new VM advertises its moved IP address. Host *B* becomes the primary host and takes over host *A*'s service.

This approach should be a fault tolerant process. The source host should remain a stable state no matter which sort of failure occurs during migration. This guarantees the service continuously running on the source VM with no risk of failure until the migration commits.

3.3 Algorithm analysis

In this section, we discuss the above algorithm in two scenarios with formalized characterization, which would direct the further system implementation and performance evolution. Some important notations and their corresponding definitions are listed as follows:

R_{log}: log growth rate, which denotes the average growth rate of the source VM execution trace for a special workload.

R_{trans}: log transfer rate, which mainly lies on the network bandwidth between the two hosts. It is also an average value.

R_{replay}: log replay rate, which denotes the average rate of replay with the log files on target host.

V_{thd}: the threshold value of log data size at which the iterative log transfer procedure should be terminated.

We define the log file list transferred at each rounds as $L=<log_1, log_2 \ldots log_n>$, and their file size as $V=<V_{log_1}, V_{log_2}, \cdots, V_{log_n}>$ correspondingly. The elapsed time sequence at each transferring round is defined as $T=<t_0, t_1, t_2, \ldots, t_n>$, while t_0 presents the elapsed time to transfer the checkpoint of the source VM and V_{ckpt} denotes the data size of the checkpoint file.

In the following analysis and experiments, the V_{thd} is set as default value (1KB). To make our model simple, we deem that the R_{trans} in different transferring phase is a constant value. We mainly concern the three performance evaluation metrics in two scenarios: *fast synchronization* and *slow synchronization with waiting-and-chasing*. The discriminate of the two scenarios is that there is an additional waiting-and-chasing phase performed in the second scenario compared with fast synchronization.

3.3.1 *Fast Synchronization*

In this scenario, the log replay rate is much higher than the log growth rate ($R_{replay} >> R_{log}$). For instance, if the VM is running a daily use workload, the R_{replay} is 33 times larger than R_{log} according to the ReVirt's experiment [9]. In this condition, the log file size is reduced to the threshold V_{thd} soon in bounded rounds of iteration. Replay is executed on the target host so fast that there is no need to perform the *waiting-and-chasing* phase. The detail migration process is shown in Figure 2. The log transfer rate and log growth rate are the main factors affecting the process of migration.

Because $R_{trans} > R_{log}$, the iteration of log transferring process is convergent. After several rounds of iteration, the last log file size may reduce to the fixed value V_{thd}. The elapsed time in each round can be calculated like this: $t_0 = \frac{V_{ckpt}}{R_{trans}}$, $t_1 = \frac{R_{log} t_0}{R_{trans}}$, …, $t_n = \frac{R_{log} t_{(n-1)}}{R_{trans}} = \frac{V_{ckpt} R_{log}^{(n-1)}}{R_{trans}^{n}}$, where t_0 presents the time cost to transfer the data of checkpoint, and t_n presents the time cost to transfer the log file generated during previous round. Let φ ($0<\varphi<1$) denote the ratio of R_{log} to R_{trans}:

$$\varphi = R_{\log} / R_{trans} \tag{1}$$

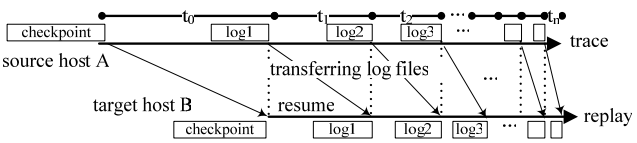

Figure 2. Migration process of fast synchronization

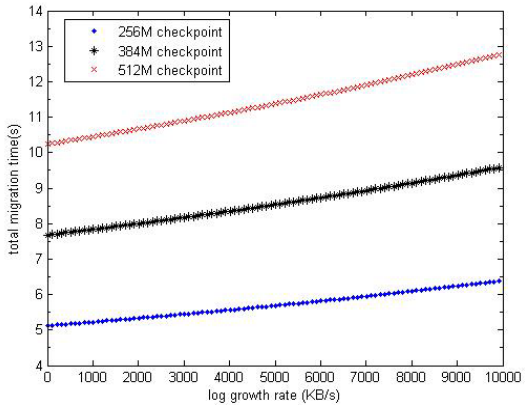

Figure 3. The total migration time varied with log growth rate

The elapsed time during the round n is presented as:

$$t_n = t_0 \varphi^{(n-1)} = \frac{V_{ckpt} \varphi^{(n-1)}}{R_{trans}} \tag{2}$$

Then the *total migration time* (TMT) can be calculated as:

$$TMT = \sum_{i=0}^{n} t_i = \frac{V_{ckpt}(1-\varphi^n)}{R_{trans}(1-\varphi)} \tag{3}$$

With equation (3), the *total data transmitted* (TDT) during a migration becomes:

$$TDT = V_{ckpt} + \sum_{i=1}^{n} V_{\log_i} = TMT * R_{trans} = \frac{V_{ckpt}(1-\varphi^n)}{1-\varphi} \tag{4}$$

Now, we analyze the *downtime* caused in the whole migration process. It is composed of three parts: t_n, the time the last log file (only 1KB or even less) is transferred during the stop-and-copy phase, it is negligible in a high speed LAN; the time the last log file is replayed on the target host; and other time are spent on start-up and service switch overhead. All the three parts can be done within a very short time interval. The total downtime can be represented as:

$$T_{downtime} = t_n + V_{log_n} / R_{replay} + t_{other} \tag{5}$$

To evaluate the convergence rate of our algorithm, we can calculate the total rounds of the iteration by the inequality (6):

$$t_{(n-1)} R_{log} \le V_{thd} \tag{6}$$

It is the condition when the iterative log transferring should be terminated. Combining with equation (1) and (2), inequality (6) can be transformed to $V_{ckpt} \varphi^{(n-1)} \le V_{thd}$, i.e., $n \le 1 + \log_{\varphi} \frac{V_{thd}}{V_{ckpt}}$, so the iteration round is:

$$n = 1 + \left\lceil \log_{\varphi} \frac{V_{thd}}{V_{ckpt}} \right\rceil \tag{7}$$

From the above equations, we can easily make the following conclusions: a smaller size of checkpoint file and faster network

transfer rate would greatly improve the convergence rate of our algorithm, and also reduce the total migration time and the total data transmitted; the log growth rate generates a little effect on the iteration rounds. A simple instance shows that when the network throughput is at a rate of 400Mbit/sec and the checkpoint file is 512MB, the iteration rounds is not more than 5 times even when the log growth rate has risen to 10MB/sec. Figure 3 shows that the total migration time only increases no more than 2.5 seconds when the log growth rate increases from 10KB/sec to 10MB/sec, and transferring the checkpoint file costs the most of migration time (5.1sec, 7.5sec and 10.2sec for 256MB, 384MB and 512MB checkpoint file, respectively). The results indicate that the log growth rate has a little effect on the iteration rounds of log transferring phase, resulting much less network packages transmission and shorter total migration time.

3.3.2 *Synchronization with Waiting-and-Chasing*

Here, all the steps executed are the same as the above scenario, but appends a waiting-and-chasing phase after the log file size is reduced to the threshold value V_{thd}. This synchronization process makes it much more complicated in this case.

When the log file size has reduced to the specified value V_{thd}, there is still much unused log that should be replayed on the target host, so a waiting-and-chasing phase is needed to postpone the stop-and-copy phase until the two VMs get a consistent state. Figure 4 shows the detail process of this scenario. The following inequality presents this condition:

$$\frac{\sum_{i=1}^{m} V_{log_i}}{R_{log}} - \frac{\sum_{i=1}^{m} V_{log_i}}{R_{replay}} <= \frac{V_{ckpt}}{R_{trans}} \tag{8}$$

As $\sum_{i=1}^{m} V_{log_i} / R_{log}$ denotes the time cost when the log file is reduced to the threshold value V_{thd}, it can also be presented with equation (3), as log replay rate can be normalized to log growth rate:

$$R_{replay} = \partial R_{log} \tag{9}$$

where ∂ denotes the ratio of log replay rate to log growth rate. Combining with equation (3) and (9), inequality (8) can be transformed to:

$$(1 - \frac{1}{\partial}) \sum_{i=0}^{m} t_i <= t_0 \tag{10}$$

$$\text{i.e.,} \quad \frac{1-\varphi^m}{1-\varphi} <= \frac{\partial}{\partial - 1} \tag{11}$$

By analyzing the whole process of migration from overall perspective, we can derive that the difference of the two VMs' runtime during the migration is just the time cost to transfer the checkpoint file. This conclusion can be expressed as:

$$\frac{\sum_{i=1}^{n} V_{log_i}}{R_{log}} - \frac{\sum_{i=1}^{n} V_{log_i}}{R_{replay}} = \frac{V_{ckpt}}{R_{trans}} \tag{12}$$

With equation (12) and (9), we can calculate the total data volume of generated log files as:

$$\sum_{i=1}^{n} V_{log_i} = \frac{\partial \varphi}{\partial - 1} V_{ckpt} \tag{13}$$

So the *total data transferred* can be expressed as:

$$TDT = V_{ckpt} + \sum_{i=1}^{n} V_{log_i} = (\frac{\partial \varphi}{\partial - 1} + 1) V_{ckpt} \tag{14}$$

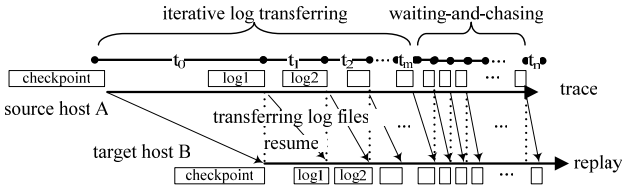

Figure 4. Migration process with a waiting-and-chasing synchronization phase

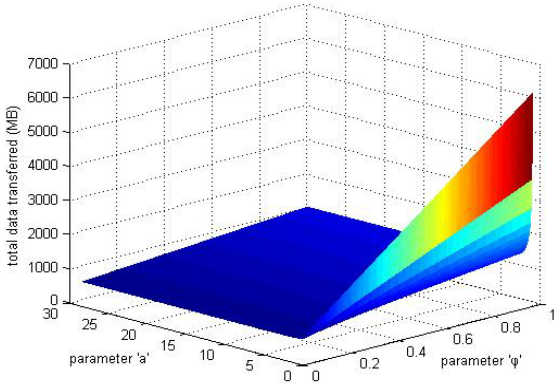

Figure 5. The total data transferred with different parameter values of ∂ and φ during a migration

Figure 5 shows that the total amount of data need to be transferred while migrating a VM with 512MB checkpoint file in a high speed LAN (400Mbit/sec bandwidth).

The *total migration time* can be calculated as:

$$TMT = \frac{V_{ckpt} + V_{log_1}}{R_{trans}} + \frac{\sum_{i=1}^{n} V_{log_i}}{R_{replay}} = (\varphi + \frac{\partial}{\partial - 1}) \frac{V_{ckpt}}{R_{trans}} \tag{15}$$

Note that we should not calculate the *TMT* with equation (3) or simply using the expression TDT/R_{trans}, because during the waiting-and-chasing phase, the log files are not being transferred all the time, but are being replayed on the target host all along.

4. SYSTEM IMPLEMENTATION

To demonstrate the utility of our scheme, we implement our prototype based on acquirable log-and-replay tool – *ReVirt* [1]. The system is implemented as a set of modifications to the host kernel 2.4.20 as ReVirt did. The following describes the details that we apply ReVirt to implement live migration of virtual machine on top of UMLinux for x86 platform.

Figure 6 shows our system structure with logging-and-replay component of ReVirt. We split the logging and replay modules into two parts and locate them on source and target host respectively. The migration daemon on source host is responsible for making transparent checkpoint of the guest OS, reserving required resource for checkpoint and log data transmission, and communicating with the migration receiver on target host. The migration receiver is responsible for reserving necessary resources to recreate the migrated VM, monitoring the log replay rate and orchestrating the migration progress with the source host.

In ReVirt system, log records are added and saved to disk in a manner similar to that used by the Linux *syslogd* daemon. For our migration strategy, we modify the logging module and redirect the

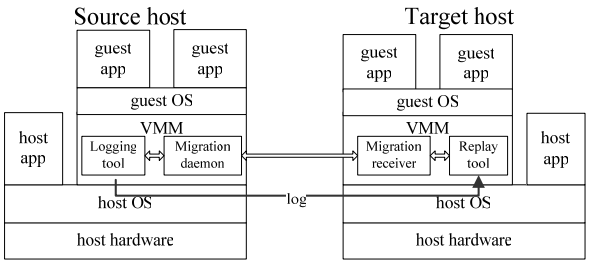

Figure 6. CR/TR-Motion system structure

logging data stream to the network interface card. The VMM kernel module and kernel hooks add log records to a circular buffer in host kernel memory, and a user-level daemon (*rlogd*) consumes the buffer and transmits the data to the target host.

4.1 COW Checkpoint

ReVirt current implementation does not provide runtime checkpointing function. It simply checkpoints the VM state by making a copy of its virtual disk. Most of current checkpoint mechanism should stop the VM for an amount of time linear in the amount of memory configured. This overhead could be mitigated by marking dirty pages as *copy-on-write* (COW) during checkpointing [8]. We use a standard COW mechanism to implement a transparent checkpoint on UMLinux. The checkpoint is performed in the following steps: 1) When a checkpoint request is issued, the VM should be suspended to record the VCPU state at the current instant. Then all the VM memory pages are configured as read-only mode. 2) The VM resumes and continues to run while all the memory pages are replicated to a network transmit buffer and then transmitted across the network. During this process, all the memory writing access would trigger a page fault, and then the pages to be dirtied should be copy to a COW buffer immediately. The replication process could extract any pages marked as copied from the COW buffer instead of reading them directly from the guest OS. When it has finished replicating pages, their space in the buffer could be marked for reuse. This COW mechanism ensures all the memory pages are at a globally consistent state and greatly reduces the checkpointing downtime. 3) When all the memory pages are replicated, all the buffered recorded data are written out to the remote host.

4.2 Local Device Migration

A key challenge of migrating a running VM is how to hold the connections to local device including SCSI disks and network interfaces. There are different solutions to address such issues. The remainder of this section describes the detail issues about migrating the two most important components.

SCSI Devices In a cluster environment, most modern data centers consolidate their storage requirements with *network-attached storage* (NAS) or *storage area networks* (SAN). The NAS can be accessed uniformly from all host machines in the cluster, and this advantage avoids the need to migrate disk storage. Unfortunately, it may cause some conflicts while synchronizing the migrated VM's execution state in our migration approach. For instance, the source VM may first read a block of the disk and then write the same block, if this process is replayed on the target host without any intervention, this may cause some mistakes, because at this time the reading data is what the source VM had written. We should avoid such style of replay that can be named as WAR (write after read) for short. There are two common approaches to address this issue. The first is to track the disk changes in a redo/undo log [11] during the synchronization phase, but it is very difficult to cope with the time sequence of disk access, because the logging and replay are performed synchronously that the two VM may compete to read or write the same block at the same time. Another approach is chosen to address this issue in our system. All the disk read operation on the source VM are intercepted and the bytes are recorded in a log file, during replay on the target host, the disk read are prohibited and redirected to the log file. All the disk writing operations are also prohibited during the replay process because the writing does not change the file state. Although this approach causes some space penalty and network bandwidth consumption, but it works well and does not cause any mistakes. Furthermore, to make the solution much more robust and efficient in a high speed LAN, a little space and network traffic is worthwhile.

Network Connections To ensure the transparency of VM migration, it is essential to guarantee all the network connections that were opened before migration keeping open after the migration finished. In a cluster environment, the network interfaces of the source and target hosts typically attach to the same switched LAN. VMware and Xen address this issue with similar mechanism of ARP broadcasting [11, 3], and we adopt such analogous method to keep ongoing network connections in a LAN.

5. PERFORMANCE EVALUATION

This section investigates the performance characteristics of the VM migration scheme described above. It presents measurements of migration downtime, total migration time and the total data transferred when a VM is migrated in a LAN. With a variety of workloads, our approach shows that VM migration can be fast and transparent to applications and operating systems.

5.1 Experimental Setup

Our experiments are performed on identical hosts with AMD Athlon 3500+ processor and 1GB DDR RAM. Storage is accessed via iSCSI protocol from a NetApp F840 network attached storage server. Each host has an Intel Pro/1000 Gbit/s NIC to transfer the state of the VMs. To reduce the effect on other ongoing network service hosted on the source host, the Linux traffic shaping interface is used to limit network bandwidth to 500Mbit/sec for the migration daemon. The guest OS is RHEL AS3 Linux with kernel 2.4.18 ported to UMLinux, and the host kernel for UMLinux is a modified version of Linux 2.4.20. All the VMs are configured to use 512MB of RAM.

The VM being migrated is the only VM running on the source machine and there are no VMs running on the target machine. In each experiment a single VM is migrated five times between two physical hosts. The results reported are the average of the five trials. The experiments use the following VM workloads:

1) *Daily use*: an idle Linux OS for daily use.

2) *Kernel-build*: the complete Linux 2.4.18 kernel compilation is a system-call intensive workload, which is expensive to virtualization. This is a balanced workload that tests CPU, memory and disk performance.

Table 1. Time and space overhead of logging and replay

Workloads	Log growth rate (KB/sec)	∂ (Replay rate Normalized to logging)
daily use	10.3	36.8
kernel-build	2.2	1.05
static web app	247	1.63
dynamic web app	722	1.24
unixbench	61.4	1.18

3) ***Static web application***: we use the Apache 2.0.63 to measure static content web server performance. Both clients are configured with 100 simultaneous connections and repetitively downloading a 256KB file from the web server.

4) ***Dynamic web application***: a more challenging Apache workload is presented by SPECweb99, a complex application-level benchmark for evaluating web servers and the systems that host them. The workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations, and 0.5% executes a CGI script. A number of client machines are used to generate the load for the server under test, with each machine simulating a collection of users concurrently accessing the web site.

5) ***Unixbench*** [2]: it is a benchmark suite for Linux that integrates CPU, file I/O, process spawning and other workloads. The following tails are performed: Dhrystone2 using register variables, arithmetic, system call overhead, pipe throughput, pipe-based context switching, process creation, execl throughput, file system throughput, concurrent shell scripts, compiler throughput, and recursion.

To compare CR/TR-Motion with previous migration schemes in LAN environments, we port pre-copy algorithm implemented in XenMotion to UMLinux and make the same experiment on the above workloads. The test environment is the same as CR/TR-Motion.

5.2 Logging and Replay Overheads

Our first concern is the log growth rate which represents the space overhead that arises from logging the VM on the host machine. Next we seek to quantify the time overhead of replay on the target host. Table 1 shows the time and space overhead of logging and replay on daily use, kernel-build, static web application, dynamic web application, unixbench workloads. Log growth rate shows the average rate of growth of the log during the workload. Log replay rate is normalized to the log growth rate of the VM with logging. It is denoted by the value of ∂. Workloads with little non-determinism (e.g., kernel-build) generate very little log traffic. The log growth rate for static web application and SPECweb99 is higher because it needs to log the incoming network packets. However, it is still small enough compared to the network transmit rate in a Gbit/s LAN. The log replay rate for kernel-build and unixbench is only a little faster than log growth rate because such applications are compute intensive and generate little non-determinism events.

5.3 Migration Time

We mostly concern about the downtime during which the VM is unavailable. This interval must be short enough to avoid any noticeable delay from the VM.

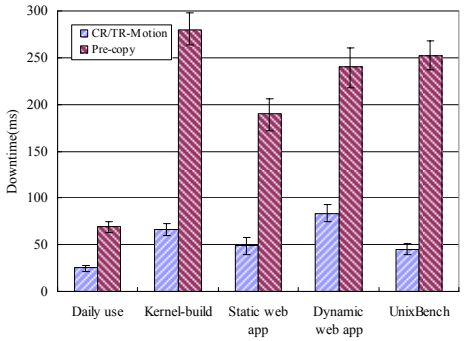

Figure 7. The downtime of CR/TR-Motion and Pre-copy for different workloads

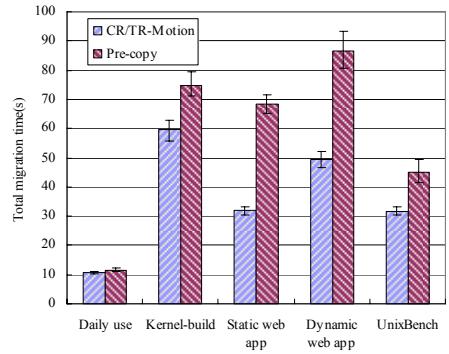

Figure 8. Total migration time of CR/TR-Motion and Pre-copy for different workloads

To compare the migration downtime of our scheme (CR/TR-Motion) with pre-copy scheme, the same workloads are migrated in a high speed LAN. The test result in figure 7 shows that our approach gets much less downtime than pre-copy algorithm. For the above workloads, it reduced the migration downtime by 62.7%, 76.5%, 75.2%, 65.2%, and 82.1% respectively, an average of 72.4%.

We also pay attention to the total migration time during which machine resources are consumed to perform the migration. Figure 8 shows the total migration time is less than one minute for various workloads while migrating in a fast LAN with our scheme, while pre-copy algorithm takes much more time for the same workloads. Our approach reduces the total migration time by 10.1%, 20.4%, 53.6%, 42.9%, 30.3%, an average of 31.5%. This improvement may give a great benefit for cluster administrators. The total migration time for Linux kernel-building seems a little longer than other workloads in our approach. The reason is that the log replay rate is more close to the log growth rate for this workload, so our algorithm executes many rounds of iterations to perform the waiting-and-chasing phase, which cost much longer migration time.

5.4 Network Throughput of Migration

Figure 9 shows our migration approach gets less network throughput than pre-copy algorithm in a LAN for different workloads. As the VM is configured to use 512MB RAM, both CR/TR-Motion and pre-copy approaches should transmit those memory pages to the target host when the VM is migrated. The other data are that should be transferred to synchronize the mi-

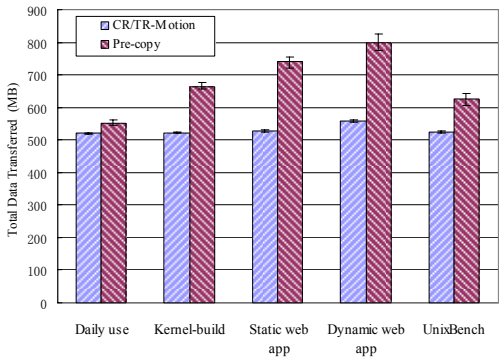

Figure 9. Total data transferred with CR/TR-Motion and Pre-copy for different workloads

Table 2. The data should be transferred to synchronize the migrated VM state for different workloads and migration scheme.

Workloads	Synchronization data volume (MB)		Reduction ratio
	CR/TR-Motion	Pre-copy	
daily use	0.48 (0.04)	38.54 (2.1)	98.8%
kernel-build	0.53 (0.06)	152.44 (8.2)	99.6%
static web app	8.34 (0.21)	228.99 (9.4)	96.4%
dynamic web app	36.4 (0.96)	288.05 (12.2)	87.4%
unixbench	2.59 (0.22)	113.38 (6.4)	97.7%

grated VM state after the memory image has been replicated. In our approach, the total data transmitted is not more than 550MB, most of which are the checkpoint file, and the other data are the negligible logging data. For pre-copy approach, the synchronization data are the dirtied memory pages which are usually considered to be coarse-granularity and bandwidth consumptive. To make a distinctly comparison with pre-copy algorithm, we only show the data need to synchronize the migrated VM state in Table 2. Each result is the mean of 5 trials, with the standard deviation in parentheses. The last list shows our approach drastically reduces the synchronization data compared with pre-copy approach. CR/TR-Motion reduces synchronization traffic by at least 87.4% (dynamic web application) and at most 99.6% (kernel-build), an average of 95.9%. This improvement may provide great benefit if our migration scheme is applied in low-bandwidth *wide area networks* (WAN).

Figure 10 and 11 show the network throughput of migration in fast and slow synchronization scenarios. We can find that the source host achieves a consistent throughput of approximate 400Mbit/sec when the checkpoint file is being transmitted, and then iterative log transferring phase is executed, resulting in the network throughput dropping to only 25Mbit/sec. During the waiting-and-chasing phase, the bandwidth even drops to approximate 4Mbit/sec. Those results demonstrate that the migrations for those workloads cause reasonable network traffic and bandwidth consumption.

5.5 Migration Overhead

Figure 12 illustrates the performance overhead during VM migration. As the selected workloads are macro and may run thousands of seconds at most, while the migration process can be finished in only tens of seconds, we only show the overhead which are

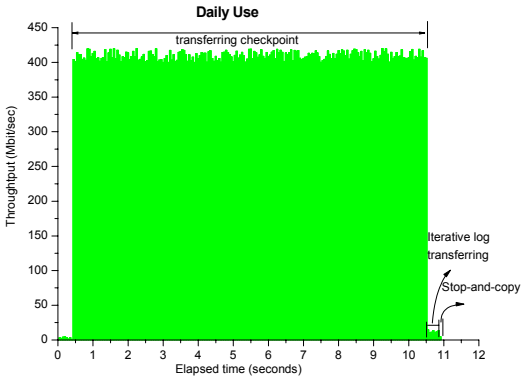

Figure 10. Network throughput of Migrating daily use workload in fast synchronization scenario

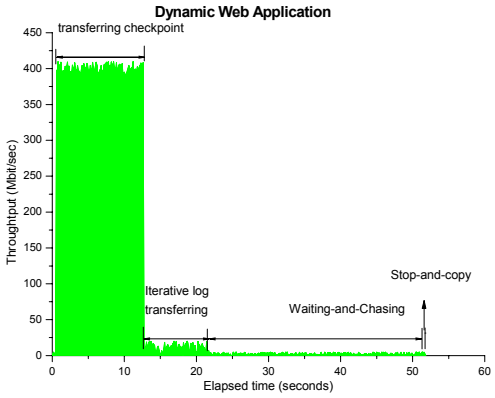

Figure 11. Network throughput of Migrating dynamic web application in slow synchronization scenario

caused by transparent checkpointing and logging during the total migration time. For all cases the overhead is low (less than 8.54% on average).

To monitor the effect of resource reserving during VM migration, the source physical machine is loaded with 5 CPU-bound virtual machines, and the time to migrate the 512MB Linux VM is measured under different resource reservations. Figure 13 shows that reserving about 30% of a CPU for migration minimizes the total migration time. This implies that it takes about 30% of a CPU to attain the maximum network throughput over the gigabit link. We also discover that even though the total migration time increased when insufficient CPU is reserved for the migration, the migration downtime remains small regardless of the amount of reserved CPU. Because it only requires little CPU time to stop the VM and transfer the remainder state during the stop-and-copy phase.

6. RELATED WORK

Recent virtual machine management tools allow live migration of servers within a local area network environment. These technologies have proven to be a very effective tool to enable data center management in a non-disruptive fashion. Pre-copying algorithm is widely used for VM live migration in a memory-to-memory approach [7, 17]. During migration, physical memory pages are pushed across network to the new destination while the source host continues running. Pages modified during the replication

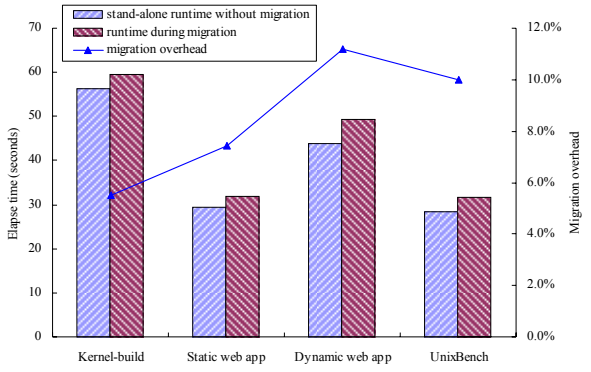

Figure 12. Migration overhead for different workloads

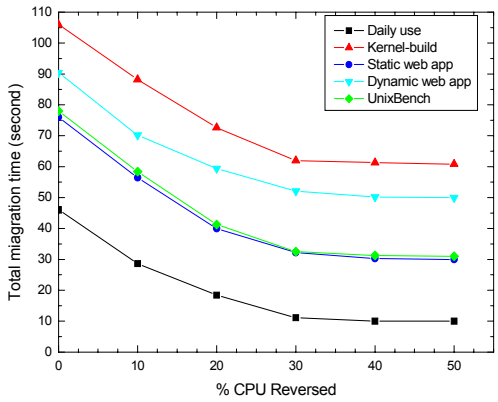

Figure 13. Effect of CPU reservation on migration from a heavily loaded source physical machine

must be re-sent to ensure consistency. After a bounded iterative push phase, a very short stop-and-copy phase is executed to transfer the remaining dirty pages. This COW (*copy-on-write*) mechanism achieves very short best-case migration downtimes, but for memory intensive workloads, the downtime may evidently increase to several seconds. To greatly reduce memory replication during migration, a high performance VM migration scheme is proposed [10] by using RDMA (*Remote Direct Memory Access*), which is a feature provided by many modern high speed interconnects (InfiniBand). Their approaches drastically reduce the VM migration overheads.

In a LAN environment, since the migrated virtual machine retains the same network address as before, any ongoing network level interactions are not disrupted. Similarly, storage requirements are normally met via either *network attached storage* (NAS) or *storage area network* (SAN), which is still reachable from the migrated VM location to allow continuous storage access. Unfortunately, in a WAN environment, live VM migration is not as easily achievable for intractable network connectivity holding and bulk storage replication. Current virtual machine software with a suspend and resume feature can be used to support WAN migration includes Collective [21], Internet Suspend/Resume [13] and μDenali [24], but those projects have explored migration over longer time spans by stopping the source VM and then transferring the VM image file and local block devices to the target host. To archive live migration of virtual machine across WANs, recent approaches using dynDNS [25] and IP tunnels to guarantee network connections is demonstrated [6, 23], where an IP tunnel between the source and target host is set up to transparently forward packets to and from the client applications. But the edge router's support is required for those approaches. To replicate the large size of local disk storage with less disruption, a block level solution combining pre-coping with write throttling is described [6].

To optimize the transfer of large amounts of disk and memory state during migration, a solution based on opportunistic replay is proposed [22], which captures user interactions with applications at the GUI level, resulting in very small replay logs that economize network utilization. This approach is somewhat similar with our mechanism, but its applicable scenario requires that the target hosts should have initially identical replica of a suspended VM with source host before the source VM is migrated to the same host again. Another difference with our approach is that it implements logging-and-replay only at the GUI level but not full system, thus results in divergent VM state that should be dealt with cryptographic hashing techniques. Moreover, their VM migration scheme is not implemented in a live fashion.

Transparent and efficient access to I/O devices during VM migration needs dynamically change the mappings of virtual to physical devices. Netchannel [14] presents a VMM-level abstraction that transparently handles pending I/O transactions, thus provides a novel mechanism for continuous and seamless device access during VM migration and device hot-swapping for networked as well as locally attached devices. VTL [15] is a framework for packet modification and creation. Its purpose is to modify network traffic to and from a VM transparently so as to enable connection persistence during long duration VM migration and hibernation.

7. CONCLUSION

In this paper we present the design, implementation, and evaluation of a novel approach for live VM migration. It shows how we adopt checkpointing/recovery and trace/replay technology to provide fast, transparent VM migration. This approach makes unperceived VM migration downtime and reasonable network bandwidth consumption. Experimental measurements show our scheme get better average performance compared with pre-copy approaches: up to 72.4% on application observed downtime, up to 31.5% on total migration time and up to 95.9% on the data to synchronize the VM state, while the application performance overhead due to migration is less than 8.54% on average.

However, in multi-processor (or multi-core) environment, as expensive memory race among different VCPUs must be recorded and replayed, this make an inherent difficult for our approach to migrate SMP guest OS. VCPU hot plug technique may address this issue by dynamically configuring the migrated VM to use only one VCPU before migration, and give back the VCPUs after the migration is finished. Although this makes some performance degradation during migration for a short time, it is desirable compared with the overheads of logging and replaying multiprocessor VMs. We will further study this issue and make a tradeoff between pre-copy and our approach for multiprocessor VMs migration.

8. ACKNOWLEDGMENTS

This work is supported by National 973 Basic Research Program of China under grant No.2007CB310900. We are grateful to the researchers of ReVirt group for sharing their work with us.

9. REFERENCES

[1] http://www.eecs.umich.edu/virtual/software.html

[2] http://www.tux.org/pub/tux/benchmarks/system/unixbench/

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP'03)*, October 19-22, 2003, Lake George, New York, USA, pp.164-177

[4] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent Advances in Checkpoint/Recovery Systems. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, April 25-29, 2006

[5] K. Buchacker and V. Sieh. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects, In *Proceedings of 6th IEEE International High Assurance Systems Engineering Symposium (HASE'01)*, October 22-24, 2001, pp.95-105

[6] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioeberg. Live Wide-Area Migration of Virtual Machines Including Local Persistent State. In *Proceedings of the third International Conference on Virtual Execution Environments (VEE'07)*, ACM Press, June 13-15, 2007, San Diego, California, USA, pp.169-179

[7] 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 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, May 2-4, 2005, Boston, MA, USA, pp.273-286

[8] B. Cully, G. Lefebvre, D. T. Meyer, A. Karollil, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of 5th Symposium on Networked Systems Design and Implementation (NSDI'08)*, April 16-18, 2008, San Francisco, CA, USA

[9] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, ACM Press, December 8-11, 2002, Boston, MA, USA, pp.211-224

[10] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *Proceedings of IEEE International Conference on Cluster Computing (Cluster'07)*, September 17-20, 2007, Austin, Texas, USA

[11] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*, April 10-15, 2005, Anaheim, CA, USA

[12] S. T. King. Operating System Extensions to Support Host-Based Virtual Machines. *Technical Report CSE-TR-465-02*, University of Michigan, September 2002

[13] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (HotMobile'02)*, June 20-21, 2002, Callicoon, NY, USA, p.40

[14] S. Kumar and K. Schwan. Netchannel: A VMM-level Mechanism for Continuous, Transparent Device Access During VM Migration. In *Proceedings of the 2008 International Conference on Virtual Execution Environments (VEE'08)*, March 5-7, 2008, Seattle, WA, USA, pp.31-40

[15] J. R. Lange and P. A. Dinda. Transparent Network Services via a Virtual Traffic Layer for Virtual Machines. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC'07)*, June 27-29, 2007, Monterey Bay, California, USA

[16] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *Proceedings of 21st ACM International Conference on Supercomputing (ICS'07)*, June 16-20, 2007, Seattle, WA, USA, pp.23-32

[17] M. Nelson, B. H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of USENIX Annual Technical Conference (USENIX'05)*, April 10-15, 2005, Marriott Anaheim, Anaheim, CA, USA, pp.391-394

[18] R. Nathuji and K. Schwan. Virtual Power: Coordinated Power Management in Virtualized Enterprise Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'07)*, October 14-17, 2007, Skamania Lodge Stevenson, WA

[19] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. ExecRecorder: VM-Based Full-System Replay for Attack Analysis and System Recovery. In *Proceedings of The 9th Asian Symposium on Information Display (ASID'06)*, October 21, 2006, San Jose, California, USA, pp.66-71

[20] C. Perkins, IP Encapsulation within IP, *RFC 2003*, 1996

[21] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 8-11, 2002, Boston, MA, USA

[22] A. Surie, H. A. Lagar-Cavilla, E. de Lara, and M. Satyanarayanan. Low-Bandwidth VM Migration via Opportunistic Replay. In *Proceedings of the Ninth Workshop on Mobile Computing Systems and Applications (HotMobile'08)*, February 25-26, 2008, Napa Valley, CA, USA

[23] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Wang, Seamless Live Migration of Virtual Machines Over the MAN/WAN, *Future Generations Computer Systems*, Vol.22, No.8, October 2006

[24] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing Services with Interposable Virtual Hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI'04)*, March 29-31, 2004, San Francisco, USA, pp.169-182

[25] B. Wellington. Secure DNS Dynamic Update, *RFC 3007*

[26] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation*, June 10, 2007, California, USA

[27] M. Zhao and R. J. Figueiredo. Experimental Study of Virtual Machine Migration in Support of Reservation of Cluster Resources. In *Proceedings of the 2nd International Workshop on Virtualization Technologies in Distributed Computing (VTDC'07)*, November 12, 2007, Reno, NV, USA