Pacer: Taking the Guesswork Out of Live Migrations in Hybrid Cloud Computing

Jie Zheng† † T. S. Eugene Ng† † Kunwadee Sripanidkulchai∗ Zhaolei Liu‡
Rice University† † † NECTEC, Thailand∗

Abstract

Hybrid cloud computing, where private and public cloud resources are combined and applications can migrate freely, ushers in unprecedented flexibility for businesses. To unleash the benefits, commercial products already enable the live migration of full virtual machines (CPU, memory, disk, network) between distant cloud datacenters. Unfortunately, no live migration progress management system exists, leading to (1) guesswork over how long a migration might take and the inability to schedule dependent tasks accordingly; (2) unacceptable application degradations – application components could become split over distant cloud datacenters for an arbitrary period during migration; (3) inability to balance application performance and migration time – e.g. to finish migration later for less performance interference.

Pacer is the first migration progress management system. Pacer’s techniques are based on robust and lightweight run-time measurements of system and workload characteristics, novel efficient and accurate analytic models for progress predictions, and online adaptation to maintain user-defined migration objectives for coordinated and timely migrations.

1 Introduction

Hybrid cloud computing (HCC) – where virtualizable compute and storage resources from private datacenters and public cloud providers are seamlessly integrated into one platform in which applications can migrate freely – is emerging as the most preferred cloud computing paradigm for commercial enterprises according to recent industry reports and surveys [3, 5, 17]. This is not surprising since HCC combines the benefits of public and private cloud computing, resulting in unprecedented flexibility for CAPEX and OPEX savings, application adaptivity, disaster survivability, zero-downtime maintenance, and privacy control. An impressive array of commercial products that facilitates HCC is already available today; Table 1 highlights some examples.

Figure 1 illustrates some beneficial usage scenarios of HCC. The ability to migrate applications freely among private and -public clouds, whether it is private-to-public, public-to-private, or -public-to-public, is a key to maximizing the benefits of HCC. Tried and true live virtual machine (VM) migration technologies are therefore central to the HCC paradigm. The migration in HCC is inter-datacenters, and consequently it typically requires the full migration of the VM, including virtual CPU, memory, disk storage, and network connection. Full migration is substantially more complex than the CPU/memory-only migration, which is commonly conducted in server cluster load-balancing scenarios. Fortunately, commercial products (Table 1) already enabled the full and live migration of VMs between distant cloud datacenters. In particular, F5 BigIP [11] leverages hypervisor support for live CPU/memory and storage migration, and automates the full migration of VMs between datacenters.

That is, however, not to say live migration in HCC is a solved problem. On the contrary, the use of live migration in HCC has exposed a fundamental weakness in existing solutions, namely the lack of live migration progress management – the ability to predict migration time, to coordinate the progress of multiple migrations, and to control the migration time.
1.1 Lack of migration progress management

In traditional datacenters, CPU/memory-only migration is the most common migration use case. Since the migration is often conducted over a fast LAN and there is relatively little data to migrate, the migration typically completes in a few seconds, and there is very little need for migration progress management. In comparison, migration in HCC is over a much slower WAN and the volume of data to migrate is several orders of magnitude larger because of the virtual disk migration, and thus the migration in HCC thus could take thousands of seconds. In general, such a large migration time could happen whenever a very large amount of data needs to be migrated, even if the migration is within a datacenter LAN. At such a timescale, the migration progress management (MPM) is sorely missed, and the users resort to guessing on a variety of issues:

**How long does migration take?** – is a popular question in live VM migration FAQs [13, 16, 15]. Unfortunately, there is no simple formula for calculating the answer, because the finish time depends on many dynamic run-time variables that are not known before migration a priori. Those variables includes application I/O workload intensity, network speed, and disk speed. As numerous online forum discussions indicate (e.g. [21, 22, 23, 20, 27, 29, 30, 28]), users routinely try to guess why migration is slow and whether it could be sped up, and how long they might have to wait.

**How to avoid application components getting split between distant datacenters during migration?** – This issue is of paramount importance to enterprises, because their applications often consist of multiple interacting components that performs different functions (e.g. content generation, custom logic, data management, etc. [8]). Without the ability to manage migration progress, individual application components could finish the migration at very different time and become split over distant cloud datacenters for an arbitrarily long period. The resulting large inter-component communication delay guarantees detrimental performance impact. Intuitively, a stopgap method is to set the data migration speeds for different components to be proportional to their virtual memory/storage sizes. Unfortunately, the migration time depends on a large number of dynamic run-time variables, and this stopgap method is bound to fail.

**How to control the trade-off between application performance and migration time?** – is another popular question raised by users [25, 24]. Studies [26, 4, 2, 18] have shown that migration can degrade application performance, and slowing migration down helps [4]. Although the administrator might be willing to slow down the migration to some extent to improve application performance, a migration task must still be finished before a deadline or other dependent tasks cannot proceed. Unfortunately, no solution exists for managing migration progress to finish at a desired time. A stopgap method is to configure the data migration speed to be proportional to the data size.

Again, this method is bound to fail due to the dynamic run-time variables (see Section 3).

1.2 Pacer

The contribution of this paper is Pacer – the first migration progress management system (MPMS). Pacer effectively addresses all three of the aforementioned issues. While much details of Pacer’s constituent techniques will be discussed in Section 2. Those techniques share the following key strengths:

**Robust and lightweight run-time measurements** – Pacer is highly effective because it continuously measures the application I/O workload intensity (both memory and disk accesses) and the bottleneck migration speed (network or disk) at run-time. Furthermore, the implementation of these techniques minimizes the run-time overhead as shown in Section 3.

**Novel efficient & accurate analytic models for predictions** – Such analytic models are used for (1) predicting the amount of remaining data to be migrated as a function of the application’s I/O workload characteristics and the migration progress, and (2) predicting the finish time of the migration (i.e. addressing the first MPM issue) as a function of the characteristics of each migration stage (i.e. disk, dirty blocks, CPU/memory, etc.).

**Online adaptation** – Addressing the second and third MPM issues requires certain migration objectives: the former requires multiple application components to complete migrations simultaneously; the latter requires a migration to complete at a specific time spot. No static migration settings can successfully meet such objectives due to run-time dynamics. Pacer continuously adapts to ensure that the objectives are met. In the former case, Pacer adapts the targeted migration finish time for all components given what is measured to be feasible. In the latter case, Pacer adapts the migration speed to main-

---

<table>
<thead>
<tr>
<th>Product category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seamless private and public cloud networking</td>
<td>Amazon VPC, Rackspace RackConnect, Microsoft Windows Azure Virtual Network</td>
</tr>
<tr>
<td>Unified hybrid cloud management console</td>
<td>RedHat CloudForms, RightScale myCloud, VMware vCloud Connector</td>
</tr>
<tr>
<td>Datacenter-to-datacenter full VM live migration</td>
<td>F5 BigIP, VMware VM/Storage vMotion, KVM VM/block migration, Microsoft Hyper-V VM/storage migration</td>
</tr>
</tbody>
</table>

Table 1: Examples of commercial products that help make hybrid cloud computing a reality.
tain a targeted migration finish time in face of application dynamics.

1.3 Road map

The rest of this paper is organized as follows. Section 2 presents the constituent techniques in Pacer for migration progress management. Section 3 and 4 present experimental results demonstrating the capability and benefit of Pacer. Section 5 introduces how to apply Pacer in the concurrent migration scenario. We discuss related work in Section 6 and conclude in Section 7.

2 Design of Pacer

2.1 Overview

Pacer is designed for the pre-copy migration model, which is widely used in virtualization platforms KVM [9] and XEN [6]. A slightly different variant found in VMware [12, 1] will be discussed later. In the pre-copy model, which is implemented by KVM, the virtual disk migration is prior to the memory migration. During the virtual disk copy, all write operations to the disk are logged. The dirty blocks are retransmitted, and the new dirty blocks generated during that time period are again logged and retransmitted. The dirty block retransmission process repeats until the number of dirty blocks falls below a threshold, and then the memory migration begins.

As far as we know, there is no existing solution to quantify the migration time for each stage, especially for the dirty iteration. The problem is challenging for two reasons. Firstly, the total number of dirty pages/blocks is not known before migration completes, since that number depends on how the application accesses the memory and the storage (i.e., the application workload). Secondly, the migration speed is unknown due to the interference brought by the migrated VM’s workload and other competing workloads, which share the network or disk with migration.

Pacer has two main functions: predicting migration time and best-effort time control.

2.2 Predicting migration time

Pacer performs prediction periodically (default configuration is every 5 seconds). To predict the remaining time during the migration, three things must be known: (1) what operation is performed in each stage of migration, (2) how much data there is to migrate, (3) how fast is the migration progressing. The paper will address these three topics in the following sections.

2.2.1 Migration Time Model

The total migration time $T$ can be modeled as four distinct parts: $P_{pre-copy}$ for the pre-copy stage, $I_{dirty-iteration}$ for the period after pre-copy but before memory migration, $M_{memory}$ for the period from the beginning of the memory migration until the time the VM is suspended, and $D_{downtime}$ for a small downtime needed to copy the remaining dirty blocks and dirty pages once they drop below a configured threshold:

$$T = P_{pre-copy} + I_{dirty-iteration} + M_{memory} + D_{downtime}$$

In all formulas, variables in upper case either have given values or their values are provided by algorithms, which the paper will discuss later in Section 2.2.2. The variables in lower case are to be computed.

For the pre-copy phase, we have:

$$P_{pre-copy} = \frac{DISK\_SIZE}{speed_{pre-copy}}$$

where $DISK\_SIZE$ is the VM virtual disk size obtained directly from the VM configuration and $speed_{pre-copy}$ is the migration speed for the pre-copy stage.

At the end of pre-copy, a set of dirty blocks need to be migrated. The amount is defined as $DIRTY\_SET\_SIZE$. This variable is crucial to the prediction accuracy during the dirty iteration. However, the exact value is unknown until the end of pre-copy phase. It is very challenging to know the dirty set size ahead-of-time while the migration is still in the pre-copy stage. The algorithm in Section 2.2.2 is the first to solve this problem.

In the dirty iteration, while dirty blocks are migrated and marked clean, the clean blocks may be overwritten concurrently and get dirty again. The number of blocks getting dirty per second is called the dirty rate. The dirty rate depends on the number of clean blocks (fewer clean blocks means fewer blocks can become dirty later) and the workload of the VM. Similar to the need for dirty set size prediction, we need an algorithm to predict the dirty rate ($AVE\_DIRTY\_RATE$) while migration is still in pre-copy. The time for dirty iteration is

$$I_{dirty-iteration} = \frac{DIRTY\_SET\_SIZE}{speed_{dirty-iteration} - AVE\_DIRTY\_RATE}$$

where $speed_{dirty-iteration}$ is the migration speed for the dirty iteration stage.

Memory migration typically behaves similarly to the storage migration dirty iteration. All memory pages are first marked dirty, then dirty pages are iteratively migrated and marked clean. Pages can become dirty again after being written. We propose an algorithm in Section 2.2.2 that is effective for predicting the average memory dirty rate ($AVE\_MEM\_DIRTY\_RATE$).

During the memory migration, different systems have different behaviors. For KVM, the VM still accesses storage in the source and disk blocks could get dirty during memory migration. Thus, in KVM, memory migration and storage dirty iteration may alternatively happen. Then, denoting the size of memory as $MEM\_SIZE$ and memory migration speed as $speed_{memory}$, we have
\[ I_{\text{Memory}} = \frac{\text{MEM_SIZE}}{\text{speed_{Memory} - AVE\_MEM\_DIRTY\_RATE} - AVE\_DIRTY\_RATE} \] (4)

**Other variants:** The previous derivation is based on a fair assumption that memory migration follows storage migration (KVM and XEN). The model can easily be adapted to other systems, e.g., VMware. Storage migration and memory migration are two separate tasks. At the end of the storage dirty iteration, the VM is suspended and the remaining dirty blocks are copied to destination. Subsequently, storage I/O requests go to the destination, and thus no more dirty blocks will be generated, while the memory and the CPU of the VM are still at the source, so the storage I/O accesses remains remote until the memory migration completes. The speed for memory migration in VMware would be lower than that in KVM, because the network bandwidth is shared between the migration and remote I/O requests. Therefore, for VMware, Equation 4 will be adjusted as follows:

\[ I_{\text{Memory}} = \frac{\text{MEM_SIZE}}{\text{speed_{Memory} - AVE\_MEM\_DIRTY\_RATE}} \] (5)

The above migration time model describes how the time is spent in each phase of live migration. The next question to address is the amount of data to migrate.

### 2.2.2 Dirty set and dirty rate prediction

Migrated data consists of two parts. The first part is the original disk and memory (\(\text{DISK\_SIZE}\) in Equation 2 and \(\text{MEM\_SIZE}\) in Equation 4), the size of which is known. The second part is the generated dirty blocks and dirty pages during migration (\(\text{DIRTY\_SET}\) in Equation 3), the size of which is unknown. We now present algorithms for predicting that unknown variable.

**Disk dirty set prediction:** We divide the VM disk into a sequence of small blocks (e.g., 1MB per block). For each block, we record the average write interval, the variance of write interval (used in dirty rate prediction), and the last written time. When a write operation is issued, Pacer updates the record for blocks accessed by the operation.

Dirty set consists of three types of blocks. \(SET_1\) is the migrated blocks, which are already dirty. This set can be computed by taking the intersection of already-migrated blocks and dirty blocks. \(SET_2\) is the migrated blocks which are clean right now, but they are predicted to be dirty before the end of pre-copy. The pre-copy finish time is predicted by the current progress, and the block write access pattern is predicted by the current access sequence. If a block will be written before pre-copy finishes, it should be included in this set. \(SET_3\) is the not-yet-migrated blocks, which are predicted to get dirty after its migration finish time and before the end of pre-copy. The fact that whether a block should be in this set is predicted by the possible migratable time of that block and its possible written time.

![Figure 2: An example of disk dirty set prediction.](image)
dirty blocks that are assumed to be cleaned immediately during this first interval is \( D_1 = \sum_{\forall \text{block} \in \text{SET}_{\text{Clean}} \cup \{ \text{dblock}_1 \}} \text{write}_{\text{interval}}(\text{block}) \). Note that \( \text{dblock}_1 \) is included because it becomes clean. In general, the expected number of new generated dirty blocks during the \( k \)-th interval is \( D_k = \sum_{\forall \text{block} \in \text{SET}_{\text{Clean}} \cup \{ \text{dblock}_1, \text{dblock}_2, \ldots, \text{dblock}_k \}} \text{write}_{\text{interval}}(\text{block}) \). Thus, the average dirty rate can be computed as follows:

\[
\text{AVG DIRTY RATE} = \frac{\sum_{i=1}^{N} D_i \cdot \text{blocksize}}{\sum_{i=1}^{n} \text{write}_{\text{interval}}(\text{block})} + \frac{\sum_{i=1}^{N} (N+1-k) \cdot \text{blocksize}}{\sum_{i=1}^{N} \text{write}_{\text{interval}}(\text{dblock}_k)}
\]

Furthermore, to remove inactive blocks from dirty rate calculation, we add the following mechanism. For simplicity, assume the write intervals of a block follow a normal distribution \([14] \sim N(\mu, \sigma)\). The possibility that the next arrival time is in \([\mu-2\sigma, \mu+2\sigma]\) is 95\%. Therefore, if the time since the last write is already longer than \(2\sigma\) for a block, that block can be safely considered inactive. The average write interval for such a block is set to infinity. This mechanism significantly improves the accuracy of dirty rate prediction.

**Memory dirty rate prediction:** The disk dirty rate prediction algorithm would incur high tracking overhead if it is applied to the memory dirty rate prediction. Therefore, we propose a sampling-based algorithm to trade precision for reduced overhead. The idea is that Pacer periodically takes a snapshot of the dirty bitmap of memory pages, resets the dirty bitmap, and updates two types of information. The first is a cumulative write access counter for each page. If a page is written to during this period, this counter is incremented. The second is the number of unique written pages \( u \) during this period obtained by counting the number of set bits. With these information, we can predict the average dirty rate as follows. We define the access ratio for each page \( i \) as

\[
\text{access\_ratio}(i) = \frac{\text{write\_counter}(i)}{\sum_{\forall \text{counter}(i) \in \text{all pages}}}.
\]

Denote the sampling interval to be \( t_s \), and then the rate that unique write pages are generated is \( \frac{u}{t_s} \). The rate is an upper bound for the true dirty page rate, and it corresponds to the worst case scenario, where all pages were clean at the beginning of the interval. With access ratio representing the contribution of a page to the overall dirty rate, the dirty rate for page \( i \) can be predicted as \( d(i) = \frac{u}{t_s} \cdot \text{access\_ratio}(i) \). Similar to the analysis for the disk dirty iteration, when migrating the \( n \)-th page, the dirty rate is \( \sum_{i=1}^{n} d(i) \). The average dirty rate is therefore

\[
\frac{\sum_{i=1}^{N} \sum_{k=1}^{N} d(i)}{N} \quad \text{where} \quad N \text{ is the total number of memory pages.}
\]

The selected sampling interval would affect the accuracy of the prediction. For example, if we sample at 2s and there is a page written every one second, its predicted dirty rate will be lower than the real dirty rate. A way to increase the accuracy is to reduce the sampling interval in consecutive rounds and see whether the predicted dirty rate increases. If the dirty rate increases, the sampling interval will be reduced further until the rate stabilizes or the interval meets a configured minimal interval. In Pacer, the sampling interval starts at 2s and is reduced by half if needed. To bound the overhead, we set a minimum sampling interval to 0.25s.

### 2.2.3 Speed measurement

In KVM, a static *configured migration speed* is used to decide how fast the migration system will copy and transmit migrated data. However, the **actual migration speed** may be smaller than the configured speed due to interference. It is therefore important to measure and use the actual speed for migration time prediction in Equations (2) (3) (4).

**Smoothing measurements:** In each interval, we measure the migrated data and compute the average actual speed during the interval. In order to smooth out short time scale variations on the measured actual speed, we apply the commonly used exponential smoothing average method to update the measured actual speed. The smoothing weight \( \alpha \) is set to 0.8 in the implementation.

\[
speed_{\text{smooth}} = \alpha \cdot speed_{\text{smooth}} + (1 - \alpha) \cdot speed_{\text{measured}}.
\]

### 2.3 Controlling migration time

Pacer divides the migration time into rounds of small intervals. In each round, Pacer adapts the configured migration speed to maintain a target migration finish time. It updates the prediction of dirty block set, dirty disk rate, and dirty memory rate based on the algorithms in Section 2.2.2, and then Pacer computes proper configured migration speed in a way as the following section describes. The speed is adjusted later based on the algorithms that handle I/O interference in Section 2.3.2.

#### 2.3.1 Solving for speeds in each phase of migration

For a specific desired migration time \( T \), many combinations of migration speeds in each phase are feasible. Pacer aims to control the migration progress in a systematic and stable way, which leads to the following speed solutions.

Migrating memory pages generally will not generate disk I/O, because for performance consideration the memory of the VM is usually mapped to the memory of the physical machine, and thus the speed of memory migration is limited by the available network bandwidth (\( \text{NETWORK\_SPEED} \) which can be directly measured) and so

\[
speed_{\text{Memory}} = \text{NETWORK\_SPEED}
\]
With above simplification, only two variables need to be solved: \( \text{speed}_{\text{precopy}} \) and \( \text{speed}_{\text{DirtyIteration}} \). There are still many combinations of such speeds that can finish migration in time \( T \). Migration generates I/O reads which interfere with the I/O workload of the VM during the storage migration. To minimize the severity of disk I/O interference caused by migration, we seek to minimize the maximum migration speed used. This policy implies that

\[
\text{speed}_{\text{precopy}} = \text{speed}_{\text{DirtyIteration}}
\]  

(8)

where \( \text{speed}_{\text{DirtyIteration}} \) is the average speed for the dirty iteration during storage migration. Thus, the appropriate \( \text{speed}_{\text{precopy}} \) can finally be solved by substituting and rearranging terms in Eq. (1).

More precisely, during the pre-copy stage, at the beginning of each interval, we solve the following equations to obtain the migration speed (\( \text{speed}_{\text{precopy}} \) or \( s_1 \) for short) to use for the interval.

\[
\text{Speed}_{\text{Precopy}} = \frac{\text{remaining}_{\text{precopy}} \cdot \text{speed}_{\text{DirtyIteration}}}{t_3}
\]

where \( t_3 \) is the remaining migration time before deadline

\[
s_1 = \frac{\text{remaining}_{\text{precopy}}}{t_1}
\]

\[
dirty_{\text{set}} \cdot \text{size} + \text{dirtyrate}_{\text{disk}} \cdot \text{t}_{2} = s_2 \cdot t_2
\]

\[
s_1 = s_2
\]

\[
s_1, s_2 \geq 0
\]

\[
0 \leq t_1, t_2 \leq \text{remaining}_{\text{time}} - T_{\text{Downtime}} - t_3
\]

(9)

During the dirty iteration, we have the total bytes of current dirty blocks \( \text{dirty}_{\text{set}} \cdot \text{size} \). The migration speed consists of two parts. One part is to migrate the current dirty blocks in the remaining time before memory migration. The other part is to migrate newly generated dirty blocks at the rate of \( \text{dirtyrate}_{\text{disk}} \).

\[
\text{speed}_{\text{DirtyIteration}} = \frac{\text{dirty}_{\text{set}} \cdot \text{size} + \text{dirtyrate}_{\text{disk}}}{t_3}
\]

(10)

During memory migration, the migration speed is set to the available network bandwidth.

We apply an algorithm which will be described in Section 2.3.2 for computing the maximal actual migration speed that can be realized under interference. When Pacer detects that the computed speed is higher than the maximal actual speed, it knows finishing by the desired time is not feasible. Then it employs disk I/O throttling to upper bound the disk dirty rate to a configurable fraction of the achievable migration speed.

Figure 3 illustrates how the migration speed might be controlled by Pacer during different migration stages.

![Diagram](image_url)

Figure 3: An example of migration speeds in different stages.

During pre-copy, Pacer aims to maintain a stable speed but adapts to workload changes if necessary. During dirty iteration, the migration speed depends on the dirty set size and the dirty rate. At the beginning of dirty iteration, the dirty set already includes the most frequently written blocks, so few new blocks will get dirty, corresponding to a low dirty rate. As more dirty blocks become clean, the dirty rate increases. The shape of the curve in practice depends on the workload. Pacer aims to migrate the dirty set at a stable pace, and thus the dirty iteration migration speed curve is parallel to the dirty rate curve. Finally, during memory migration, migration can typically proceed at a higher speed than the speed in the previous two stages because the bottleneck is most likely in the network.

**Other variants:** Similar to the discussion in Section 2.2.1 for migration time model, speed control can readily be adapted to other hypervisor. As an example, for VMware, Equations 7 will be adjusted as follows:

\[
\text{speed}_{\text{memory}} = \text{NETWORK \_SPEED} - \text{I/O \_RATE}
\]

(10)

where \( \text{I/O \_RATE} \) denotes the bandwidth consumed by remote storage I/O and can be predicted by monitoring the application workload.

2.3.2 Maximal speed prediction and tuning

Due to interference from disk or network, the achieved actual migration speed may vary. Therefore, it is important to predict the true maximal actual migration speed and ensure that the configured migration speed is realized.

We predict the maximal actual speed by comparing the configured speeds as specified by Pacer and the measured actual speeds in reality. When the migration starts, if we detect that the measured actual speed cannot reach the configured speed, we will record this speed values pair. In subsequent rounds, if the new measured actual speed is lower than or equal to the previous recorded actual speed, and the new configured speed is higher than
previous recorded configured speed, we predict that the maximal actual speed has been reached. The maximal actual speed is updated by the current actual speed. In the future, when any measured actual speed is higher than the maximal actual speed, the maximal actual speed is updated. In order to smooth out short time scale variations on the maximal actual speed, we use an exponential smoothing average for updating the maximal actual speed. The smoothing weight $\beta$ in Pacer is set to 0.8.

When the measured actual speed cannot reach the configured speed in a round, Pacer will scale up the configured speed for the next round and set a scale-up flag to indicate that the speed has been scaled up. In the next round, if the new measured configured speed is not higher than the previous measured actual speed, that means the scaling up did not help. Pacer then does not perform scale up for the next round.

3 Evaluation

3.1 Implementation

Pacer is implemented on the kernel-based virtual machine (KVM) platform. KVM consists of a loadable kernel module, a processor specific module and a user-space program – a modified QEMU emulator. QEMU performs management tasks for the VM. Pacer is implemented on QEMU version 0.12.50. We add two options in the migration command to enable prediction and to specify desired migration time with about 1500 lines of code.

3.2 Experiment Setup

The experiments are set up on two physical machines. Each machine has a 3GHz Quad-core AMD Phenome II X4 945 processor, 8GB DRAM, 640GB WD Caviar Black SATA hard drive, and Ubuntu 9.10 with Linux kernel (with the KVM module) version 2.6.31. In all experiments unless otherwise specified, the migration speed is restricted to no more than 32MBps to mimic the level of available bandwidth in inter-datacenters scenarios.

In our test platform, the I/O write speed on the destination disk for migration is at most 15MBps, while RAID is widely used in commercial clouds to increase the I/O speed to be over a hundred MBps. To fully measure the migration prediction with a wide range of configured speeds and to meet the time control requirement of various desired migration times, we modify QEMU at the destination machine not to write the received data to the disk. To ensure that the result is not biased by the disabled writing, we run a set of experiments with enabling and disabling writing at the destination. We vary the client number and compare the average prediction error in both cases. The difference is less than 1%. We vary the desired migration time and compare the difference between the actual migration time and desired time in both cases. The difference is less than 1s again. The results show that disabling writing does not bias the experiment results.

The experiment VMs run VMmark Virtualization Benchmark [19]. VMmark consists of five types of workloads: file server, mail server, database server, web server, java server, each representing different types of applications. Table 2 shows the configurations of those servers. We vary the number of client threads to generate different levels of workload intensity. A simple program is used to generate competing disk I/O traffic on the source machine to create more challenging test scenarios that are more representative of multi-tenancy clouds. It randomly accesses the disk by generating read/write I/O requests. Three models are applied to control the I/O rate by varying the interval between two I/O requests. The static model generate I/O with a constant interval. Two dynamic models generate I/O following the exponential distribution or the Pareto distribution. Each experiment is run for 3 times with different random number seeds.

<table>
<thead>
<tr>
<th>Workload Name</th>
<th>VM Configuration</th>
<th>Server Application</th>
<th>Default # Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>SLES 10 32-bit</td>
<td>Exchange 2003</td>
<td>1000</td>
</tr>
<tr>
<td>Mail</td>
<td>Windows 2003 32-bit</td>
<td>SPECjbb 2003</td>
<td>5</td>
</tr>
<tr>
<td>Java</td>
<td>Windows 2003 64-bit</td>
<td>SPECjbb @2005-based</td>
<td>8</td>
</tr>
<tr>
<td>Web</td>
<td>SLES 10 64-bit</td>
<td>SPECjbb @2005-based</td>
<td>100</td>
</tr>
<tr>
<td>Database</td>
<td>SLES 10 64-bit</td>
<td>MySQL</td>
<td>16</td>
</tr>
<tr>
<td>Server</td>
<td>2 CPU,1GB RAM,3GB disk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>2 CPU,2GB RAM,4GB disk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: VMmark workload summary.

3.3 Prediction of migration time

3.3.1 Naive predictors do not work

In the following experiment, we will show that predicting migration time based on the naive static prediction or dynamic prediction does not work.

Static prediction is based on the formula

$$\text{configured migration speed} = \text{storage size} + \text{memory size}.$$  

It is commonly used when users want to predict their migration time.

We also implement a dynamic predictor, called progress meter based on the migration progress. Whenever migration progress increases by 1%, the predictor records the current migration time $t$ and progress $x\%$, computes the progress rate $\frac{\beta^x}{t}$, and uses that rate to predict the finish time $100\% - \alpha$ dynamically.

The experiment runs on two types of image sizes to represent the typical image sizes in the industrial environment. 160GB is the size of the Amazon EC2 small instance and 8GB is the image size of the VMmark file server workload. We use a micro benchmark to repeatedly write to a data region of the VM’s virtual disk at the specified write rate. The size of the written region and the write rate are varied to create different dirty set sizes.
Table 3: Prediction errors for naive solutions are several orders of magnitude higher than Pacer.

and dirty rates during migration.

Table 3 shows the results. The prediction errors for static prediction and progress meter are several orders of magnitude higher than Pacer, because they ignore the dirty iteration and memory migration periods. They scales up with higher write rates and larger written region sizes. Pacer always achieves small prediction errors.

3.3.2 Pacer in face of uncertain dynamics

We vary multiple dimensions in the migration environment to illustrate that Pacer has good performance when it predicts migration time in those different scenarios. We use fileserver with 8GB storage as the representative workload in many experiments because it is the most I/O intensive workload in VMmark and it challenges Pacer the most. In the first three experiments there is no competing traffic. Pacer computes a predicted time every five seconds and reports it. To avoid prediction spikes due to the sudden, temporary workload shifts, a cumulative average predicted time over all individual predicted times is generated and reported to the user. After migration, we evaluate the accuracy of the prediction by computing the absolute difference of the actual migration time and each cumulative average time, and then report the average of those values of absolute difference in Table 4.

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Vary Write Rate</th>
<th>Vary Written Region size</th>
</tr>
</thead>
<tbody>
<tr>
<td>5MBps</td>
<td>10MBps</td>
<td>15MBps</td>
</tr>
<tr>
<td>Static</td>
<td>32s</td>
<td>95s</td>
</tr>
<tr>
<td>Dirty</td>
<td>31s</td>
<td>97s</td>
</tr>
<tr>
<td>Average</td>
<td>31s</td>
<td>96s</td>
</tr>
</tbody>
</table>

Table 4: Prediction with Pacer types of workload.

**Vary additional competing traffic:** This experiment is based on file server with 15 clients. We vary the intensity of additional competing traffic based on the Pareto model of average 50ms and 90ms sleeping time. The average prediction errors are 4s and 6s.

We observe one more advantage of Pacer, which is that it achieves accurate prediction from the very beginning of the migration. We take the predictions in the first minute and compute the average prediction error for each experiment above. The result errors are within the range of [2s, 12s], which is slightly larger than the one for the entire duration of migration. The reason why Pacer achieves accurate prediction from the very beginning is that the dirty set and dirty rate prediction algorithms are very effective. We will quantify the benefits of these algorithms in Section 3.4.3 in detail.

In summary, Pacer provides accurate average prediction in various scenarios. The prediction error ranges from 1s to 8s across all the above scenarios.

3.4 Best-effort migration time control

3.4.1 Naive time controller does not work

We implement a naive adaptive time controller. It is similar to Pacer except that it assumes no future dirty blocks. The migration speed is computed from remain_pre_copy + existing_dirty_blocks. Similar to the setup in Section 3.3.1, the experiment is on two types of image size, 160GB and 8GB. The micro benchmark is leveraged to generate dynamic write workload on VM. The desired migration time is 6500s for the migration of VM (160GB) and is 400s for the migration of VM (8GB).

Table 5 shows the migration time deviation. The actual migration time with Pacer is very close to the desired time, with maximal deviation of [-1s, +6s]. The migration time with naive solution exceeds the desired time up to 1528s and the deviation gets larger when the workload
Figure 4: Migration with different desired finish times. Pacer almost matches the ideal case when the desired time is larger than 176s. The deviation is very small in [-2s, 2s].

Pacer is more write intensive, because it lacks the capability to predict the amount of remaining blocks for migration and selects a wrong speed. We will show how key components in Pacer help to reduce the deviation later in Section 3.4.3.

3.4.2 Pacer in face of uncertain dynamics

Similar to the experiments for prediction, we vary multiple dimensions in the migration environment to show that Pacer can perform adaptive pacing to realize the desired migration time.

**Vary desired migration time:** This experiment is based on the file server with the workload of 30 clients. We vary the desired migration time from 150s to 400s. The Figure 4 shows that when the desired time is within the range of [200s, 400s], the migration time in the three runs is very close to the desired time, with maximal deviation of [-2s, 2s]. When we lower the desired migration time way beyond anything feasible, the I/O bottleneck limits the migration speed, and consequently Pacer will hit its minimal migration time of 176s, while the default QEMU with the configured speed of 32MBps can finish the migration in 362s.

**Vary the number of clients:** We vary the number of clients from 0 to 60 on the file server. As Figure 5 shows, there exists a lower bound for migration time that is the minimal migration time due to the I/O bottleneck. Default QEMU can only achieve a non-configurable migration time for a specific number of clients. Moreover, when the number of clients increases to 35 or more, QEMU cannot converge and the migration time becomes infinite. This is because QEMU uses a configured speed which is a constant and will not increase even if there is enough I/O bandwidth.

We choose six different desired migration times from 144s to 400s in the feasible region, and migrate VM with different number of clients with those different desired migration times. The results in Table 6 show that Pacer can achieve the desired time in all cases with maximal deviation of [-2s, 1s].

**Vary workload type:** We perform live migration with Pacer for five types of VMmark workloads. In order to guarantee that the default QEMU can converge in the migration, we decrease the number of clients. We run default QEMU first and get the migration time, and then we set this time to be Pacer’s desired migration time. Table 7 shows that Pacer can achieve desired migration time with a small deviation in [-2s, +2s].

**Vary additional competing traffic:** To test whether...
Table 7: Migration time on different types of workload. Pacer can achieve the desired migration time.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Desired Migr Time(s)</th>
<th>Pacer Migr Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs-30 clients</td>
<td>302</td>
<td>360</td>
</tr>
<tr>
<td>iso-30 clients</td>
<td>307</td>
<td>399</td>
</tr>
<tr>
<td>p-16 clients</td>
<td>264</td>
<td>274</td>
</tr>
<tr>
<td>vs-16 clients</td>
<td>287</td>
<td>287</td>
</tr>
<tr>
<td>di-16 clients</td>
<td>471</td>
<td>473</td>
</tr>
</tbody>
</table>

Table 8: Migration time for Pacer when the additional competing traffic varies. Pacer can achieve the desired migration time with a small finish time deviation.

<table>
<thead>
<tr>
<th>Sleeping Time</th>
<th>Run1 Migr Time Dev(s)</th>
<th>Run2 Migr Time Dev(s)</th>
<th>Run3 Migr Time Dev(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Add Traffic</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pareto (ave 30ms)</td>
<td>5</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Pareto (ave 90ms)</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Pareto (ave 90ms)</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

3.4.3 Benefits of key components in Pacer

Dirty set and dirty rate prediction: In order to understand the benefit of key components in Pacer, we design an experiment to compare the systems with and without dynamic dirty set and dirty rate prediction to evaluate the effectiveness of the algorithms. The workload is file server. As Table 9 shows, the actual migration time will exceed the desired migration time significantly without any prediction algorithm. When only the dynamic dirty set prediction algorithm is added into the system, migration time improves but still exceeds the desired time. When both the dirty set and dirty rate prediction algorithms are used in Pacer, Pacer can perform adaptive pacing with very little deviation $[-2s, 3s]$ which is small comparing to the desired time of 264s.

Table 9: Importance of dynamic dirty set and dirty rate prediction. Without any of these algorithms, it is hard to achieve desired migration time.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Desired Time(s)</th>
<th>Pacer w/o dirty set rate prediction(s)</th>
<th>Pacer w/o dirty set rate prediction(s)</th>
<th>Pacer (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 clients</td>
<td>697</td>
<td>286</td>
<td>206</td>
<td>476</td>
</tr>
</tbody>
</table>

Table 10: Importance of speed scaling up algorithm.

<table>
<thead>
<tr>
<th>Desired Time (s)</th>
<th>With</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td>260s</td>
<td>200s</td>
<td>300s</td>
</tr>
<tr>
<td>300s</td>
<td>300s</td>
<td>350s</td>
</tr>
<tr>
<td>350s</td>
<td>350s</td>
<td>350s</td>
</tr>
</tbody>
</table>
4 EC2 Demonstration

To demonstrate the functions of Pacer in a commercial hybrid cloud environment, we conduct a set of experiments using the Amazon EC2 cloud. In these experiments we migrate VMs from the Rice campus network to EC2. On EC2, we use High-CPU Medium instances running Ubuntu 12.04. EC2 instances do not support KVM, thus we use the “no-kvm” mode in QEMU in EC2. The downside is that the performance of VM is not as good as with KVM’s hardware virtualization support.

4.1 Network and disk speed measurements

We run experiments to characterize the network and disk speed that can be achieved between Rice and EC2 and make several interesting observations. First, we use “iperf” to measure network throughput for 200s. We find that when transmitting data from Rice to EC2, the throughput increases gradually and linearly for a surprisingly long 30s before it maximizes at roughly 60MBps. More specifically, 50% of the speed samples fall between 58MBps to 62MBps. After the initial 30s, 5% of the speed samples are below 40MBps and 3.5% are below 30MBps. Based on these findings, we cap the migration speed in the experiments to 50MBps. Second, we use “scp” to copy a 8GB file from Rice to EC2 to measure achievable disk speed. We sample the reported speed every 0.5s. The average speed is 30.9MBps and the variation is 5MBps. Thus, disk speed is the most likely bottleneck for migration in the EC2 experiments.

4.2 Prediction of migration time

To measure the accuracy of Pacer’s prediction, we migrate one VM that runs the file server from Rice to EC2. We vary the number of clients to emulate different workload intensities of the VM server. The CPU utilization rate is 30-45% for the low workload, 45-55% for the medium workload, and 55-70% for the high workload.

For each intensity of the workload we run three sets of experiments and report the average prediction error in Table 11. The first observation is that the accuracy of the prediction does not decrease as the workload increases.

4.3 Best-effort migration time control

In this experiment we migrate the 8GB file server with medium workload and vary the desired migration time from 500s to 800s. For each desired time we run three experiments and report the range of the deviations in Table 12. Although we have reported that the network and disk speeds between Rice and EC2 are not very stable, Pacer still works very well in controlling the migration time to within a small deviation $[-3\sigma, +2\sigma]$ of the desired time.

Table 11: Prediction accuracy with Pacer in EC2.

<table>
<thead>
<tr>
<th>Workload intensity</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Migration Time</td>
<td>22%</td>
<td>20%</td>
<td>25%</td>
</tr>
<tr>
<td>Average Prediction Error</td>
<td>6%</td>
<td>5%</td>
<td>5%</td>
</tr>
</tbody>
</table>

Table 12: Migration time control with Pacer in EC2.

Secondly, given the fact that the network and disk speeds are quite unstable, Pacer still can predict with an average absolute error of about 5s. We find that if disk writes at the destination are disabled to eliminate the impact of disk speed variation, the average prediction error is reduced to 2s. Given the disk speed typically fluctuates 16% from the average speed, the obtained average prediction error ranging from 2% to 3% of the actual migration time is quite desirable.

5 Discussion

Live VM migration progress management functions are sorely missing from today’s hybrid cloud computing management systems. Our contribution is Pacer – the first complete system capable of accurately predicting migration time and managing the progress of a migration to finish as close to a desired finish time as possible. Through extensive experimentations, including a real-world commercial hybrid cloud scenario with Amazon EC2, we show that Pacer is highly effective in all its functions.

Because of space limitation, we will briefly introduce how Pacer help in the use case of coordinating multiple VM migrations (refer to the second use case in Section 1.1). Enterprises usually have applications consisting of multiple interacting components. To coordinate the concurrent migration of VMs, we design a centralized controller by leveraging Pacer’s prediction and time control capabilities. It continuously gathers and analyzes the predicted migration time for each VM, dynamically update the desired migration time for each VM to a feasible target, and paces the migration of the whole set of VMs in such way. We test it on EC2 platform by migrating a SPECweb 2005 webserver cluster which includes a web server and a database server. Pacer could reduce the performance degradation time from 394s to 3s. We are currently extending Pacer to analyze and model the behavior of complex enterprise applications so as to automatically and optimally manage the migrations of such
complex applications.

6 Related work

While to our knowledge no previous work is directly comparable to Pacer, there exists related work on setting the speed or estimating the time of CPU/memory-only VM migration. Breitgand et al. [4] propose a cost function for computing the network bandwidth allocated to CPU/memory-only migration in order to minimize the theoretical number of application delay bound violations as given by a queuing theory model. Akoush et al. [2] simulate the execution of the iterative data copy algorithm of CPU/memory-only migration so as to estimate the required migration time. The simulation makes certain simplifying assumptions such as fixed network bandwidth and fixed or historically known memory page dirty rate. Relative to these previous works, not only does Pacer address a different set of problems in migration progress management for full VM migration, Pacer also takes a system-building approach, using real measurements and run-time adaptation, which are found to be crucial to cope with workload and performance interference dynamics, realized in a complete system.

Beyond VM migration, there is also interesting related work in disk-data-only migration. In [10], Lu et al. presents Aqueduct, a disk-data-only migration system that minimizes the impact on the application performance. However, Aqueduct simply treats the migration as a low-priority task and does not provide a predictable migration time. Dasgupta et al. [7] and Zhang et al. [31] present different rate controlling schemes that attempt to meet a disk-data-only migration time goal. However, these schemes are only simulated. Furthermore, these schemes ignore the problem caused by dirty disk-data generated by write operations during migration.

7 Conclusions

We introduce Pacer – the first migration progress management system. Using a fully implemented system on KVM, we show that Pacer is highly effective and provides large performance benefits across a wide range of realistic and popular virtual machine workloads.

References
