Dependable Estimation of Downtime for Virtual Machine Live Migration

Felix Salfner
SAP Innovation Center Potsdam
Potsdam, Germany
felix.salfner@sap.com

Peter Tröger and Matthias Richly
Hasso-Plattner-Institute at University of Potsdam
Potsdam, Germany
peter.troeger@hpi.uni-potsdam.de
matthias.richly@student.hpi.uni-potsdam.de

***Abstract*—Modern virtualization environments allow the live migration of running systems for load balancing and failover purposes in case of failing hosts. The overall duration of such migration and the short downtime during this process are essential properties when implementing service availability agreements. However, both metrics are currently only determinable through direct experimentation. For this reason, we present a new model for estimating the worst-case values of migration time and downtime in live migration. The prediction is based on a small set of input parameters characterizing the application load and the behavior of the host. We performed a large set of experimental evaluations for the model with three different virtualization products. The results show that total migration time as well as downtime are mainly influenced by the memory utilization pattern inside the virtualized system. The experiments also confirm that the proposed model can predict worst-case live migration performance with high accuracy, rendering the model a useful tool for implementing proactive virtual machine migration.**

***Keywords*-virtual machine; live migration; downtime analysis;**

I. Introduction

The concept of virtualization is known in computer science and IT industry since the late 60's [1]. Today, virtualization can be considered a standard technique in data centers. It is the foundation of modern computing and storage infrastructures such as used in cloud computing. Virtualization provides the following main advantages:

- *Hardware consolidation*. When running a software service on a server, the provider must offer enough capacity to handle peak load. This leads to under-utilization of the resources for most of the time. Virtualization enables the execution of multiple logical servers on the same physical host, which helps to reduce the total number of servers in the data center.
- *Load balancing*. Virtualization allows to control the assignment of physical server resources, such as memory and CPUs, to virtual machines. This assignment can even be changed during runtime. Additionally, services can be moved from one physical host to another by *virtual machine migration*. These techniques are used to manage and balance the load of services.
- *Maintenance*. In the case that maintenance needs to be performed at some physical host, virtual machine migration can be used to move virtual machines away so that the service can be provided while the physical machine is under maintenance.

First approaches to implement migration of a virtual machine relied on suspending the virtual machine before the transmission. In order to reduce the resulting downtime of the virtualized system, researchers and later on vendors turned to so-called *live migration* which reduces virtual machine downtime significantly. Today, the majority of virtualization products support live migration for moving a running virtual machine (VM) to a new physical host with minimal service interruption. This renders live migration an attractive tool also for dependable computing. However, each migration procedure still consumes time and still involves some short service unavailability. In the context of dependable computing the length of both time intervals are of great interest for two reasons: Service availability and proactive fault management.

Hosting a service in a data center is usually accompanied by a service level agreement (SLA) that promises some level of *service availability*. SLAs include not only such requirements, but also penalties if the agreed-on level of service is not met. In case that the service interruption introduced by live migration exceeds the client's expectation on responsiveness, a service unavailability is perceived that decreases overall service availability. Therefore, it is important for data center providers to estimate the worst case downtime for virtual machine migration as precise as possible. This becomes particularly important if the service should be migrated repeatedly following a predefined schedule.

The key notion of *proactive fault management* is to act upon a potential failure even before the failure has actually occurred. The goal is to either perform some action that is able to prevent an imminent failure so that it does not occur or to prepare recovery mechanisms for the likely occurrence of a failure. Both types of actions improve availability, by increasing mean-time-to-failure (MTTF) and decreasing mean-time-to-repair (MTTR). Virtual machine migration has been proven to be an effective tool for proactive fault management (see, e.g., [2]).

One of the key components of any proactive fault management is an online failure predictor that is able to accurately

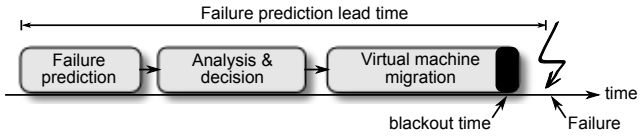


Figure 1. Time intervals involved in proactive fault management. The total duration of virtual machine migration needs to be known in order to determine the necessary failure prediction lead time.

identify imminent failures ahead of time. This must be performed during runtime based on monitoring data (see, e.g., [3] for an overview). Accuracy in this context means that it identifies as many true failures as possible with as few false alarms as possible. Accuracy is inversely related to the length of the time interval how far prediction reaches into the future, which is called the *failure prediction lead time* (see Figure 1): Prediction of failures that happen in the very near future (short lead time) is easier, i.e., more accurate, than a prediction that reaches further into the future (longer lead time). However, short lead times also require faster preventive counter-measure to be conducted. Most failure prediction algorithms allow to adjust their lead time. It should be as short as possible for maximum accuracy, but it has to be sufficiently long such that there is enough time to analyze the current situation, to decide upon which action to take and finally to execute and finish the action before the failure strikes (see Figure 1). In the case of applying virtual machine migration as proactive action, it is hence imperative to have a robust estimate of the maximum duration of the migration procedure.

A. Problem statement and contribution

The majority of existing work assumes some fixed, in many cases arbitrary, duration of the live migration procedure and the virtual machine downtime involved by it. This article systematically investigates the factors determining the duration and downtime of VM live migration (Section II). Building on this analysis, we propose a theoretical model by which the worst-case migration time and downtime can be estimated based on only a few well-defined parameters (Section IV). We demonstrate that our worst-case estimator is making accurate predictions by experiments using a worst-case synthetic load generator (Section VI) and by two benchmark applications for typical data center workload (Section VII).

This article is an extension to our previous work [4]. Whereas previous work focused solely on measuring the effects of various parameters such as the rate at which memory pages become dirty, this work introduces an analytic model that enables to estimate and predict the duration and downtime of virtual machine live migration. We test the estimator using the same load generator that was used in [4] and additionally performed experiments with two benchmark applications.

II. Related Work and Foundations

System virtualization has been a traditional approach for hardware consolidation and resource partitioning in the history of IT systems. The first operating system offering complete virtualization support was CP-40 by IBM in the 1960s. This first design was invented for time sharing operation of virtualized S/360 instances, and still acts as conceptional foundation for all later IBM virtualization technology in the mainframe area.

Meanwhile, virtualization also gained larger attention as research and development topic for other processor platforms. Popek and Goldberg formulated in 1974 [5] a set of essential characteristics for virtualizing *host* system resources for a *guest* operating system:

- *Equivalence*: The execution of software in a virtualized environment should be identical to the execution on pure hardware, despite timing effects.
- *Efficiency*: The majority of code running in the virtualized environment should run at native speed.
- *Resource Control*: The virtualization environment must have exclusive control over the physical hardware resources.

The same publication introduced the notion of a *virtual machine monitor (VMM)* that acts as execution platform fulfilling the given conditions.

Traditionally, the VMM is executing virtual machine instances in a less-privileged processor mode, in order to control relevant system state changes performed by the virtualized system. Popek and Goldberg classified the guest processor instructions accordingly: *Privileged instructions* lead to a hardware trap when they are executed in an unprivileged system mode, and *sensitive instructions* show a behavior that depends on the current system state (e.g. memory, registers). The most relevant aspect for any VMM solution is the handling of instructions that are sensitive, but not privileged.

Adams and Agesen [6] describe three major building blocks for a VMM implementation to deal with the obstacles of a given instruction set architecture. *De-privileging* makes use of the nature of privileged instructions by executing the guest operating system in a lower privilege level of the CPU. The handling of hardware traps occurring from the execution of privileged instructions in this mode is implemented by the VMM, based on a distinct virtual machine state. This relies on *shadow structures* of the hardware state (memory, processor registers) relevant for execution of the guest operating system. When unprivileged instructions can modify relevant system state too, the VMM must also implement *tracing* of such changes by built-in hardware protection mechanisms. One example is the modification of page table entry information to trap on unprivileged memory access operations in the guest operating system.

With the revival of virtualization in recent years, different

optimization strategies were introduced for performance reasons. One is the tighter integration of guest operating system and VMM by implementing a dedicated communication path. This concept, commonly known as *paravirtualization*, relaxes the original equivalence condition by Popek and Goldberg in favor of efficiency improvements.

If the guest software system must remain unchanged, there are two major approaches to deal with critical instructions. A *software-only VMM* implementation utilizes binary translation techniques for critical instructions. Typical solutions apply dynamic late translation during run-time to keep the performance penalty at a minimum. In contrary, a *hardware-assisted VMM* implementation can utilize virtualization support from the physical hardware devices. Modern processor architectures support the management of shadow structures and the de-privileging of guests as explicit features in the instruction set, which reduces again the overhead in comparison to software approaches. Examples are Intel VT, Intel EPT or AMD-V. Most of these techniques rely on the configuration of privileged instructions as part of the *virtual machine control structure* [6] in the processor hardware.

A. Investigated hypervisors

With the given variety of modern VMM approaches, we focused our investigations on three representative system virtualization products. The *Kernel-based Virtual Machine (KVM)* is a hardware-assisted open source VMM for Linux as host operating system [7]. Starting from the Linux kernel version 2.6.20, it is part of the main line and therefore available on all hosts. The virtualized devices for guest systems are provided by a modified version of the QEMU system emulator.

Xen is an open source VMM solution that acts as bare-metal hypervisor. It uses a modified Linux or Solaris operating system as privileged guest in the so-called 'dom0' domain. This domain has exclusive hardware access and management privileges. Guest systems for Xen run in additional domains and access the hardware through paravirtualization interfaces provided by the 'dom0' domain. Xen Linux guests are executed in paravirtualized mode, which requires a modified kernel using the paravirtualization interfaces. For other operating systems, such as Windows, hardware-assisted virtualization is also available in Xen. Different performance studies have shown that the usage of hardware virtualization demands some consideration in the guest operating system configuration [8].

VMware vSphere is a commercial product line for virtualization that relies on *bare-metal* virtualization, meaning that there is no explicitly installed host operating system. KVM and vSphere do not demand changes to the guest operating system due to the utilization of virtualization hardware support. vSphere can also apply binary translation techniques to the guest system, in case the X86 processor hardware is not suited for virtualization support.

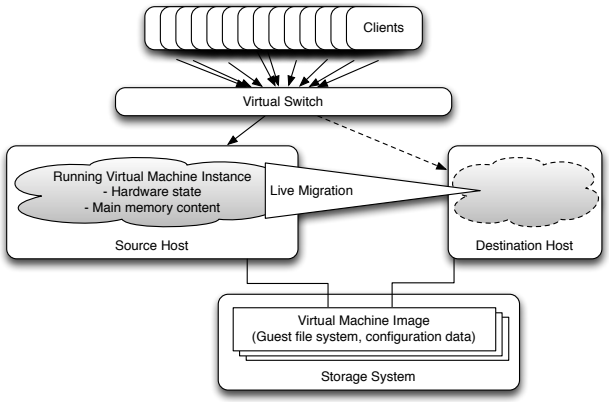


Figure 2. Principle of virtual machine live migration

B. Live migration of virtual machines

The migration of virtual machines is a stable feature of all modern hypervisor implementations, including the presented ones. Starting from early research prototypes ([9], [10]), companies such as VMware made this capability a part of their products since 2005.

Live migration describes the basic principle of moving a virtualized system from one host system to another while the guest is still running (see Figure 2). This activity must consider all hardware state, memory data, storage and network connections of the running virtual machine. Today's products realize this by a two-phase approach:

In the initial *warm-up phase*, the constantly changing main memory of the running guest is incrementally transferred to the destination host. When a product-specific threshold for main memory transfers is reached (time- or amount-based), the implementation switches to the *stop-and-copy* phase. The virtual machine is suspended for a short time period, the remaining resources are copied and the virtual machine is resumed on the destination host.

There are two relevant performance indicators for live migration arising from this concept:

- *Migration time* is the time from start of the live migration process until the virtualization framework declares the physical source host to be no longer relevant for the execution of the migrated virtual machine. The maximum tolerable migration time is determined by internal dependability assumptions at the provider side, e.g., maintenance intervals. It also plays a crucial role in proactive migration scenarios as motivated in Section I.
- *Downtime* or *blackout time* is the phase during live migration when there is a temporary (potentially user-perceptible) service unavailability, caused by the virtual machine suspending execution for the finalization of the movement. From a dependability perspective, blackout time is a crucial quantity when a virtualized service (e.g. server application) needs to fulfill reliability guar-

antees. Blackout time limits are therefore driven by dependability contracts between service provider and customer. Downtime and blackout time are used synonymously in this paper.

The most relevant part of live migration operation is the transfer of main memory state. Since live migration hosts share a common storage system within the migration cluster (see Figure 2), all information kept in the guest file system does not have to be considered. This relates especially to memory regions being swapped out by the memory management of the guest operating system. Some hypervisors also perform their own swapping of memory regions to secondary storage. Several virtualization frameworks use this property for reducing the length of the warm-up phase. A specialized *ballooning driver* allocates large amounts of memory inside the guest operating system, in order to enforce swapping of relevant memory information to secondary storage before migration.

Read-only memory regions (such as code pages) need to be copied to the destination host only once. This makes them a perfect candidate for bulk transfers in the warm-up phase. All remaining main memory information (data, stack, heap, ...) is potentially modified after the live migration process was started, and therefore needs a dedicated copying approach.

Clark et al. discuss the phases of copying memory information in more detail (see Figure 3) [11]:

In the initial *push phase*, the set of pages used actively is copied in rounds to the destination host. Memory regions being modified after transfer are re-sent, or marked for later bulk move when their modification happens too often. We define these modified but not yet transferred pages as *dirty pages*.

In the subsequent *stop-and-copy phase*, the virtual machine is suspended on the source host and resumed again on the destination host. The length of this phase determines the blackout time. Depending on the type of live migration, only portions or all of the remaining dirty pages are copied in this phase. In the former case, a sub-sequent *pull phase* takes care of moving remaining dirty pages from to source to the destination host on demand after VM execution has been resumed at the destination host. The end of the pull phase marks the end of the migration time. Most live migration products combine the first two phases as *pre-copy* approach, and omit the pull phase.

The time of transition from one phase to the next is controlled by product-specific adaptive algorithms. A quick move from push phase to stop-and-copy phase can have a positive influence on migration time, especially when the memory modification happens at high frequency. In contrast, a reduction of blackout time can be achieved by stretching the push phase so that nearly all memory is already transmitted before suspending the machine.

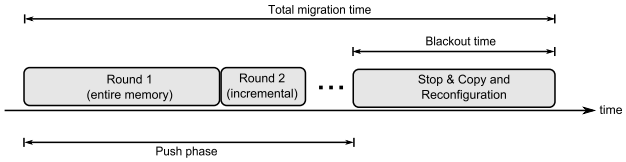


Figure 3. Phases of virtual machine live migration.

C. Analysis of Live Migration

In the area of dependable computing, virtual machine live migration has primarily been used as coarse-grained failover mechanism. Two examples are *proactive fault tolerance* [2] and the approach to resource allocation proposed by Fu [12].

A second group of related work deals with various aspects of implementing VM live migration. Several publications discuss the optimization of live migration and according usage scenarios, mostly with a focus on Xen technology.

Hines and Gopalan [13] discuss the modification of Xen for *post-copy* live migration. In this approach only the execution context of the VM is moved during the push-phase and memory pages are transferred on demand in the subsequent pull-phase after the VM has resumed execution on the destination host. The authors evaluate their solution with a stress test similar to the dirty page generator presented in this paper, the main difference being the distinction between read and write attempts in the memory load, as post-copy requires network interaction also on read attempts. This aspect is less relevant for the commercial products with pre-copy semantics, where read attempts can be served locally. However, for post-copy frameworks, our load model would need an extension with the access type as additional control parameter.

Sapuntzakis et al. [14] introduced several optimization approaches for VM live migration, among which *ballooning* is best-known, which forces the VM to swap out as much memory as possible. The performance investigation was conducted on a simulated work load with GUI end user applications, whereas our work targets server environments with periodic request-response interactions.

Du et al. [15] propose an extension of the Xen live migration mechanisms for improving overall migration performance. They identify the memory page re-writing rate as relevant factor for the migration time and downtime, which is in adherence to our results. The approach relies on a modification of the Xen hypervisor, whereas our work is intentionally restricted to un-modified standard virtualization products.

Nagarajan et al. [16] describe how to achieve pro-active fault tolerance through live migration in a high-performance computing environment. Experiments were conducted with several MPI benchmark applications, where the benchmark type defines the kind of load applied to the Xen live migration facility. Under the consideration of hardware

performance differences, the absolute migration time results from this study are similar to the ones obtained in our measurements. There were no investigations of downtime issues.

To the best of our knowledge this is the first work to introduce a prediction model for virtual machine live migration times and the involved blackout time. A work with close relation to this work is from Clark et al. [11], which – in addition to introducing the phases of live migration – investigates the effect of the size of the *writable working set*, which is the small set of memory pages that are updated too frequently to be coherently maintained on the destination machine. Based on different SPEC benchmarks as application-alike load generators, the authors developed a rate-adaptive algorithm to align the utilized bandwidth for memory pages transmissions. They also propose to stun processes that make live migration difficult. This corresponds to experiences with our Xen environment, where virtual machines with a running dirty page generator were marked as 'uncooperative'. The results are not directly comparable, since they focus on much smaller virtual machine sizes and application requirements.

Several publications discuss the application of live migration over Internet connections [17], [18]. The effects of network latency and bandwidth are more relevant in these cases, but from the perspective of migration load the load model remains the same.

III. Definitions and Assumptions

Having described the fundamentals of virtual machine live migration, a set of major influence factors can be identified that directly affect the performance of virtual machine live migration:

- System load on the source / destination host
- Capacity of the migration network link
- Static configuration of the migrated virtual machine
- System load of the migrated virtual machine

Specific higher-level activity (e.g., application workload) should also be reflected in these basic variables (see also Section VII).

For our further investigation in this article, we assume a typical (and recommended) setup with server applications only running in virtual machine installations. No additionally running processes with significant load are allowed on the physical hosts, except the hypervisor and its support code. This removes the physical host CPU load and memory utilization as control variables to be considered.

Concurrent system load could result from multiple large virtual machines being executed on the same host, so that they have to compete for host resources. This is typically denoted as *over-commitment*. One example is a scenario in which virtual machines with their configured RAM size sum up to an amount larger than the physical RAM available at the host. The hypervisor can make this possible through dedicated swapping to the attached storage system. In such cases, physical RAM for the virtualized operating system might have different performance characteristics, depending on the current over-commitment situation.

Since over-commitment would make hypervisor resource management strategies another variable to be controlled, we favored a performance-oriented system setup, where one virtual machine is running per host at a time. Similar behavior could be achieved by strict partitioning of hardware resources per virtual machine, which is common in mainframe virtualization. With standard processors, a pinning of virtual machines to physical CPUs and the avoidance of host memory exhaustion can lead to comparable results. If such a strong resource partitioning scheme is given, our results, which are based on a single host assumption, can also be applied for multiple virtual machines per physical host.

As final precondition, we assume that the network link between source and destination host has an appropriate available data rate, so that the live migration performance is not influenced by network saturation effects. Practical tests showed that current virtualization products handle the network capacity on the migration link carefully enough, so that this assumption appears valid.

With the given restrictions to static configuration and dynamic load of the migrated virtual machine, the following key factors can be identified:

1) CPU load inside the migrated virtual machine, based on a continuously running application.
2) Memory usage pattern of application and operating system inside the migrated virtual machine.
3) Main memory size configured for the virtual machine.

The memory usage pattern of the running virtual machine must be further separated into relevant factors for live migration performance. Since the memory transfer activities happen in parallel to the operation of the virtual machine, their characteristics can have a relevant impact on the migration performance.

We express the memory utilization pattern using four parameters (see also Figure 4):

The **virtual machine size (VMSIZE)** is the configured main memory size for the virtual machine. This is a constant value during run-time. In typical non-overload situations, the actually used amount of memory is much smaller.

The **working set (WSET)** is the region of the main memory on the source host that must be transferred to the destination host to finish the migration. This value can be roughly equal to the amount of main memory used by the guest operating system and all its processes, or can also be roughly equal to the configured amount of main memory for the virtual machine. This depends on the particular migration strategy of the hypervisor.

The **hot working set (HWSET)** is a subset of the WSET memory set. In our workload model, these memory regions are frequently changed while the migration is taking place.

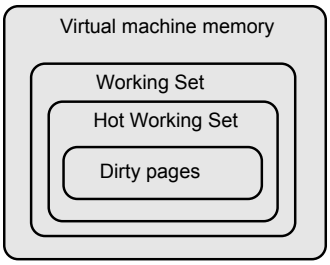


Figure 4. A classification of memory pages

The **modification rate (RATE)** expresses the average amount of memory modified per time unit while the migration takes place. The modifications described by this parameter are assumed to take place only in the memory region described by the HWSET parameter.

We distinguish the HWSET and the WSET to model the fact that there are different kinds of data stored in memory: some data are primarily read and have to be copied only once while others are subject to frequent updates and may have to be copied several times during the push phase, which affects overall live migration performance.

It should be noted that virtual machine live migration – and therefore all parameters listed above – rely on the concept of pages from operating system memory management. Amounts of memory and modification rates are hence expressed in pages but can easily be translated into bytes.

IV. A Predictor for Live Migration Downtimes

Based on the identification of core influence factors for live migration, we propose a predictor that allows to estimate both downtime and total live migration time for a specific application running at a specific (measurable) load. As indicated earlier, there are various use cases for such a predictor: For example, it can be used to plan SLA-bound data center operations, or it can be used to assess arbitrary load conditions which can be useful in system performance and reliability analysis.

The prediction model is based on a set of abstract parameters that express the memory load generated by a particular virtual machine instance, which has been introduced in Section III. Measurement of all parameters will be discussed in Section V.

A. Approach

In all existing virtualization frameworks, memory is managed at the level of memory pages and our model hence also works on this level of granularity. More specifically, the model estimates the number of pages that remain to be copied from the source to the destination host. As discussed in Sect. II, live migration copies the entire memory in a first iteration and then only copies pages that have become dirty. We presume that the number of remaining pages to copy is the key determining factor for the switch from pre-copy to the stop-and-copy phase, and therefore for the amount of time required for both migration and blackout time.

In order to determine the remaining number of pages we estimate the rate at which the number of dirty pages changes during the first iteration as well as during the subsequent copy rounds. The estimate is based on the workload parameters introduced in the last section, as well as on static execution environment characteristics.

The duration of the live migration procedure is determined by the sum of lengths of the various phases. More precisely, migration time is determined by the length of memory pre-copying in rounds, plus the time spent in the stop-and-copy and reconfiguration phase (see Figure 3). The decision to switch from pre-copy to stop-and-copy phase is always influenced by the number of pages that remain to be copied. There are two abstract conditions that can serve as trigger for changing from the first to the second phase:

1) *Condition 1:* The remaining number of memory pages to be copied is sufficiently small.
2) *Condition 2:* The pre-copy phase has already consumed a maximum amount of time.

When the live migration procedure hits one of the two boundaries it enters the stop-and-copy phase. An example for such a scenario is shown in Figure 5. It should be noted that Condition 2 is not present in all virtualization frameworks. In such cases the timeout can simply be set to infinity.

Figure 5 depicts the number of memory pages that remain to be copied over time. As can be seen from the Figure, our model distinguishes between the first round, in which the entire memory is copied, and the subsequent rounds, in which only pages that have become dirty are copied. The two stopping conditions are depicted by dash-dotted lines. The times of moving from one phase to the next are indicated as well. Time t_0 denotes the start of the live migration procedure, t_1 marks the end of the first round of memory page copying, t_2 the end of the pre-copy phase, and t_3 the end of the migration procedure.

B. Computing the Number of Remaining Pages

In order to determine migration times more precisely, we distinguish between different types of memory pages according to the classification shown in Figure 4. We estimate the progression of the number of pages that remain to be copied separately for each category:

- Unused memory pages (those that do not belong to the working set) do not contain data and can be copied in a compressed format and at a higher speed. The remaining number of such *empty* pages is estimated by $e(t)$. The rate at which such pages can be copied is denoted by $r_e \left[\frac{\text{pages}}{\text{s}}\right]$. Some hypervisor implementations might not copy empty pages at all, in which r_e equals infinity.
- Pages that belong to the working set but not to the hot working set need to be transferred only once. The

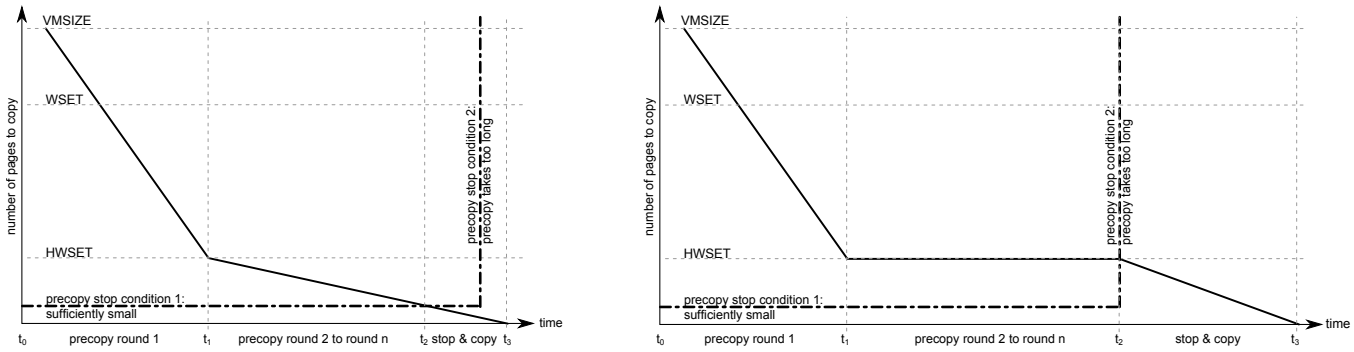


Figure 5. Prediction of live migration times. The predictor is based on the remaining number of dirty pages. The plot on the left shows the case where RATE is significantly smaller than the migration speed r_u. The plot on the right sketches the case when RATE is larger than the migration speed r_u.

remaining number of such *passive* pages is estimated by $p(t)$. The rate at which *used* memory pages can be copied is denoted by $r_u \left[\frac{\text{pages}}{\text{s}}\right]$.

- Pages that belong to the hot working set might become dirty between two successive rounds of copying. The remaining number of such pages is estimated by $h(t)$. These pages are also copied at a rate of r_u.

The number of remaining pages to be copied for a given point in time t is then determined by

$$f(t) = e(t) + p(t) + h(t) \tag{1}$$

The first round of copying in the pre-copy phase is significantly different from the subsequent rounds. Let t_1 denote the time of the end of the first round. We hence divide the definition of $f(t)$ in two phases ($t \leq t_1$ and $t > t_1$).

The remaining number of empty pages $e(t)$ is determined as follows:

$$e(t) = \begin{cases} \text{ESET} - r_e\, t & \text{for } 0 \leq t \leq t_1 \\ 0 & \text{for } t > t_1 \end{cases} \tag{2}$$

where

$$\text{ESET} = \text{VMSIZE} - \text{WSET} \tag{3}$$

is the set of unused, i.e., empty pages. Hence $e(t)$ describes the copying of unused pages at a higher rate. Since the pages in ESET are not used by the virtual machine, there is no contribution of $e(t)$ after the first round.

The remaining number of non-HWSET pages $p(t)$ is defined as follows:

$$p(t) = \begin{cases} \text{PSET} - \frac{\text{PSET}}{\text{WSET}}\, r_u\, t & \text{for } 0 \leq t \leq t_1 \\ 0 & \text{for } t > t_1 \end{cases} \tag{4}$$

where

$$\text{PSET} = \text{WSET} - \text{HWSET} \tag{5}$$

represents the passive (non-hot) working set. Hence $p(t)$ describes the copying of used, i.e., non-empty, pages that the virtual machine does not write to during the live migration process. Such pages have to be copied only once, which is the explanation why $p(t)$ equals zero in subsequent rounds of the pre-copy phase ($t > t_1$).

We assume that the system can copy non-empty memory pages at a fixed rate r_u. However, one fraction of non-empty pages are passive (they belong to PSET) while others are actively written to. We assume that the copy rate r_u is split proportionally among pages from within PSET and active non-empty pages (belonging to HWSET). Hence the copy rate equals $\frac{\text{PSET}}{\text{WSET}}\, r_u$.

The hot working set $h(t)$ is determined by

$$h(t) = \begin{cases} \min\left(\text{HWSET}, \max\left(0, h_1(t)\right)\right); & 0 \leq t \leq t_1 \\ \min\left(\text{HWSET}, \max\left(0, h_2(t)\right)\right); & t_1 < t \leq t_2 \\ h_3(t) & ;\ t_1 < t \leq t_3 \\ 0 & ;\ t > t_3 \end{cases} \tag{6}$$

The formula expresses that $h(t)$ stays within the interval $[0, \text{HWSET}]$. Not surprisingly, the number of remaining pages cannot be negative. The upper limit stems from the fact that applications and the operating system in a virtual machine write to only a subset of the working set – the hot working set HWSET. If RATE (the rate at which memory pages are written) becomes very large, the same pages within HWSET are written several times. The functions $h_1(t)$, $h_2(t)$, and $h_3(t)$ have been introduced for typesetting reasons, only. They express the evolution of the number of memory pages from within HWSET over time.

$$h_1(t) = \text{HWSET} + \left(\text{RATE} - \frac{\text{HWSET}}{\text{WSET}}\, r_u\right)\, t \tag{7}$$

$$h_2(t) = f(t_1) + (\text{RATE} - r_u)\, t \tag{8}$$

$$h_3(t) = f(t_2) - r_u\, t \tag{9}$$

As can be seen from (6), $h_1(t)$ determines the behavior for the first round in the pre-copy phase. As already mentioned, during this phase the copying of used, i.e., non-empty memory pages takes place at a rate of r_u, but the rate is split among passive and active pages (see explanations for Equation 4). The difference between pages of the hot working set (HWSET) and the pages of the passive working

set (PSET) is that hot working pages are written to with a rate determined by RATE. The progression of $h_1(t)$ is hence determined by the difference of the writing rate RATE and the fraction of the copy rate assigned to the copying of pages from the hot working set.

After the first round of copying has finished, copying progresses determined by the function $h_2(t)$. Now the full rate of non-empty page copying r_u can be assigned to the copying of pages from the hot working set, i.e., the copy rate is determined by the difference between RATE and r_u. Due to the use of the minimum operator in (6), it cannot be known a-priori what the value of $f(t)$ is at the end of the first round. We hence refer to this value by $f(t_1)$.

As discussed before, t_2 determines the end of the pre-copy phase, either caused by stop condition 1 or 2. At this point, the virtual machine is stopped and pages do not become dirty anymore. Hence the remaining dirty pages can be copied at the full rate r_u until no dirty pages are left. The latter time is denoted by t_3. Expression $h_3(t)$ describes the final copying. Again, since t_2 is determined by several factors, $f(t_2)$ cannot be known a-priori and we refer to the value $f(t_2)$ in (9).

C. Computing total migration and blackout time

So far we introduced $f(t)$, which estimates the number of memory pages that remain to be copied. However, the ultimate goal is to estimate both total migration time and blackout time of virtual machine live migration. As might have become clear already, both time intervals can be computed from t_2 and t_3:

$$\text{migration time} = t_3 - t_0 \tag{10}$$

$$\text{blackout time} = t_3 - t_2 \tag{11}$$

During the first phase all memory pages are copied once. Hence t_1 can be computed from the sizes of the empty and used memory pages, and the corresponding transmission rates. However, since the definitions of t_3 and t_2 are based on $f(t)$, we need to compute $f(t)$. Specifically, in order to compute t_2, the time of the intersection between $f(t)$ and break condition 1 needs to be determined.[1] The following equations provide the formulas to compute t_1 to t_3:

$$t_1 = \frac{\text{ESET}}{r_e} + \frac{\text{WSET}}{r_u} \tag{12}$$

$$t_2 = \min\left(t_{c1}, t_{c2}\right)) \tag{13}$$

$$t_3 = t_2 + \frac{f(t_2)}{r_u} \tag{14}$$

In (13), t_{c1} denotes the time when $f(t)$ equals the value of the pre-copy stop condition 1 and t_{c2} is the time predetermined by pre-copy stop condition 2. Finally, t_3 determines the end of the copying process and the time can be simply computed by how long it takes to transfer the remaining number of pages when entering the stop-and-copy phase, which is $f(t_2)$.

After the stop-and-copy phase some reconfiguration takes place. Assuming that such reconfiguration is constant and short and that copying memory pages is the determining factor, we neglect this aspect in our model. Results in subsequent sections will show that this already provides sufficiently accurate predictions of migration time and blackout time. We also assumed that the two rates r_e and r_u are independent, which is a valid assumption since the overhead for copying empty pages is very low or even zero, in case the framework does not copy them.

Table I lists the parameters that need to be determined in order to compute (10) and (11). The table lists four types of parameters:

- *Virtual machine-specific.* There is only one such parameter in Table I, which is VMSIZE. This parameter is statically configured when the virtual machine is set up.
- *Situation-specific.* These parameters require online measurements in the running virtual machine. If used for what-if-analyses, these are also the parameters that define the investigated scenario. The following Section V provides details about how situation-specific parameters can be measured.
- *System-specific.* Such parameters have to be determined once for any given setup of networking and physical host equipment. Section V provides an example of how such system-specific parameters can be obtained.
- *Hypervisor-specific.* There are two parameters that depend on the implementation and/or configuration of the hypervisor and which define the criteria to stop the pre-copy phase.

V. Experiment setup

To prove the feasibility of the presented prediction model, we conducted a large set of experiments with both artificial and real application load inside migrated virtual machines.

The test environment consisted of two Fujitsu Primergy RX300 S5 machines acting as migration source and destination. Both machines were equipped with an Intel E5540 QuadCore processor, 12GB of RAM and two Gigabit NICs each. One of the network cards per host was used for the dedicated migration network link. The other network card was used to connect the machine to a shared storage system via iSCSI. The storage system contained all virtual machine image data. If required a third machine was attached to the storage network as controller node.

In all experiments, the migrated virtual machine was either running a Linux 2.6.26-2 (64 bit) or a Windows Server 2008 R2 installation. All virtual machines were configured to have one virtual CPU and a varying amount of (virtualized) physical memory. In all cases, the virtualization guest tools / drivers were installed. Native operating system swapping

[1] t_{c1} has to be set to ∞ in the case that there is no intersection.

Table I
SUMMARY OF PARAMETERS OF THE PREDICTION MODEL

Parameter	Description	Type	Comment
VMSIZE	Configured memory size of the virtual machine	VM-specific	Setup of the the virtual machine
WSET	Working set (allocated memory)	Situation-specific	Measured during runtime
HWSET	Hot working set (actively used memory)	Situation-specific	Measured during runtime
RATE	Dirty page rate (rate at which pages are written)	Situation-specific	Measured during runtime
r_e	Transmission rate of empty pages	System-specific	Measured once for the system
r_u	Transmission rate of non-empty pages	System-specific	Measured once for the system
$c1$	Threshold value for pre-copy stop condition 1	Hypervisor-specific	Predetermined by the hypervisor
t_{c2}	Threshold value for pre copy stop condition 2	Hypervisor-specific	Predetermined by the hypervisor

was activated, but not aggressively in use due to the explicit limitation of the allocated amount of memory.

Experiments for VMware were performed using ESX 4.0.0 (build 208167), using the vCenter server software for migration coordination. High availability features had been deactivated. Experiments for Xen were performed using Citrix XenServer 5.6 (Xen 3.4.2). Both Xen hosts had been configured to form a pool, the test scripts were executed in the 'dom0' partition of the pool master. Experiments for KVM had been conducted with ProxmoxVE 1.7, which relies on QEMU 0.13.0 and a 2.6.32 Linux kernel.

One specific issue was memory management in the Xen environment, namely the *Dynamic Memory Control (DMC)* feature [19]. It allows the Xen hypervisor to change the amount of physical memory made available to the virtual machine at runtime, without reboot of the guest operating system. DMC is an advanced feature necessary to permit memory over commitment in Xen, since Xen never swaps out guest pages, as VMware or KVM do in case.

With activated DMC feature, it was observable that Xen tried to reduce the memory utilization by *ballooning* [14] inside the virtual machine instance before actually starting the migration process. This lead to problems with Linux as guest operating system, since its *out-of-memory (OOM) killer* wrongly assumed an out-of-memory condition from the many locked pages created by the load generator. In several constellations, the combination of DMC, our memory-locking load application and Xen led to random process termination by the OOM killer. We hence deactivated DMC explicitly to achieve repeatable measurements.

Total migration time was measured by capturing the runtime of the products command-line tool that triggers a migration. Downtime was measured by a high-speed ping (50 ms) from another host, since the virtualization products do not expose this performance metric by themselves. The downtime is expressed as the number of lost Ping messages multiplied by the ping interval. We assume here that all ping messages get lost in one continuous time interval during VM downtime.

A. Measuring memory utilization

In contrast to other performance metrics, the RATE parameter is not provided by the OS or any of the hypervisor products directly, probably because of the performance implied by monitoring memory activities. The usual operating system information about dirty pages is not usable here, since this information relates only to the pages not being swapped out by the memory management.

One possible solution could be to obtain direct information from the MMU hardware. Modern processors have special support to monitor low-level activities by performance monitoring units (PMUs). The utilization of such units is supported in Linux through the *libpfm* toolkit or the *perf_events* kernel interface.

We conducted a set of experiments to determine a set of hardware performance events that grow with the RATE parameter of an artificial load. It turned out that for the Intel Nehalem processor under investigation, 21 PMU events showed a strong correlation to the applied dirty page load. Even though this renders PMU a promising mechanism for memory activity monitoring, the application of this approach inside the virtual machine under test is still infeasible. The virtualization hardware and software simply does not support the necessary access to hardware registers.

Reading PMUs on the hypervisor level to infer memory activities of the virtual machine turned out to be infeasible, as we have confirmed in several experiments.

The second possibility for accurate measurements of the memory load is the hypervisor itself. By default, the virtualization products do not expose these metrics to the outside. Nevertheless, the hypervisor and its live migration facility use a tracking mechanism to identify pages that have become dirty. Therefore, we modified the source code of KVM slightly to facilitate measuring of the RATE parameter as will be documented in the next section.

B. KVM hypervisor extension for memory tracking

KVM consists of two parts, the KVM subsystem in the kernel and the qemu-kvm user space application. The user space application creates the virtual machine inside its own address space and communicates to the

KVM subsystem using I/O controls. The KVM kernel subsystem interfaces *KVM_SET_MEMORY_REGION* and *KVM_GET_DIRTY_LOG* allow the caller to keep track of dirty page state changes for the given virtual machine. KVM_SET_MEMORY_REGION can enable/disable dirty pages tracking, and KVM_GET_DIRTY_LOG returns a bitmap with all dirty pages since the last call. This allowed us to enable dirty page tracing on demand for measurements, even without having an actual live migration taking place.

C. Classifying hypervisor and system

The two system-specific parameters of our model, namely the transmission rates for empty and non-empty memory pages have been estimated using a virtual machine with a defined non-paged memory footprint as migrant.

In KVM, the hypervisor-specific parameters are determined by two parameters that can be passed to the hypervisor when initiating the live migration procedure. The number of pages that is considered to be sufficiently small is computed by the product of the KVM configuration parameters `migrate_speed` and `migrate_downtime`. The first parameter determines the maximum speed (in bytes per second) for the pre-copy phase of migrations while the second specifies the maximum tolerated downtime. If there are less remaining pages than `migrate_speed` × `migrate_downtime`, the stop-and-copy phase can be performed faster than the maximum tolerated time `migrate_downtime`.

For the pre-copy stop condition 2, the behavior of ProxmoxVE KVM can be expressed by the following equation:

$$t_{c2} = \frac{2\,\mathrm{VMSIZE}}{\texttt{migrate\_speed}} \tag{15}$$

This means that whenever the time has passed that would be sufficient to copy two times the entire virtual machine memory, the pre-copy mechanism is stopped and the remaining pages are copied at a modified rate in the stop-and-copy phase.

D. Determining situation-specific parameters

To determine the RATE parameter for dirty pages, we enabled the described dirty pages logging on all memory regions. Our modified KVM implementation measures the number of pages that have become dirty once every second. The RATE parameter is the average of the measured numbers.

The determination of the HWSET is more complex. We defined the HWSET to consist of all pages that are *frequently* changed. We estimated HWSET by determining the set of pages that have become dirty in a series of measurements. After computing the union of all pages that have become dirty in these measurements we counted only pages that have been marked dirty in a minimum number of measurements. The time interval between the individual measurements should be at least as long as we expect one migration pre-copy round to take, which is $WSET/migrate\_speed$ in the worst case. In our experiments with KVM, we used ten consecutive measurements within one minute and we considered only those pages as hot pages that were marked dirty in all ten measurements.

VI. Experiments with artificial load

Based on the theoretical investigation of relevant workload parameters and the described setup, we conducted a set of experiments for proving the feasibility of the model. In the first step, we conducted experiments with artificial work load generators. The intention was to stress the virtual machine migration in a controlled and reproducible way, before analyzing the impact of real-world application workload.

Since our set of relevant dynamic factors is restricted to the behavior of the guest operating system, we were able to perform all experiments with load generators inside the virtual machine. For the worst case analysis, we utilized load generators for CPU, locked pages and dirty pages.

The *CPU load generator* was used to produce artificial CPU load inside the virtual machine, in order to prove the independence of migration performance from the virtual machine computational load. We used the commonly known *burnP6* and *cpulimit* tools for generating a controllable CPU utilization. Our experiments proved that CPU load has negligible impact on virtual machine migration (see also [4]).

The *locked pages generator* was used to analyze the effects of static memory allocation. With this tool, locked pages are pinned in memory through operating system calls so that they cannot be swapped out. This ultimately increases the WSET value alone, without influence on both RATE and HWSET. The implementation first allocates a given amount of locked pages memory. In the next step, random data is written once to this memory region, in order to trigger delayed page table modification schemes in the operating system [20]. After that, the according regions are pinned by a system call.

The *dirty pages generator* was developed to artificially influence HWSET and RATE parameters in an experimental environment. This load application simulates a cyclic memory modification pattern by continuously writing pre-computed random data to pinned memory in round-robin fashion. This execution model is motivated by server applications that modify memory regions based on incoming requests. Those modifications have comparable characteristics for the majority of requests. Such servers are always reading some data, storing logging information in main memory, and return the computational result. The request inter-arrival time is assumed to show a constant average rate, so the modification attempts in memory can be modeled just by using parameters expressing the frequency and intensity of using a block of memory.

In order to remove systematic errors, a proper design of experiments usually demands randomization of the runs. In our scenario, an experiment is the migration of a virtual machine for a specific configuration of parameters. Randomizing this would identify potential unconsidered influences on the dependent variables. However, due to the closed experimental environment (no other users had access to the machines), and full automation of the measurements, we expect no major additional influence on the dependent variables. Selective tests confirmed this assumption. We therefore relied only on measurement series with predefined continuous data ranges.

The migrated virtual machine was running a load generator in a fresh operating system installation only. Before migration start, several minutes of warm-up time have been reserved for the virtualized operating system.

A. Influence of CPU load

For the investigation of the influence of the CPU load factor, we performed at least 10 migrations per CPU utilization degree, ranging from 0% to 100% artificial load in steps of ten. The results show that migration times of all virtualization products are not influenced by the CPU load. More precisely, migration times varied around mean values of up to 26 seconds within a 95% confidence interval of not more than +/- 1s (see [21] for details). As additional feasibility test, we investigated Xen both with and without activated DMC feature, which had serious impact on the absolute migration time, but the impact of CPU remained negligible.

The results suggest that virtualization frameworks reserve enough CPU time for their own management (migration) purposes. Live migration scenarios seems to be dependent only on non-CPU utilization factors.

The result convinced us that we could safely drop CPU load as an influencing factor in subsequent experiments.

B. Influence of WSET and filling degree

Using the locked pages generator, we varied the WSET parameter from zero to 90% of the main memory configured for the virtual machine (VMSIZE).

Our results showed that the VMware hypervisor has a linear dependency of migration time on memory utilization, while the downtime is not influenced significantly. With Xen, both the downtime and the migration time remained nearly constant in all memory utilization scenarios. The Xen virtual machine migration time depends mainly on the absolute amount of configured main memory.

In order to rely on the trap and page table mechanisms of the operating system, all virtual machine migration approaches copy memory content in the granularity of pages. Hence, an entire page has to be migrated even when writing only to a fraction of a page. We tested this assumption by "filling" memory blocks inside the locked region to a varying degree. We used a block size equal to the system page size (4kB) and conducted experiments with varying filling degrees of such blocks. As expected, all three virtualization toolkits showed no effect on downtime or migration time.

Changing only a single bit in a memory page makes it dirty from the viewpoint of the hypervisor, and therefore also a relevant candidate for live migration. We see a relevant issue here for 64 bit systems with potentially larger page sizes. In such systems the overhead of migrating only marginally modified pages could become significant. For our purposes, the conclusion is that the filling degree does not have to be considered in subsequent experiments.

C. Influence of HWSET + RATE + VMSIZE

We conducted a large set of multi-parameter experiments with the dirty page load generator, in order to determine the basic patterns of influence in virtual machine migration. The goal here was to determine the different worst-case settings for the combination of HWSET, RATE and VMSIZE. For this reason, we performed experiments according to a full factorial design, meaning that all possible combinations of parameter levels have been measured in the experiment. In each experiment we measured migration time and downtime as response variables. For Xen, we investigated a total number of 528 combinations (treatments), each with 20 measurements resulting in an overall number of 10560 migrations. In case of the VMware hypervisor, we performed experiments for 352 combinations resulting in 7040 migrations. For KVM, we tested 1652 combinations resulting in 33040 migrations.

As we have three factors (plus a response variable) we cannot present the entire results in one plot. Since VMSIZE has significantly less levels, we decided to plot the mean response, i.e. mean migration time or downtime, over HWSET and RATE for a fixed value of VMSIZE. The experiments have been performed using the DPG load generator, which simulates worst-case behavior in terms of memory usage.

One example for the results is the behavior of the Xen hypervisor. Downtime in general increases with increasing HWSET and increasing RATE (see Figure 6). This is not surprising as an increased usage of memory (more pages written at an increasing rate) requires more memory to be transferred in the stop-and-copy phase. We can also conclude from the figure that HWSET seems to have a linear effect on downtime, if the RATE is above some threshold value and regardless of the VMSIZE. This threshold value is around 30,000 pages/s or 117 MB/s with 4KB pages, which corresponds well to the expected migration speed over a 1 Gigabit Ethernet link.

One peculiarity in Figure 6 is the abrupt change at a RATE level around $30{,}000\frac{1}{s}$. In order to analyze this further, we conducted additional "zoom-in" experiments that investigated a sub-range of values for RATE at greater level of detail (see Figure 8-a). As it can be seen from the plot,

the change is not as abrupt as might have been concluded from Figure 6.

Turning to total migration time (Figure 7), we observe a sudden change at the same level of RATE as we have observed for downtime. Again, the "zoom-in" analysis shows that the change is smooth although rather steep. However, in general the mean migration time is more irregular. It came as a little surprise to us that for RATE levels "above the jump" total migration time decreases with increasing RATE. In order to check that this behavior really occurs we have carried out separate experiments specifically targeted to this question with the same consistent result. Although we cannot give a precise explanation, we presume that it is caused by the rate-adaptive algorithm employed by the hypervisor. This supports our assumption that a load model is essential in order to assess duration of live VM migration.

The effect of VMSIZE can be observed by comparing the two sub-figures 7 (a) and (b). It can be seen that VMSIZE has a non-trivial effect on migration time: since the shapes look very different at different levels of VMSIZE, the effect does not appear to be linear, except for the case where RATE equals zero.

There is no effect of any HWSET value if RATE is zero, which is consistent with the single variable experiments described in Section VI-B.

The plots in Figures 6 to 8 show migration times averaged over all measurements. In order to assess the variability in the data, we describe the ratio of maximum to minimum values as well as standard deviation for the data in Table II. Two ratios and two standard deviations are reported: the ratio of the maximum treatment mean to the minimum treatment mean and the ratio of the maximum to the minimum values across all measurements. Regarding standard deviations, the table describes the largest standard deviation computed within each treatment (parameter combination) as well as the standard deviation for the overall data set. In addition, the table reports the mean time averaged across all measurements. The data quantifies what has also been observable from the plots: Both migration time as well as downtime vary tremendously depending on VMSIZE and RATE.

For XenServer, one can observe that downtime is only 8.6% of the overall migration time. The fact that the overall standard deviation is far greater than the within-cell standard deviation supports the observation that there is a strong systematic variability in the live migration algorithm.

For KVM, Figure 9 and Figure 10 show the behavior under different conditions for HWSET and RATE. While the downtime behavior is comparable to Xen, the migration time development shows a completely different behavior. Here, above a certain RATE the migration times line up at a constant level, which is independent of the HWSET. A comparison of the two subfigures shows that this level is dependent on the VMSIZE. What we see here is the effect of the second stop condition explained above, which strikes if the RATE is larger than the effective migrate speed.

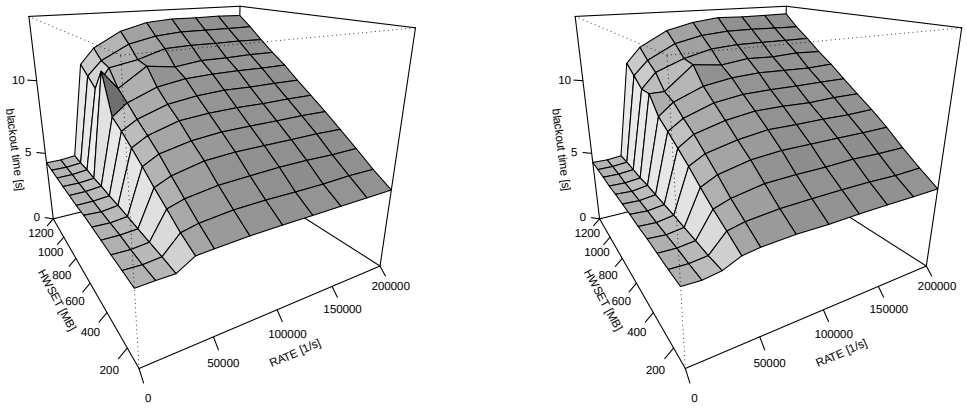
Since the VMware end user license agreement does not allow the publication of performance numbers, we omit the presentation of the gathered data here. We can report that the observed behavior of vSphere differs significantly from the one of Xen, which emphasizes that the choice of the hypervisor product can have significant impact on availability. The main reason for the different behavior seems to be the different rate-adaptive algorithms employed in the virtualization products. Rather than arguing which behavior is better we want to emphasize that it is mandatory to take the specific virtualization product into consideration when making assumptions on migration duration.

Regarding the max:min ratio of downtime computed from treatment means with VMware, we have observed a ratio of 16.27. This shows that due to different memory load, the maximum mean downtime can be 16.27 times as large as the minimum mean downtime. If we do not consider mean downtimes but the maximum and minimum value observed across all experiments, the factor even goes up to 23.83. The conclusion from this observation is that if service downtime is critical for meeting reliability goals, a realistic assessment of reliability can only be achieved if the maximum downtime for the application-specific memory load is figured out, which is the goal of our prediction model.

D. Comparing the predictor with experimental results

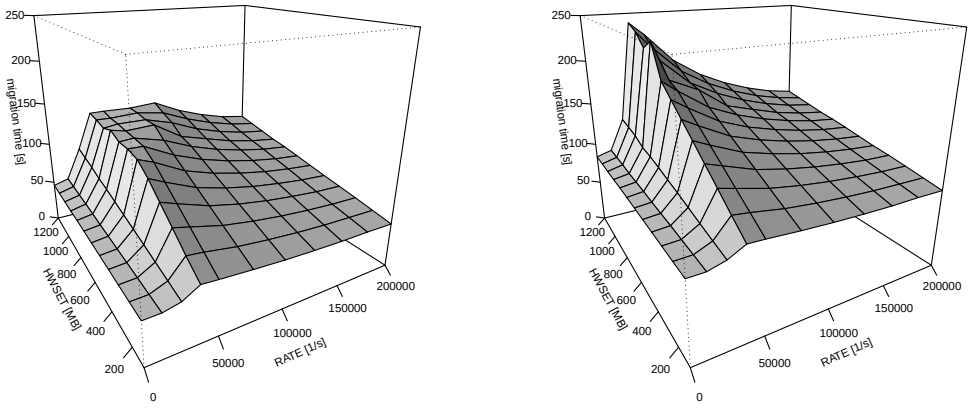
In order to test the feasibility of our predictor model, we compared the experimental results for KVM with the theoretical worst case assumptions from our model. We relied on the KVM results here, since the hypervisor modifications described in Section V-B allowed a fine-grained monitoring of the relevant metrics. Figure 11 shows the comparison for blackout time and migration time. In the absolute majority of cases, the model was able to provide a worst case prediction close to the real-world experiment results:

- For a total of 33040 measurement points, the model predicted migration times that were larger or equal to the corresponding measured migration times in 95.6% of the measurements. For blackout time, the predictor was right in 97.08% of the cases.
- The average absolute error, meaning the distance between the computed worst case value and the measured value, for the migration time prediction was 25,75s. For blackout time prediction, the average absolute error was 2,45s.
- The average under-prediction, meaning average error in the cases where the predicted value was below the actual measured value, was 2,95s for the migration time. For blackout time, the average was 0.13s.



(a) VMSIZE=4096 (b) VMSIZE=8192

Figure 6. Mean downtime for Xen



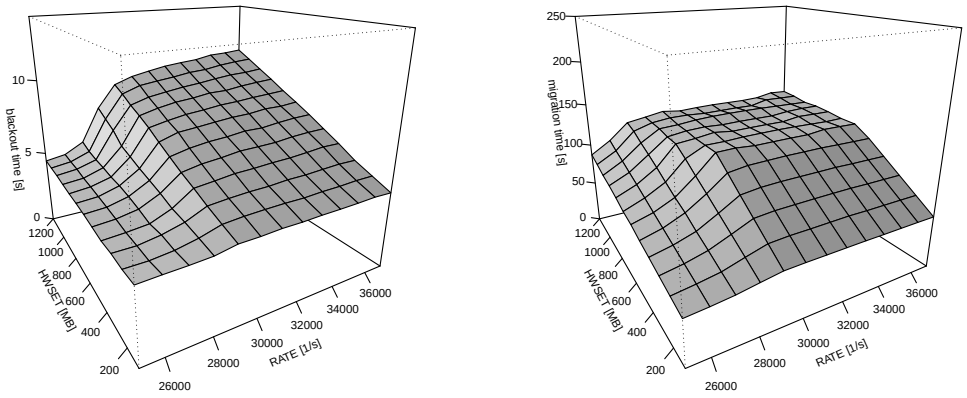
(a) VMSIZE=4096 (b) VMSIZE=8192

Figure 7. Mean migration time for Xen

E. Discussion

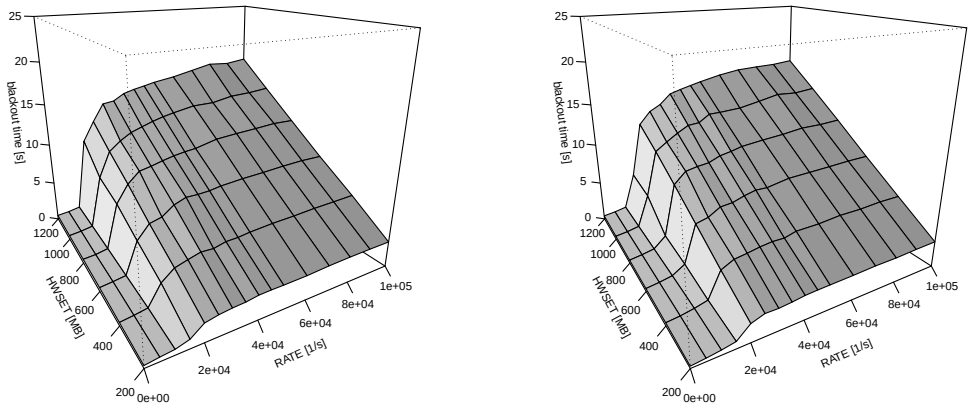
The statistical evaluation proves that our prediction model worked in more than 95% of the worst case load tests, which is especially important for dependability-related use cases. If a proactive failure predictor is able to implement a lead time larger than the worst case value from the migration time prediction model, than virtual machine migration can be used as preventive recovery strategy.

In order to understand the experiment results in more detail, we performed a source code analysis of Xen and had personal communication with VMware representatives. Live migration in fact is mainly related to the rate-adaptive migration control algorithm realized in the product. The relevant aspect here is the dirty page diff set – the fraction of pages that is scheduled to be copied in each next round of the pre-copy phase. The virtualization products identify ”hot pages” in this set and shift such pages more aggressively to the stop-and-copy phase, since the transfer in the stop-and-copy phase is potentially more effective, depending on “hotness” of the page, network link speed and other factors. This also appears to be an explanation for the increasingly large gap between predicted and measured migration and blackout times for large memory allocation sizes. Future extensions to our prediction model could take such effects



(a) Mean Downtime (b) Mean Migration Time

Figure 8. Xen behavior in the zoom-in area (VMSIZE=4096)



(a) VMSIZE=4096 (b) VMSIZE=8192

Figure 9. Mean downtime for KVM

into account to improve prediction accuracy.

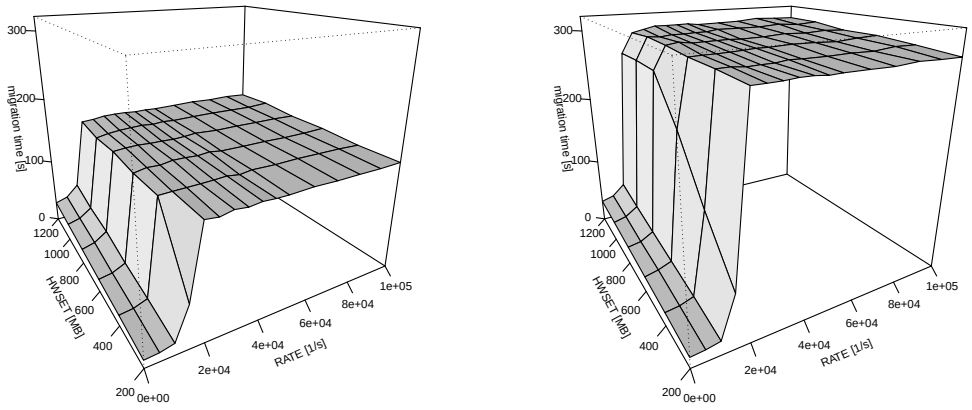
Akoush et al. [22] made similar investigations in their live migration performance analysis. Surprisingly, the downtime seems not to be influenced by the chosen strategy, which can be explained by the broad transmission capacity of the network link. Comparative measurements of the network saturation supported this assumption.

VII. Experiments with Real Load

For a further proof of the proposed migration and blackout time prediction model, we conducted another large set of experiments with real application load. We decided for two typical server application representatives – the SPEC jAppServer benchmark, and the Postal SMTP server benchmark in conjunction with the Postfix mail server.

A. SPEC Benchmark Results

The first set of tests relied on the SPEC jAppServer 2004 1.08 benchmark application. This program is intended to measure the performance of Java 2 Enterprise Edition (JavaEE) application servers. The benchmark simulates manufacturing, supply chain management, and order/inventory business processes. It consists of a database part and several JavaEE applications to be deployed. A driver component



(a) VMSIZE=4096

(b) VMSIZE=8192

Figure 10. Mean migration time for KVM

Table II
DATA VARIABILITY

Hypervisor / Guest	time	Mean time [s]	Max:Min Ratio		Standard Deviation	
			Mean	Overall	Treatment Max [s]	Overall [s]
XenServer / CentOS	migration	89.73	9.01	9.10	6.32	39.08
	downtime	7.69	3.17	3.46	0.62	2.94

simulates parallel client requests, where the request rate is controlled by a parameter called *txRate*.

We performed measurements for a total of ten configurations, each corresponding to a specific setting of the benchmark's *txRate* parameter (see Table III). For each setting we conducted more than 2300 migration experiments to collect statistically significant data. In each experimental run we measured total migration time and blackout time of the migration. We also determined for each configuration the values for WSET, HWSET and RATE using our modified KVM hypervisor.

Figure 12 shows the experimental results. The graphs plot measured blackout times (Fig. 12-a) and measured migration times (Fig. 12-b) together with the times predicted by our model. For measured blackout and migration times we also plotted 95% confidence intervals shown by vertical bars. We decided to plot absolute times rather than relative prediction accuracy since in real world dependable application scenarios absolute numbers are much more relevant.

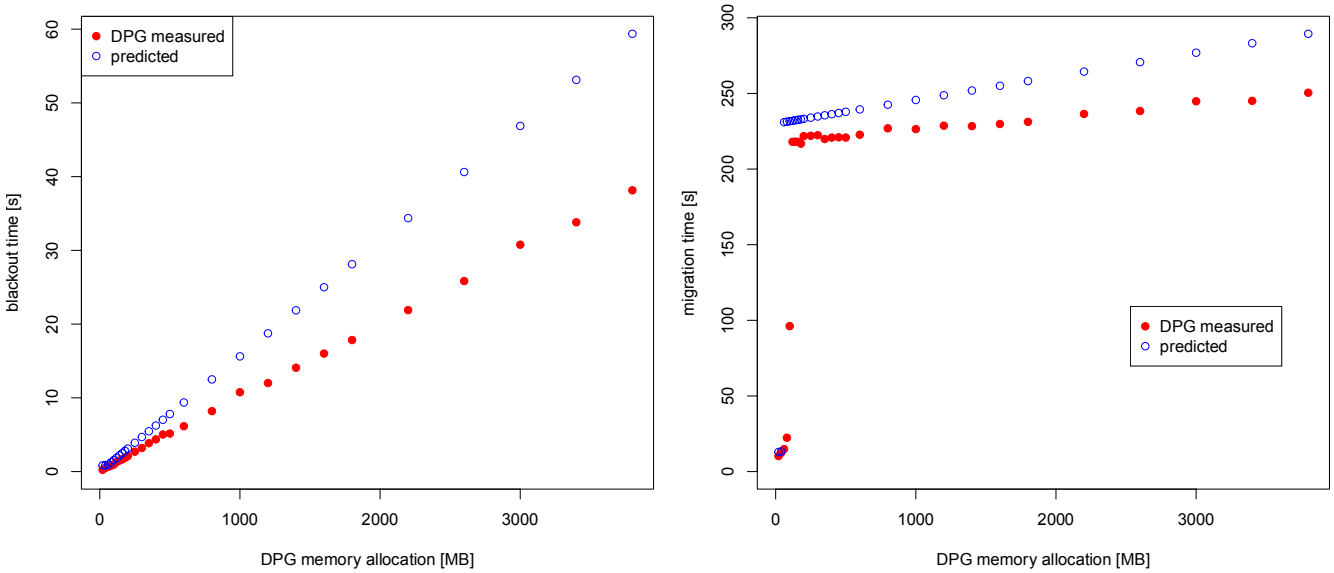
The graphs show that the proposed prediction model works well also for real applications. It can be seen that overall the predictor follows the non-linear shape of the curve, although there is significant over-prediction for parameter settings three and four for blackout time, and for settings two and three for total migration time.

In 98.03% of the cases, the worst case predictor returned a blackout time greater or equal to the corresponding measured time. The absolute error for blackout time prediction was 1.26s on average (4.01s maximum). Due to the fact that an under-estimation of downtime is critical, we separately investigated the cases in which our model predicted shorter migration times than the measured ones. The average deviation in these cases was 0.40s (3.20s maximum). For total migration time, the corresponding numbers are 97.56% accuracy, with an average absolute error of 38.12s (267.5s maximum) and an average under-estimation of 1.15s (9.50s maximum).

B. Postal SMTP Benchmark Results

As a second application benchmark we used the Postal SMTP benchmark 0.7 in conjunction with a Posfix 2.5.5. mail server. To get as close as possible to a realistic workload, we added a Spamassassin 3.2.5. installation to the configuration of the mail server. Postal sends SMTP requests of different kinds to the mail server running in the virtual machine. The varied parameter in our experiments is the number of SMTP messages per minute sent by the Postal application.

Similar to the SPEC benchmark we measured blackout and migration times for ten settings and determined the corresponding values for WSET, HWSET and RATE (see Table IV). Due to increased volatility of the Postal SMTP



(a) Blackout time (b) Migration time

Figure 11. Dirty page load generator (DPG) vs. worst case predictor

Table III
SPEC PARAMETER SETS

Setting	SPEC Driver txRate	Average WSET (pages/s)	Average HWSET (pages/s)	Average RATE (pages/s)
1	5	371228	41962	7802
2	10	388346	58787	13111
3	15	488656	71594	17993
4	20	569836	82140	22911
5	25	695151	86636	27744
6	30	688627	90284	33707
7	35	705575	93165	37911
8	40	732491	100686	43989
9	45	761932	104089	58090
10	50	756850	114790	59533

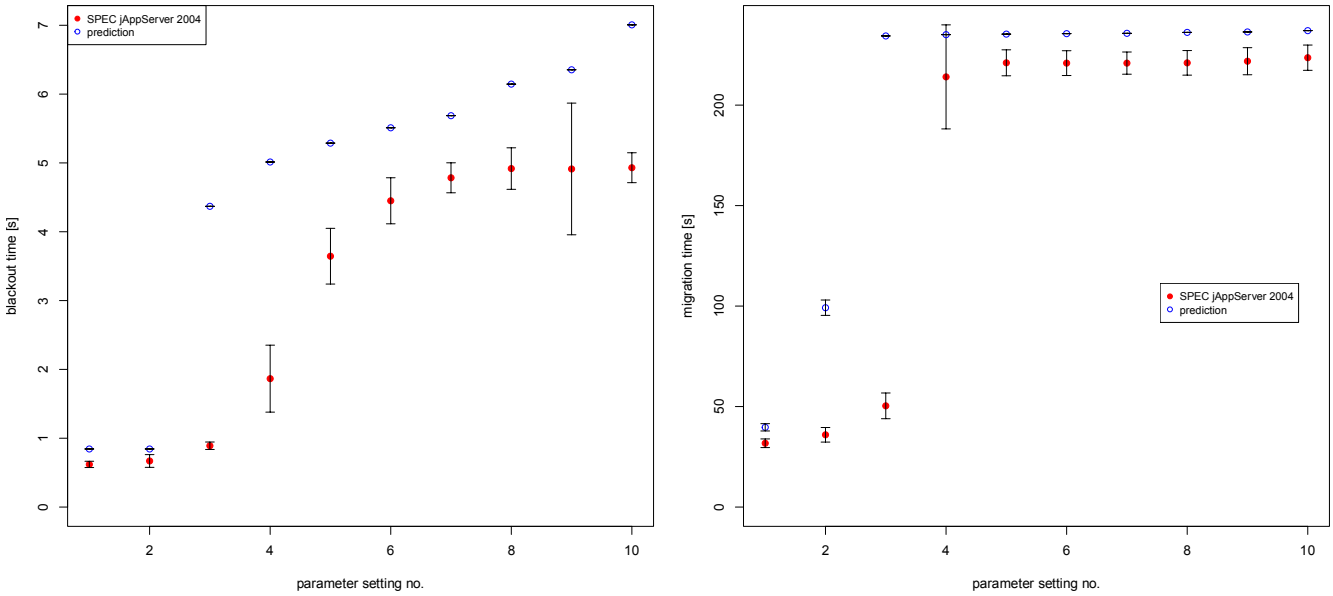
benchmark scenario we performed more than 3500 runs per setting in order to obtain statistically significant numbers.

Figure 13 shows the experimental results. Again, our prediction model resulted in relatively accurate predictions. The plots also show the necessity to leave some headroom for predictions. As can be seen from Figure 13-b, due to the increased volatility the 95% confidence intervals get close to the predicted values. More specifically, our predictor delivered a migration time that was above or equal to the measured performance in 90.18% of the measurements. The average absolute error in the migration time prediction was 62.38s (287.58s maximum), and the averge under-prediction was 36.27s (69.23s maximum). For blackout time, the worst case predictor was safe in 83.6% of all measurements. The average absolute error for blackout time prediction was 0.57s (1.47s maximum), and the average under-prediction error was 0.45s (1.05s maximum).

The results for the SMTP benchmark showed sub-optimal prediction quality for virtual machines with small VMSIZE. If only experiments with a VMSIZE value larger or equal to 4GB are considered, the migration time prediction success rate improve significantly. More specifically, the numbers are for migration time 98.85% accuracy with an average under-estimation of 2.01s (5.42s maximum), and for blackout time prediction accuracy goes up to 96.98% with an average under-estimation of 0.09s (0.27s maximum).

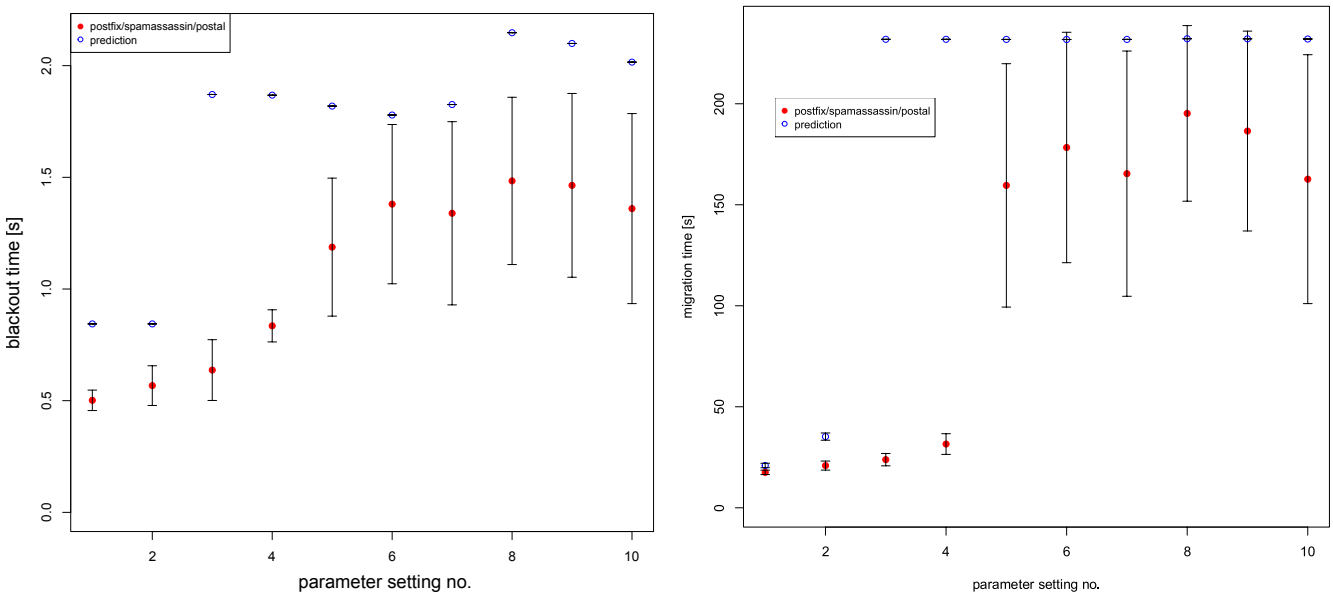
C. Discussion

The experiments have shown that our prediction model is able to forecast both total migration times as well as blackout times of real world applications. As it is the case for all worst-case predictors, predicted values have to be larger than the measured numbers but should nevertheless be as close as possible. This trade-off between accuracy and safety is well-known from other areas such as determination of the worst-case execution time (WCET). In our case the prediction is on the safe side in more than 96.98% of all cases.



(a) Blackout time (b) Migration time

Figure 12. SPEC jAppServer 2004 load vs. worst case predictor



(a) Blackout time (b) Migration time

Figure 13. Postal 0.7 + Postfix 2.5.5 application load vs. worst case predictor

Table IV
POSTAL / POSTFIX PARAMETER SETS

Setting	SMTP messages / minute	Average WSET (pages/s)	Average HWSET (pages/s)	Average RATE (pages/s)
1	100	154035	18348	8002
2	200	185759	32494	12377
3	300	210190	30654	16217
4	400	244276	30604	21003
5	500	443361	29802	23968
6	600	516652	29148	26023
7	700	559266	29917	27488
8	800	712506	35185	28080
9	900	765597	34396	27631
10	1000	728675	33028	27903

VIII. CONCLUSION

With growing capacity of commodity server hardware and increased consolidation efforts, virtualization has become a standard approach for cloud data center operation. Live migration of virtual server workloads can be employed to implement workload-driven system management as well as a mechanism to free server hardware that is due for maintenance and repair. However, in order to give guarantees on application availability or responsiveness as well as for proactive fault management, solid estimations either about the total duration of live migration or the length of service downtime are badly needed.

In this paper, we have presented a model that predicts total migration time as well as service blackout times based on a small number of characteristic parameters: virtual machine-specific parameters, i.e., the overall size of the virtual machine's memory, situation-specific parameters such as the size of working set, the size of the hot subset of the working set, i.e., the number of memory that are actively written, and the memory page modification rate, system-specific parameters such as memory page transmission rates over the network as well as hypervisor-specific parameters modeling the hypervisor's live migration strategy.

By carrying out a large number of experiments, we have shown that the prediction model is able to reliably forecast migration times in more than 95% of all cases. This holds for a worst-case load generator as well as for real-world server applications.

Our results are promising in the sense that they show applicability of live migration for scenarios where workloads have to be moved off potentially breaking servers. The experiment results show a remarkable performance of virtual machine migration even under unfair conditions. The performance numbers typically do not exceed the lead-time of state-of-the-art failure prediction algorithms, which makes the idea of proactive virtual machine migration a promising topic for future research.

REFERENCES

[1] R. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.

[2] C. Engelmann, G. Vallée, T. Naughton, and S. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 252–257.

[3] F. Salfner, M. Lenk, and M. Malek, "A Survey of Online Failure Prediction Methods," *ACM Computing Surveys*, vol. 42, no. 3, pp. 10:1–10:42, Mar. 2010.

[4] F. Salfner, P. Tröger, and A. Polze, "Downtime Analysis of Virtual Machine Live Migration," in *The Fourth International Conference on Dependability (DEPEND 2011)*. IARIA, 2011, pp. 100–105.

[5] G. Popek and R. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[6] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 2–13, Oct. 2006.

[7] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Ottawa Linux Symposium*, Jul. 2007, pp. 225–230.

[8] N. Bhatia, "Performance Evaluation of Intel EPT Hardware Assist," http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf, Mar. 2009.

[9] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 25–25.

[10] K. Onoue and Y. Oyama, "A Virtual Machine Migration System Based on a CPU Emulator," in *2nd International Workshop on Virtualization Technology in Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2006, p. 3.

[11] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[12] S. Fu, "Failure-aware resource management for high-availability computing clusters with distributed virtual machines," *Journal of Parallel and Distributed Computing*, vol. 70, no. 4, pp. 384–393, 2010.

[13] M. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2009, pp. 51–60.

[14] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 377–390, 2002.

[15] Y. Du, H. Yu, G. Shi, J. Chen, and W. Zheng, "Microwiper: Efficient Memory Propagation in Live Migration of Virtual Machines," in *39th International Conference on Parallel Processing*, 2010.

[16] A. Nagarajan, F. Mueller, C. Engelmann, and S. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 23–32.

[17] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. Laat, J. Mambretti, I. Monga, B. Oudenaarde, S. Raghunath, and P. Wang, "Seamless live migration of virtual machines over the MAN/WAN," *Future Gener. Comput. Syst.*, vol. 22, no. 8, pp. 901–907, Oct. 2006.

[18] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *3rd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2007, pp. 169–179.

[19] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[20] M. Russinovich, D. Solomon, I. Books24x7, and S. B. Online, *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, 2005.

[21] P. Tröger, A. Polze, and F. Salfner, "On the Applicability of Virtual Machine Migration for Proactive Failover," in *SDPS International Conference, Special Track on Virtualization*, 2011.

[22] S. Akoush, R. Sohan, A. Rice, A. Moore, and A. Hopper, "Predicting the Performance of Virtual Machine Migration," in *18th Annual IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 37–46.