Live Migration of Virtual Machines

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen†,
Eric Jul†, Christian Limpach, Ian Pratt, Andrew Warfield
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK
firstname.lastname@cl.cam.ac.uk
† *Department of Computer Science*
University of Copenhagen, Denmark
{jacobg,eric}@diku.dk

Abstract

Migrating operating system instances across distinct physical hosts is a useful tool for administrators of data centers and clusters: It allows a clean separation between hardware and software, and facilitates fault management, load balancing, and low-level system maintenance.

By carrying out the majority of migration while OSes continue to run, we achieve impressive performance with minimal service downtimes; we demonstrate the migration of entire OS instances on a commodity cluster, recording service downtimes as low as $60ms$. We show that that our performance is sufficient to make live migration a practical tool even for servers running interactive loads.

In this paper we consider the design options for migrating OSes running services with liveness constraints, focusing on data center and cluster environments. We introduce and analyze the concept of *writable working set*, and present the design, implementation and evaluation of high-performance OS migration built on top of the Xen VMM.

1 Introduction

Operating system virtualization has attracted considerable interest in recent years, particularly from the data center and cluster computing communities. It has previously been shown [1] that paravirtualization allows many OS instances to run concurrently on a single physical machine with high performance, providing better use of physical resources and isolating individual OS instances.

In this paper we explore a further benefit allowed by virtualization: that of live OS migration. Migrating an entire OS and all of its applications as one unit allows us to avoid many of the difficulties faced by process-level migration approaches. In particular the narrow interface between a virtualized OS and the virtual machine monitor (VMM) makes it easy avoid the problem of 'residual dependencies' [2] in which the original host machine must remain available and network-accessible in order to service certain system calls or even memory accesses on behalf of migrated processes. With virtual machine migration, on the other hand, the original host may be decommissioned once migration has completed. This is particularly valuable when migration is occurring in order to allow maintenance of the original host.

Secondly, migrating at the level of an entire virtual machine means that in-memory state can be transferred in a consistent and (as will be shown) efficient fashion. This applies to kernel-internal state (e.g. the TCP control block for a currently active connection) as well as application-level state, even when this is shared between multiple cooperating processes. In practical terms, for example, this means that we can migrate an on-line game server or streaming media server without requiring clients to reconnect: something not possible with approaches which use application-level restart and layer 7 redirection.

Thirdly, live migration of virtual machines allows a separation of concerns between the users and operator of a data center or cluster. Users have 'carte blanche' regarding the software and services they run within their virtual machine, and need not provide the operator with any OS-level access at all (e.g. a root login to quiesce processes or I/O prior to migration). Similarly the operator need not be concerned with the details of what is occurring within the virtual machine; instead they can simply migrate the entire operating system and its attendant processes as a single unit.

Overall, live OS migration is a extremelely powerful tool for cluster administrators, allowing separation of hardware and software considerations, and consolidating clustered hardware into a single coherent management domain. If a physical machine needs to be removed from service an administrator may migrate OS instances including the applications that they are running to alternative machine(s), freeing the original machine for maintenance. Similarly, OS instances may be rearranged across machines in a cluster to relieve load on congested hosts. In these situations the combination of virtualization and migration significantly improves manageability.

We have implemented high-performance migration support for Xen [1], a freely available open source VMM for commodity hardware. Our design and implementation addresses the issues and tradeoffs involved in live local-area migration. Firstly, as we are targeting the migration of active OSes hosting live services, it is critically important to minimize the *downtime* during which services are entirely unavailable. Secondly, we must consider the *total migration time*, during which state on both machines is synchronized and which hence may affect reliability. Furthermore we must ensure that migration does not unnecessarily disrupt active services through resource contention (e.g., CPU, network bandwidth) with the migrating OS.

Our implementation addresses all of these concerns, allowing for example an OS running the SPECweb benchmark to migrate across two physical hosts with only $210ms$ unavailability, or an OS running a Quake 3 server to migrate with just $60ms$ downtime. Unlike application-level restart, we can maintain network connections and application state during this process, hence providing effectively seamless migration from a user's point of view.

We achieve this by using a *pre-copy* approach in which pages of memory are iteratively copied from the source machine to the destination host, all without ever stopping the execution of the virtual machine being migrated. Page-level protection hardware is used to ensure a consistent snapshot is transferred, and a rate-adaptive algorithm is used to control the impact of migration traffic on running services. The final phase pauses the virtual machine, copies any remaining pages to the destination, and resumes execution there. We eschew a 'pull' approach which faults in missing pages across the network since this adds a residual dependency of arbitrarily long duration, as well as providing in general rather poor performance.

Our current implementation does not address migration across the wide area, nor does it include support for migrating local block devices, since neither of these are required for our target problem space. However we discuss ways in which such support can be provided in Section 7.

2 Related Work

The Collective project [3] has previously explored VM migration as a tool to provide mobility to users who work on different physical hosts at different times, citing as an example the transfer of an OS instance to a home computer while a user drives home from work. Their work aims to optimize for slow (e.g., ADSL) links and longer time spans, and so stops OS execution for the duration of the transfer, with a set of enhancements to reduce the transmitted image size. In contrast, our efforts are concerned with the migration of live, in-service OS instances on fast neworks with only tens of milliseconds of downtime. Other projects that have explored migration over longer time spans by stopping and then transferring include Internet Suspend/Resume [4] and μDenali [5].

Zap [6] uses partial OS virtualization to allow the migration of process domains (pods), essentially process groups, using a modified Linux kernel. Their approach is to isolate all process-to-kernel interfaces, such as file handles and sockets, into a contained namespace that can be migrated. Their approach is considerably faster than results in the Collective work, largely due to the smaller units of migration. However, migration in their system is still on the order of seconds at best, and does not allow live migration; pods are entirely suspended, copied, and then resumed. Furthermore, they do not address the problem of maintaining open connections for existing services.

The live migration system presented here has considerable shared heritage with the previous work on NomadBIOS [7], a virtualization and migration system built on top of the L4 microkernel [8]. NomadBIOS uses pre-copy migration to achieve very short best-case migration downtimes, but makes no attempt at adapting to the writable working set behavior of the migrating OS.

VMware has recently added OS migration support, dubbed *VMotion*, to their VirtualCenter management software. As this is commercial software and strictly disallows the publication of third-party benchmarks, we are only able to infer its behavior through VMware's own publications. These limitations make a thorough technical comparison impossible. However, based on the VirtualCenter User's Manual [9], we believe their approach is generally similar to ours and would expect it to perform to a similar standard.

Process migration, a hot topic in systems research during the 1980s [10, 11, 12, 13, 14], has seen very little use for real-world applications. Milojicic *et al* [2] give a thorough survey of possible reasons for this, including the problem of the *residual dependencies* that a migrated process retains on the machine from which it migrated. Examples of residual dependencies include open file descriptors, shared memory segments, and other local resources. These are undesirable because the original machine must remain available, and because they usually negatively impact the performance of migrated processes.

For example Sprite [15] processes executing on foreign nodes require some system calls to be forwarded to the home node for execution, leading to at best reduced performance and at worst widespread failure if the home node is unavailable. Although various efforts were made to ameliorate performance issues, the underlying reliance on the availability of the home node could not be avoided. A similar fragility occurs with MOSIX [14] where a deputy process on the home node must remain available to support remote execution.

We believe the residual dependency problem cannot easily be solved in any process migration scheme – even modern mobile run-times such as Java and .NET suffer from problems when network partition or machine crash causes class loaders to fail. The migration of entire operating systems inherently involves fewer or zero such dependencies, making it more resilient and robust.

3 Design

At a high level we can consider a virtual machine to encapsulate access to a set of physical resources. Providing live migration of these VMs in a clustered server environment leads us to focus on the physical resources used in such environments: specifically on memory, network and disk.

This section summarizes the design decisions that we have made in our approach to live VM migration. We start by describing how memory and then device access is moved across a set of physical hosts and then go on to a high-level description of how a migration progresses.

3.1 Migrating Memory

Moving the contents of a VM's memory from one physical host to another can be approached in any number of ways. However, when a VM is running a live service it is important that this transfer occurs in a manner that balances the requirements of minimizing both *downtime* and *total migration time*. The former is the period during which the service is unavailable due to there being no currently executing instance of the VM; this period will be directly visible to clients of the VM as service interruption. The latter is the duration between when migration is initiated and when the original VM may be finally discarded and, hence, the source host may potentially be taken down for maintenance, upgrade or repair.

It is easiest to consider the trade-offs between these requirements by generalizing memory transfer into three phases:

Push phase The source VM continues running while certain pages are pushed across the network to the new destination. To ensure consistency, pages modified during this process must be re-sent.

Stop-and-copy phase The source VM is stopped, pages are copied across to the destination VM, then the new VM is started.

Pull phase The new VM executes and, if it accesses a page that has not yet been copied, this page is faulted in ("pulled") across the network from the source VM.

Although one can imagine a scheme incorporating all three phases, most practical solutions select one or two of the three. For example, *pure stop-and-copy* [3, 4, 5] involves halting the original VM, copying all pages to the destination, and then starting the new VM. This has advantages in terms of simplicity but means that both downtime and total migration time are proportional to the amount of physical memory allocated to the VM. This can lead to an unacceptable outage if the VM is running a live service.

Another option is *pure demand-migration* [16] in which a short stop-and-copy phase transfers essential kernel data structures to the destination. The destination VM is then started, and other pages are transferred across the network on first use. This results in a much shorter downtime, but produces a much longer total migration time; and in practice, performance after migration is likely to be unacceptably degraded until a considerable set of pages have been faulted across. Until this time the VM will fault on a high proportion of its memory accesses, each of which initiates a synchronous transfer across the network.

The approach taken in this paper, *pre-copy* [11] migration, balances these concerns by combining a bounded iterative push phase with a typically very short stop-and-copy phase. By 'iterative' we mean that pre-copying occurs in *rounds*, in which the pages to be transferred during round n are those that are modified during round $n-1$ (all pages are transferred in the first round). Every VM will have some (hopefully small) set of pages that it updates very frequently and which are therefore poor candidates for pre-copy migration. Hence we bound the number of rounds of pre-copying, based on our analysis of the *writable working set* (WWS) behavior of typical server workloads, which we present in Section 4.

Finally, a crucial additional concern for live migration is the impact on active services. For instance, iteratively scanning and sending a VM's memory image between two hosts in a cluster could easily consume the entire bandwidth available between them and hence starve the active services of resources. This *service degradation* will occur to some extent during any live migration scheme. We address this issue by carefully controlling the network and CPU resources used by the migration process, thereby ensuring that it does not interfere excessively with active traffic or processing.

3.2 Local Resources

A key challenge in managing the migration of OS instances is what to do about resources that are associated with the physical machine that they are migrating away from. While memory can be copied directly to the new host, connections to local devices such as disks and network interfaces demand additional consideration. The two key problems that we have encountered in this space concern what to do with network resources and local storage.

For network resources, we want a migrated OS to maintain all open network connections without relying on forwarding mechanisms on the original host (which may be shut down following migration), or on support from mobility or redirection mechanisms that are not already present (as in [6]). A migrating VM will include all protocol state (e.g. TCP PCBs), and will carry its IP address with it.

To address these requirements we observed that in a cluster environment, the network interfaces of the source and destination machines typically exist on a single switched LAN. Our solution for managing migration with respect to network in this environment is to generate an unsolicited ARP reply from the migrated host, advertising that the IP has moved to a new location. This will reconfigure peers to send packets to the new physical address, and while a very small number of in-flight packets may be lost, the migrated domain will be able to continue using open connections with almost no observable interference.

Some routers are configured not to accept broadcast ARP replies (in order to prevent IP spoofing), so an unsolicited ARP may not work in all scenarios. If the operating system is aware of the migration, it can opt to send directed replies only to interfaces listed in its own ARP cache, to remove the need for a broadcast. Alternatively, on a switched network, the migrating OS can keep its original Ethernet MAC address, relying on the network switch to detect its move to a new port[1].

In the cluster, the migration of storage may be similarly addressed: Most modern data centers consolidate their storage requirements using a network-attached storage (NAS) device, in preference to using local disks in individual servers. NAS has many advantages in this environment, including simple centralised administration, widespread vendor support, and reliance on fewer spindles leading to a reduced failure rate. A further advantage for migration is that it obviates the need to migrate disk storage, as the NAS is uniformly accessible from all host machines in the cluster. We do not address the problem of migrating local-disk storage in this paper, although we suggest some possible strategies as part of our discussion of future work.

3.3 Design Overview

The logical steps that we execute when migrating an OS are summarized in Figure 1. We take a conservative approach to the management of migration with regard to safety and failure handling. Although the consequences of hardware failures can be severe, our basic principle is that safe migration should at no time leave a virtual OS more exposed

[1]Note that on most Ethernet controllers, hardware MAC filtering will have to be disabled if multiple addresses are in use (though some cards support filtering of multiple addresses in hardware) and so this technique is only practical for switched networks.

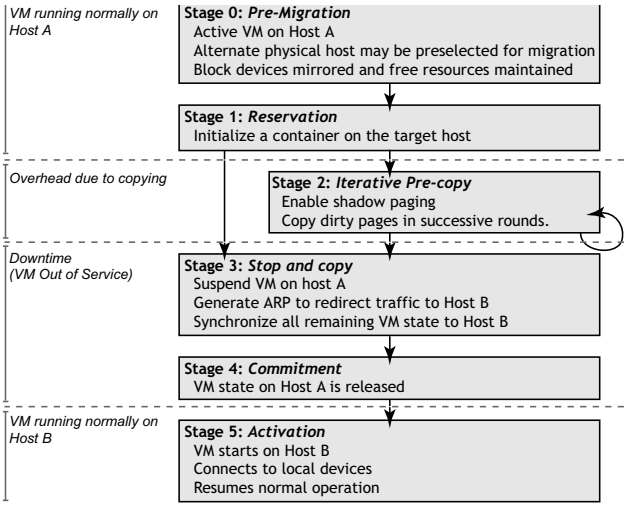

Figure 1: Migration timeline

to system failure than when it is running on the original single host. To achieve this, we view the migration process as a transactional interaction between the two hosts involved:

Stage 0: Pre-Migration We begin with an active VM on physical host A. To speed any future migration, a target host may be preselected where the resources required to receive migration will be guaranteed.

Stage 1: Reservation A request is issued to migrate an OS from host A to host B. We initially confirm that the necessary resources are available on B and reserve a VM container of that size. Failure to secure resources here means that the VM simply continues to run on A unaffected.

Stage 2: Iterative Pre-Copy During the first iteration, all pages are transferred from A to B. Subsequent iterations copy only those pages dirtied during the previous transfer phase.

Stage 3: Stop-and-Copy We suspend the running OS instance at A and redirect its network traffic to B. As described earlier, CPU state and any remaining inconsistent memory pages are then transferred. At the end of this stage there is a consistent suspended copy of the VM at both A and B. The copy at A is still considered to be primary and is resumed in case of failure.

Stage 4: Commitment Host B indicates to A that it has successfully received a consistent OS image. Host A acknowledges this message as commitment of the migration transaction: host A may now discard the original VM, and host B becomes the primary host.

Stage 5: Activation The migrated VM on B is now activated. Post-migration code runs to reattach device drivers to the new machine and advertise moved IP addresses.

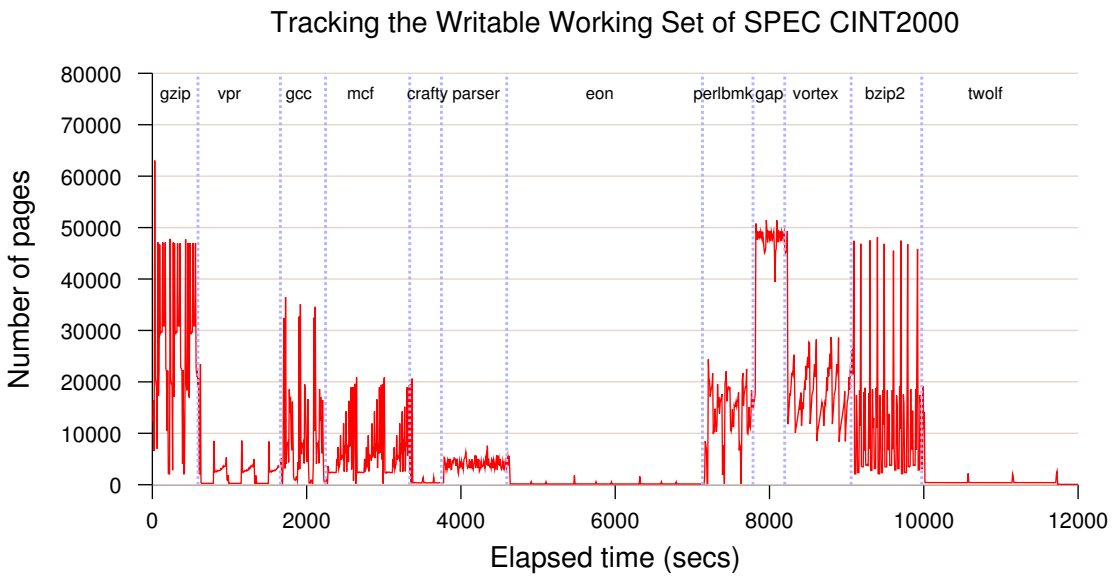

Figure 2: WWS curve for a complete run of SPEC CINT2000 (512MB VM)

This approach to failure management ensures that at least one host has a consistent VM image at all times during migration. It depends on the assumption that the original host remains stable until the migration commits, and that the VM may be suspended and resumed on that host with no risk of failure. Based on these assumptions, a migration request essentially attempts to move the VM to a new host, and on any sort of failure execution is resumed locally, aborting the migration.

4 Writable Working Sets

When migrating a live operating system, the most significant influence on service performance is the overhead of coherently transferring the virtual machine's memory image. As mentioned previously, a simple stop-and-copy approach will achieve this in time proportional to the amount of memory allocated to the VM. Unfortunately, during this time any running services are completely unavailable.

A more attractive alternative is pre-copy migration, in which the memory image is transferred while the operating system (and hence all hosted services) continue to run. The drawback however, is the wasted overhead of transferring memory pages that are subsequently modified, and hence must be transferred again. For many workloads there will be a small set of memory pages that are updated very frequently, and which it is not worth attempting to maintain coherently on the destination machine before stopping and copying the remainder of the VM.

The fundamental question for iterative pre-copy migration is: how does one determine when it is time to stop the pre-copy phase because too much time and resource is being wasted? Clearly if the VM being migrated never modifies memory, a single pre-copy of each memory page will suffice to transfer a consistent image to the destination. However, should the VM continuously dirty pages faster than the rate of copying, then all pre-copy work will be in vain and one should immediately stop and copy.

In practice, one would expect most workloads to lie somewhere between these extremes: a certain (possibly large) set of pages will seldom or never be modified and hence are good candidates for pre-copy, while the remainder will be written often and so should best be transferred via stop-and-copy – we dub this latter set of pages the *writable working set* (WWS) of the operating system by obvious extension of the original working set concept [17].

In this section we analyze the WWS of operating systems running a range of different workloads in an attempt to obtain some insight to allow us build heuristics for an efficient and controllable pre-copy implementation.

4.1 Measuring Writable Working Sets

To trace the writable working set behaviour of a number of representative workloads we used Xen's shadow page tables (see Section 5) to track dirtying statistics on all pages used by a particular executing operating system. This allows us to determine within any time period the set of pages written to by the virtual machine.

Using the above, we conducted a set of experiments to sam-

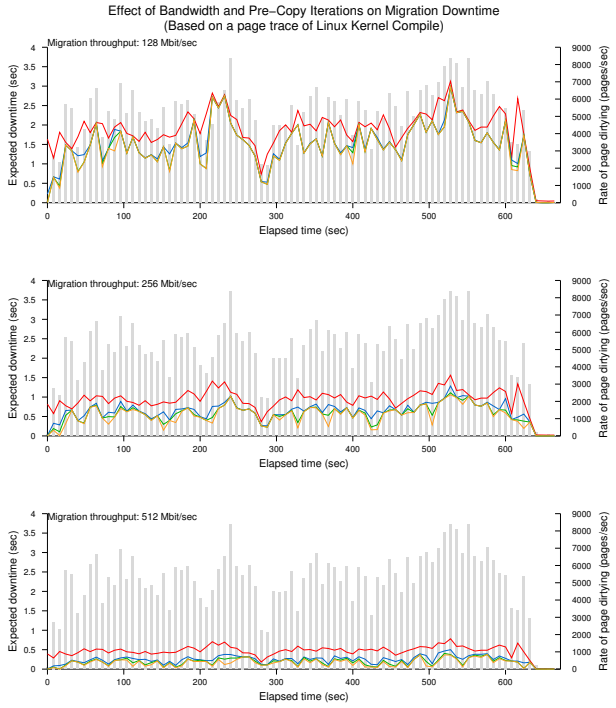

Figure 3: Expected downtime due to last-round memory copy on traced page dirtying of a Linux kernel compile.

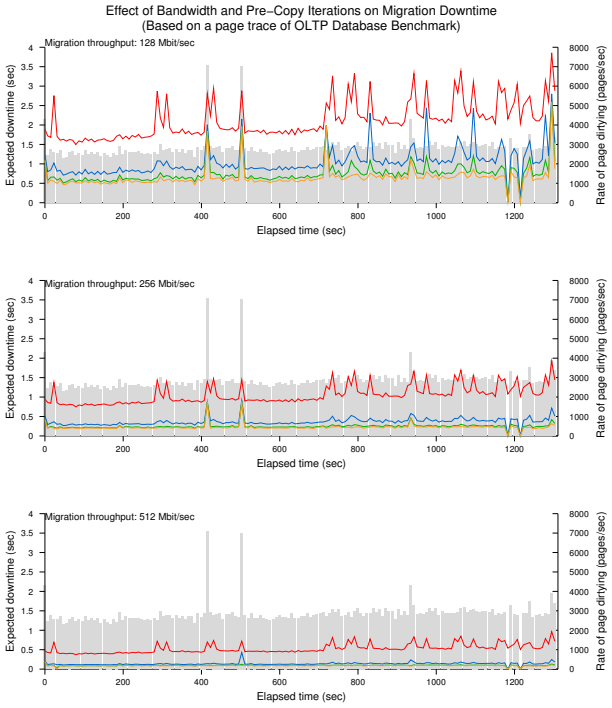

Figure 4: Expected downtime due to last-round memory copy on traced page dirtying of OLTP.

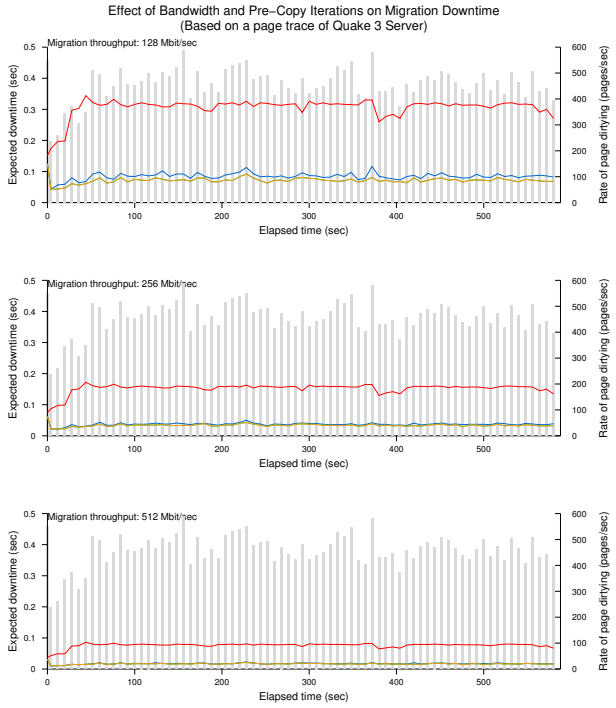

Figure 5: Expected downtime due to last-round memory copy on traced page dirtying of a Quake 3 server.

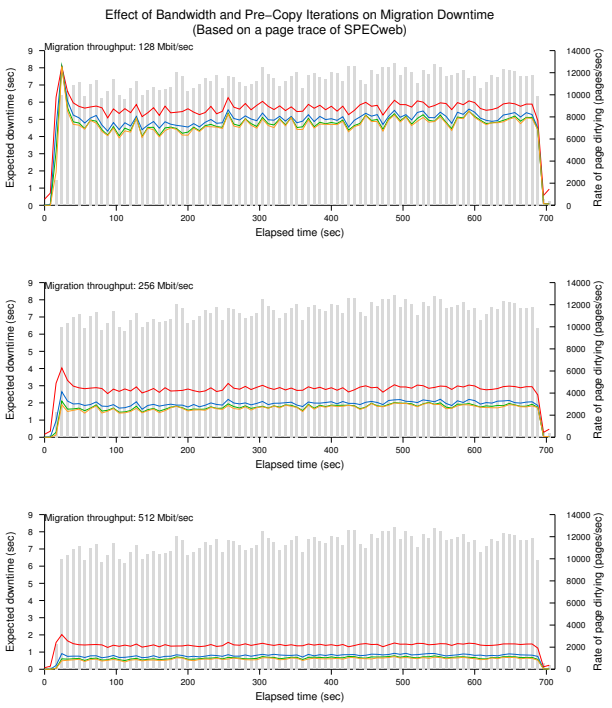

Figure 6: Expected downtime due to last-round memory copy on traced page dirtying of SPECweb.

ple the writable working set size for a variety of benchmarks. Xen was running on a dual processor Intel Xeon 2.4GHz machine, and the virtual machine being measured had a memory allocation of 512MB. In each case we started the relevant benchmark in one virtual machine and read the dirty bitmap every $50ms$ from another virtual machine, cleaning it every 8 seconds – in essence this allows us to compute the WWS with a (relatively long) 8 second window, but estimate it at a finer (50ms) granularity.

The benchmarks we ran were SPEC CINT2000, a Linux kernel compile, the OSDB OLTP benchmark using PostgreSQL and SPECweb99 using Apache. We also measured a Quake 3 server as we are particularly interested in highly interactive workloads.

Figure 2 illustrates the writable working set curve produced for the SPEC CINT2000 benchmark run. This benchmark involves running a series of smaller programs in order and measuring the overall execution time. The x-axis measures elapsed time, and the y-axis shows the number of 4KB pages of memory dirtied within the corresponding 8 second interval; the graph is annotated with the names of the sub-benchmark programs.

From this data we observe that the writable working set varies significantly between the different sub-benchmarks. For programs such as '`eon`' the WWS is a small fraction of the total working set and hence is an excellent candidate for migration. In contrast, '`gap`' has a consistently high dirtying rate and would be problematic to migrate. The other benchmarks go through various phases but are generally amenable to live migration. Thus performing a migration of an operating system will give different results depending on the workload and the precise moment at which migration begins.

4.2 Estimating Migration Effectiveness

We observed that we could use the trace data acquired to estimate the effectiveness of iterative pre-copy migration for various workloads. In particular we can simulate a particular network bandwidth for page transfer, determine how many pages would be dirtied during a particular iteration, and then repeat for successive iterations. Since we know the approximate WWS behaviour at every point in time, we can estimate the overall amount of data transferred in the final stop-and-copy round and hence estimate the downtime.

Figures 3–6 show our results for the four remaining workloads. Each figure comprises three graphs, each of which corresponds to a particular network bandwidth limit for page transfer; each individual graph shows the WWS histogram (in light gray) overlaid with four line plots estimating service downtime for up to four pre-copying rounds.

Looking at the topmost line (one pre-copy iteration), the first thing to observe is that pre-copy migration always performs considerably better than naive stop-and-copy. For a 512MB virtual machine this latter approach would require 32, 16, and 8 seconds downtime for the 128Mbit/sec, 256Mbit/sec and 512Mbit/sec bandwidths respectively. Even in the worst case (the starting phase of SPECweb), a single pre-copy iteration reduces downtime by a factor of four. In most cases we can expect to do considerably better – for example both the Linux kernel compile and the OLTP benchmark typically experience a reduction in downtime of at least a factor of sixteen.

The remaining three lines show, in order, the effect of performing a total of two, three or four pre-copy iterations prior to the final stop-and-copy round. In most cases we see an increased reduction in downtime from performing these additional iterations, although with somewhat diminishing returns, particularly in the higher bandwidth cases.

This is because all the observed workloads exhibit a small but extremely frequently updated set of 'hot' pages. In practice these pages will include the stack and local variables being accessed within the currently executing processes as well as pages being used for network and disk traffic. The hottest pages will be dirtied at least as fast as we can transfer them, and hence must be transferred in the final stop-and-copy phase. This puts a lower bound on the best possible service downtime for a particular benchmark, network bandwidth and migration start time.

This interesting tradeoff suggests that it may be worthwhile increasing the amount of bandwidth used for page transfer in later (and shorter) pre-copy iterations. We will describe our rate-adaptive algorithm based on this observation in Section 5, and demonstrate its effectiveness in Section 6.

5 Implementation Issues

We designed and implemented our pre-copying migration engine to integrate with the Xen virtual machine monitor [1]. Xen securely divides the resources of the host machine amongst a set of resource-isolated virtual machines each running a dedicated OS instance. In addition, there is one special *management virtual machine* used for the administration and control of the machine.

We considered two different methods for initiating and managing state transfer. These illustrate two extreme points in the design space: *managed migration* is performed largely outside the migratee, by a migration daemon running in the management VM; in contrast, *self migration* is implemented almost entirely within the migratee OS with only a small stub required on the destination machine.

In the following sections we describe some of the implementation details of these two approaches. We describe how we use dynamic network rate-limiting to effectively

balance network contention against OS downtime. We then proceed to describe how we ameliorate the effects of rapid page dirtying, and describe some performance enhancements that become possible when the OS is aware of its migration — either through the use of self migration, or by adding explicit paravirtualization interfaces to the VMM.

5.1 Managed Migration

Managed migration is performed by migration daemons running in the management VMs of the source and destination hosts. These are responsible for creating a new VM on the destination machine, and coordinating transfer of live system state over the network.

When transferring the memory image of the still-running OS, the control software performs *rounds* of copying in which it performs a complete scan of the VM's memory pages. Although in the first round all pages are transferred to the destination machine, in subsequent rounds this copying is restricted to pages that were dirtied during the previous round, as indicated by a *dirty bitmap* that is copied from Xen at the start of each round.

During normal operation the page tables managed by each guest OS are the ones that are walked by the processor's MMU to fill the TLB. This is possible because guest OSes are exposed to real physical addresses and so the page tables they create do not need to be mapped to physical addresses by Xen.

To log pages that are dirtied, Xen inserts *shadow page tables* underneath the running OS. The shadow tables are populated on demand by translating sections of the guest page tables. Translation is very simple for dirty logging: all page-table entries (PTEs) are initially read-only mappings in the shadow tables, regardless of what is permitted by the guest tables. If the guest tries to modify a page of memory, the resulting page fault is trapped by Xen. If write access is permitted by the relevant guest PTE then this permission is extended to the shadow PTE. At the same time, we set the appropriate bit in the VM's dirty bitmap.

When the bitmap is copied to the control software at the start of each pre-copying round, Xen's bitmap is cleared and the shadow page tables are destroyed and recreated as the migratee OS continues to run. This causes all write permissions to be lost: all pages that are subsequently updated are then added to the now-clear dirty bitmap.

When it is determined that the pre-copy phase is no longer beneficial, using heuristics derived from the analysis in Section 4, the OS is sent a control message requesting that it suspend itself in a state suitable for migration. This causes the OS to prepare for resumption on the destination machine; Xen informs the control software once the OS has done this. The dirty bitmap is scanned one last time for remaining inconsistent memory pages, and these are transferred to the destination together with the VM's checkpointed CPU-register state.

Once this final information is received at the destination, the VM state on the source machine can safely be discarded. Control software on the destination machine scans the memory map and rewrites the guest's page tables to reflect the addresses of the memory pages that it has been allocated. Execution is then resumed by starting the new VM at the point that the old VM checkpointed itself. The OS then restarts its virtual device drivers and updates its notion of wallclock time.

Since the transfer of pages is OS agnostic, we can easily support any guest operating system – all that is required is a small paravirtualized stub to handle resumption. Our implementation currently supports Linux 2.4, Linux 2.6 and NetBSD 2.0.

5.2 Self Migration

In contrast to the managed method described above, self migration [18] places the majority of the implementation within the OS being migrated. In this design no modifications are required either to Xen or to the management software running on the source machine, although a migration stub must run on the destination machine to listen for incoming migration requests, create an appropriate empty VM, and receive the migrated system state.

The pre-copying scheme that we implemented for self migration is conceptually very similar to that for managed migration. At the start of each pre-copying round every page mapping in every virtual address space is write-protected. The OS maintains a dirty bitmap tracking dirtied physical pages, setting the appropriate bits as write faults occur. To discriminate migration faults from other possible causes (for example, copy-on-write faults, or access-permission faults) we reserve a spare bit in each PTE to indicate that it is write-protected only for dirty-logging purposes.

The major implementation difficulty of this scheme is to transfer a consistent OS checkpoint. In contrast with a managed migration, where we simply suspend the migratee to obtain a consistent checkpoint, self migration is far harder because the OS must continue to run in order to transfer its final state. We solve this difficulty by logically checkpointing the OS on entry to a final *two-stage* stop-and-copy phase. The first stage disables all OS activity except for migration and then peforms a final scan of the dirty bitmap, clearing the appropriate bit as each page is transferred. Any pages that are dirtied during the final scan, and that are still marked as dirty in the bitmap, are copied to a shadow buffer. The second and final stage then transfers the contents of the shadow buffer — page updates are ignored during this transfer.

5.3 Dynamic Rate-Limiting

It is not always appropriate to select a single network bandwidth limit for migration traffic. Although a low limit avoids impacting the performance of running services, analysis in Section 4 showed that we must eventually pay in the form of an extended downtime because the hottest pages in the writable working set are not amenable to pre-copy migration. The downtime can be reduced by increasing the bandwidth limit, albeit at the cost of additional network contention.

Our solution to this impasse is to dynamically adapt the bandwidth limit during each pre-copying round. The administrator selects a minimum and a maximum bandwidth limit. The first pre-copy round transfers pages at the minimum bandwidth. Each subsequent round counts the number of pages dirtied in the previous round, and divides this by the duration of the previous round to calculate the *dirtying rate*. The bandwidth limit for the next round is then determined by adding a constant increment to the previous round's dirtying rate — we have empirically determined that 50Mbit/sec is a suitable value. We terminate pre-copying when the calculated rate is greater than the administrator's chosen maximum, or when less than 256KB remains to be transferred. During the final stop-and-copy phase we minimize service downtime by transferring memory at the maximum allowable rate.

As we will show in Section 6, using this adaptive scheme results in the bandwidth usage remaining low during the transfer of the majority of the pages, increasing only at the end of the migration to transfer the hottest pages in the WWS. This effectively balances short downtime with low average network contention and CPU usage.

5.4 Rapid Page Dirtying

Our working-set analysis in Section 4 shows that every OS workload has some set of pages that are updated extremely frequently, and which are therefore not good candidates for pre-copy migration even when using all available network bandwidth. We observed that rapidly-modified pages are very likely to be dirtied again by the time we attempt to transfer them in any particular pre-copying round. We therefore periodically 'peek' at the current round's dirty bitmap and transfer only those pages dirtied in the previous round that have not been dirtied again at the time we scan them.

We further observed that page dirtying is often physically *clustered* — if a page is dirtied then it is disproportionally likely that a close neighbour will be dirtied soon after. This increases the likelihood that, if our peeking does not detect one page in a cluster, it will detect none. To avoid this unfortunate behaviour we scan the VM's physical memory space in a pseudo-random order.

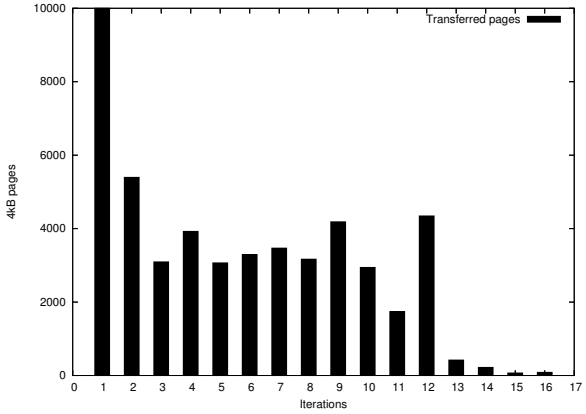

Figure 7: Rogue-process detection during migration of a Linux kernel build. After the twelfth iteration a maximum limit of forty write faults is imposed on every process, drastically reducing the total writable working set.

5.5 Paravirtualized Optimizations

One key benefit of paravirtualization is that operating systems can be made aware of certain important differences between the real and virtual environments. In terms of migration, this allows a number of optimizations by informing the operating system that it is about to be migrated – at this stage a migration stub handler within the OS could help improve performance in at least the following ways:

Stunning Rogue Processes. Pre-copy migration works best when memory pages can be copied to the destination host faster than they are dirtied by the migrating virtual machine. This may not always be the case – for example, a test program which writes one word in every page was able to dirty memory at a rate of 320 Gbit/sec, well ahead of the transfer rate of any Ethernet interface. This is a synthetic example, but there may well be cases in practice in which pre-copy migration is unable to keep up, or where migration is prolonged unnecessarily by one or more 'rogue' applications.

In both the managed and self migration cases, we can mitigate against this risk by forking a monitoring thread within the OS kernel when migration begins. As it runs within the OS, this thread can monitor the WWS of *individual processes* and take action if required. We have implemented a simple version of this which simply limits each process to 40 write faults before being moved to a wait queue – in essence we 'stun' processes that make migration difficult. This technique works well, as shown in Figure 7, although

one must be careful not to stun important interactive services.

Freeing Page Cache Pages. A typical operating system will have a number of 'free' pages at any time, ranging from truly free (page allocator) to cold buffer cache pages. When informed a migration is to begin, the OS can simply return some or all of these pages to Xen in the same way it would when using the ballooning mechanism described in [1]. This means that the time taken for the first "full pass" iteration of pre-copy migration can be reduced, sometimes drastically. However should the contents of these pages be needed again, they will need to be faulted back in from disk, incurring greater overall cost.

6 Evaluation

In this section we present a thorough evaluation of our implementation on a wide variety of workloads. We begin by describing our test setup, and then go on explore the migration of several workloads in detail. Note that none of the experiments in this section use the paravirtualized optimizations discussed above since we wished to measure the baseline performance of our system.

6.1 Test Setup

We perform test migrations between an identical pair of Dell PE-2650 server-class machines, each with dual Xeon 2GHz CPUs and 2GB memory. The machines have Broadcom TG3 network interfaces and are connected via switched Gigabit Ethernet. In these experiments only a single CPU was used, with HyperThreading enabled. Storage is accessed via the iSCSI protocol from an NetApp F840 network attached storage server except where noted otherwise. We used XenLinux 2.4.27 as the operating system in all cases.

6.2 Simple Web Server

We begin our evaluation by examining the migration of an Apache 1.3 web server serving static content at a high rate. Figure 8 illustrates the throughput achieved when continuously serving a single 512KB file to a set of one hundred concurrent clients. The web server virtual machine has a memory allocation of 800MB.

At the start of the trace, the server achieves a consistent throughput of approximately 870Mbit/sec. Migration starts twenty seven seconds into the trace but is initially rate-limited to 100Mbit/sec (12% CPU), resulting in the server throughput dropping to 765Mbit/s. This initial low-rate pass transfers 776MB and lasts for 62 seconds, at which point the migration algorithm described in Section 5 increases its rate over several iterations and finally suspends the VM after a further 9.8 seconds. The final stop-and-copy phase then transfer the remaining pages and the web server resumes at full rate after a 165ms outage.

This simple example demonstrates that a highly loaded server can be migrated with both controlled impact on live services and a short downtime. However, the working set of the server in this case is rather small, and so this should be expected to be a relatively easy case for live migration.

6.3 Complex Web Workload: SPECweb99

A more challenging Apache workload is presented by SPECweb99, a complex application-level benchmark for evaluating web servers and the systems that host them. The workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations, and 0.5% execute a CGI script. As the server runs, it generates access and POST logs, contributing to disk (and therefore network) throughput.

A number of client machines are used to generate the load for the server under test, with each machine simulating a collection of users concurrently accessing the web site. SPECweb99 defines a minimum quality of service that each user must receive for it to count as 'conformant'; an aggregate bandwidth in excess of 320Kbit/sec over a series of requests. The SPECweb score received is the number of conformant users that the server successfully maintains. The considerably more demanding workload of SPECweb represents a challenging candidate for migration.

We benchmarked a single VM running SPECweb and recorded a maximum score of 385 conformant clients — we used the RedHat `gnbd` network block device in place of iSCSI as the lighter-weight protocol achieves higher performance. Since at this point the server is effectively in overload, we then relaxed the offered load to 90% of maximum (350 conformant connections) to represent a more realistic scenario.

Using a virtual machine configured with 800MB of memory, we migrated a SPECweb99 run in the middle of its execution. Figure 9 shows a detailed analysis of this migration. The x-axis shows time elapsed since start of migration, while the y-axis shows the network bandwidth being used to transfer pages to the destination. Darker boxes illustrate the page transfer process while lighter boxes show the pages dirtied during each iteration. Our algorithm adjusts the transfer rate relative to the page dirty rate observed during the previous round (denoted by the height of the lighter boxes).

As in the case of the static web server, migration begins

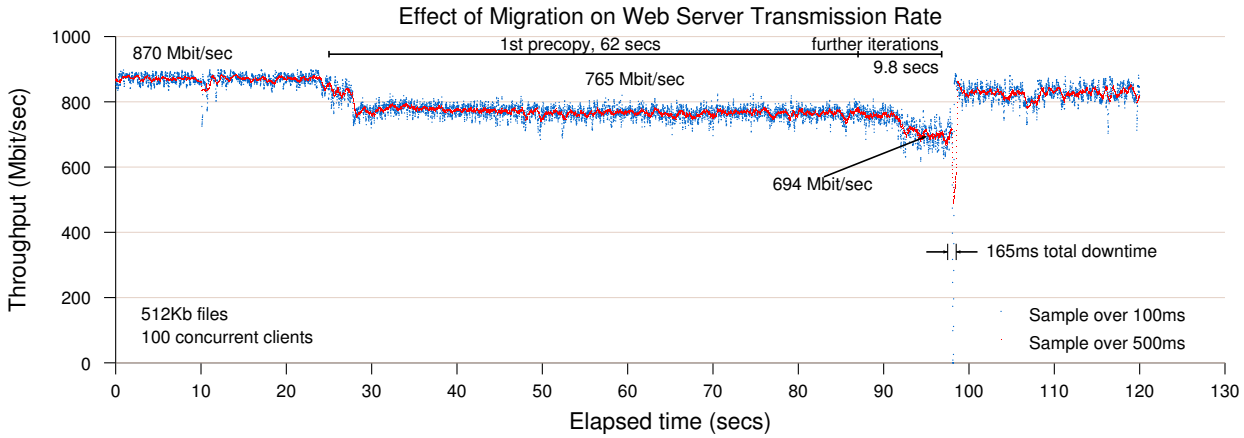

Figure 8: Results of migrating a running web server VM.

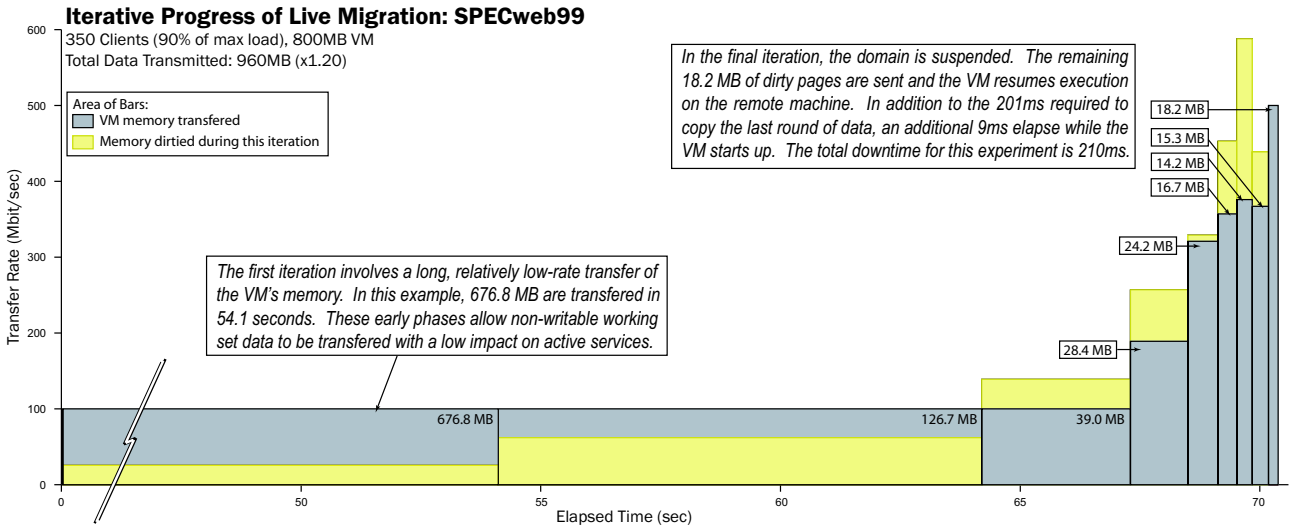

Figure 9: Results of migrating a running SPECweb VM.

with a long period of low-rate transmission as a first pass is made through the memory of the virtual machine. This first round takes 54.1 seconds and transmits 676.8MB of memory. Two more low-rate rounds follow, transmitting 126.7MB and 39.0MB respectively before the transmission rate is increased.

The remainder of the graph illustrates how the adaptive algorithm tracks the page dirty rate over successively shorter iterations before finally suspending the VM. When suspension takes place, 18.2MB of memory remains to be sent. This transmission takes $201ms$, after which an additional $9ms$ is required for the domain to resume normal execution.

The total downtime of $210ms$ experienced by the SPECweb clients is sufficiently brief to maintain the 350 conformant clients. This result is an excellent validation of our approach: a heavily (90% of maximum) loaded server is migrated to a separate physical host with a total migration time of seventy-one seconds. Furthermore the migration does not interfere with the quality of service demanded by SPECweb's workload. This illustrates the applicability of migration as a tool for administrators of demanding live services.

6.4 Low-Latency Server: Quake 3

Another representative application for hosting environments is a multiplayer on-line game server. To determine the effectiveness of our approach in this case we configured a virtual machine with 64MB of memory running a

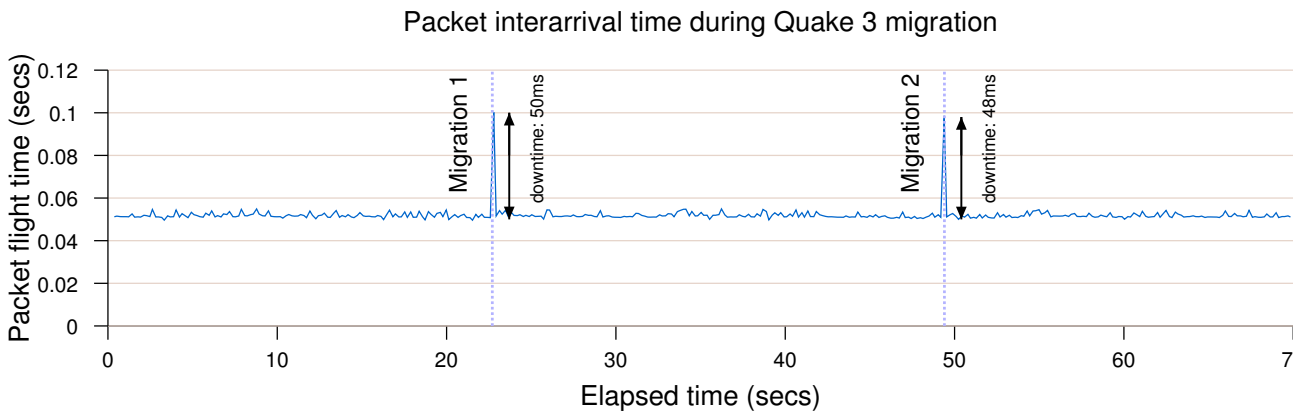

Figure 10: Effect on packet response time of migrating a running Quake 3 server VM.

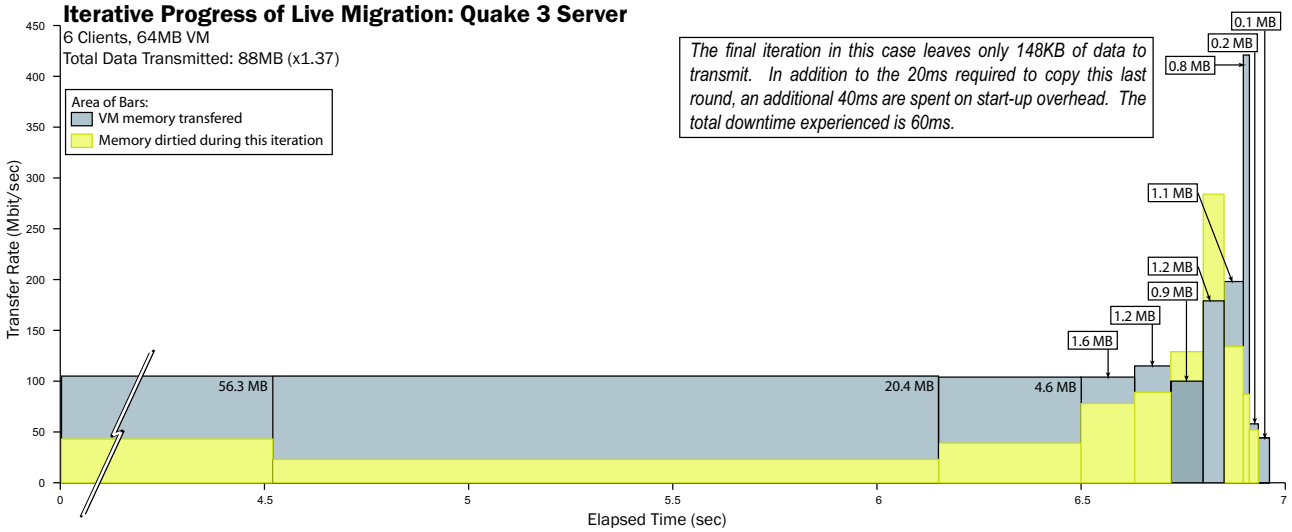

Figure 11: Results of migrating a running Quake 3 server VM.

Quake 3 server. Six players joined the game and started to play within a shared arena, at which point we initiated a migration to another machine. A detailed analysis of this migration is shown in Figure 11.

The trace illustrates a generally similar progression as for SPECweb, although in this case the amount of data to be transferred is significantly smaller. Once again the transfer rate increases as the trace progresses, although the final stop-and-copy phase transfers so little data (148KB) that the full bandwidth is not utilized.

Overall, we are able to perform the live migration with a total downtime of $60ms$. To determine the effect of migration on the live players, we performed an additional experiment in which we migrated the running Quake 3 server twice and measured the inter-arrival time of packets received by clients. The results are shown in Figure 10. As can be seen, from the client point of view migration manifests itself as a transient increase in response time of $50ms$. In neither case was this perceptible to the players.

6.5 A Diabolical Workload: MMuncher

As a final point in our evaluation, we consider the situation in which a virtual machine is writing to memory faster than can be transferred across the network. We test this diabolical case by running a 512MB host with a simple C program that writes constantly to a 256MB region of memory. The results of this migration are shown in Figure 12.

In the first iteration of this workload, we see that half of the memory has been transmitted, while the other half is immediately marked dirty by our test program. Our algorithm attempts to adapt to this by scaling itself relative to the perceived initial rate of dirtying; this scaling proves in-

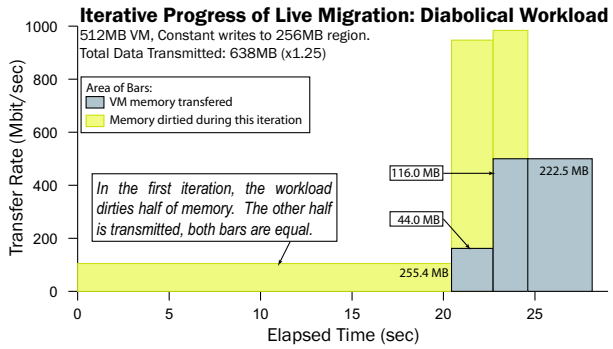

Figure 12: Results of migrating a VM running a diabolical workload.

sufficient, as the rate at which the memory is being written becomes apparent. In the third round, the transfer rate is scaled up to 500Mbit/s in a final attempt to outpace the memory writer. As this last attempt is still unsuccessful, the virtual machine is suspended, and the remaining dirty pages are copied, resulting in a downtime of 3.5 seconds. Fortunately such dirtying rates appear to be rare in real workloads.

7 Future Work

Although our solution is well-suited for the environment we have targeted – a well-connected data-center or cluster with network-accessed storage – there are a number of areas in which we hope to carry out future work. This would allow us to extend live migration to wide-area networks, and to environments that cannot rely solely on network-attached storage.

7.1 Cluster Management

In a cluster environment where a pool of virtual machines are hosted on a smaller set of physical servers, there are great opportunities for dynamic load balancing of processor, memory and networking resources. A key challenge is to develop cluster control software which can make informed decision as to the placement and movement of virtual machines.

A special case of this is 'evacuating' VMs from a node that is to be taken down for scheduled maintenance. A sensible approach to achieving this is to migrate the VMs in increasing order of their observed WWS. Since each VM migrated frees resources on the node, additional CPU and network becomes available for those VMs which need it most. We are in the process of building a cluster controller for Xen systems.

7.2 Wide Area Network Redirection

Our layer 2 redirection scheme works efficiently and with remarkably low outage on modern gigabit networks. However, when migrating outside the local subnet this mechanism will not suffice. Instead, either the OS will have to obtain a new IP address which is within the destination subnet, or some kind of indirection layer, on top of IP, must exist. Since this problem is already familiar to laptop users, a number of different solutions have been suggested. One of the more prominent approaches is that of Mobile IP [19] where a node on the home network (the *home agent*) forwards packets destined for the client (*mobile node*) to a *care-of address* on the foreign network. As with all residual dependencies this can lead to both performance problems and additional failure modes.

Snoeren and Balakrishnan [20] suggest addressing the problem of connection migration at the TCP level, augmenting TCP with a secure token negotiated at connection time, to which a relocated host can refer in a special SYN packet requesting reconnection from a new IP address. Dynamic DNS updates are suggested as a means of locating hosts after a move.

7.3 Migrating Block Devices

Although NAS prevails in the modern data center, some environments may still make extensive use of local disks. These present a significant problem for migration as they are usually considerably larger than volatile memory. If the entire contents of a disk must be transferred to a new host before migration can complete, then total migration times may be intolerably extended.

This latency can be avoided at migration time by arranging to *mirror* the disk contents at one or more remote hosts. For example, we are investigating using the built-in software RAID and iSCSI functionality of Linux to implement disk mirroring before and during OS migration. We imagine a similar use of software RAID-5, in cases where data on disks requires a higher level of availability. Multiple hosts can act as storage targets for one another, increasing availability at the cost of some network traffic.

The effective management of local storage for clusters of virtual machines is an interesting problem that we hope to further explore in future work. As virtual machines will typically work from a small set of common system images (for instance a generic Fedora Linux installation) and make individual changes above this, there seems to be opportunity to manage copy-on-write system images across a cluster in a way that facilitates migration, allows replication, and makes efficient use of local disks.

8 Conclusion

By integrating live OS migration into the Xen virtual machine monitor we enable rapid movement of interactive workloads within clusters and data centers. Our dynamic network-bandwidth adaptation allows migration to proceed with minimal impact on running services, while reducing total downtime to below discernable thresholds.

Our comprehensive evaluation shows that realistic server workloads such as SPECweb99 can be migrated with just $210ms$ downtime, while a Quake3 game server is migrated with an imperceptible $60ms$ outage.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[2] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[3] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M.Rosenblum. Optimizing the migration of virtual computers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.

[4] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 2002.

[5] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[6] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI-02)*, pages 361–376, December 2002.

[7] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002.

[8] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of micro-kernel-based systems. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 66–77. ACM Press, 1997.

[9] VMWare, Inc. *VMWare VirtualCenter Version 1.2 User's Manual*. 2004.

[10] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the ninth ACM Symposium on Operating System Principles*, pages 110–119. ACM Press, 1983.

[11] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the tenth ACM Symposium on Operating System Principles*, pages 2–12. ACM Press, 1985.

[12] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

[13] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[14] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[15] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

[16] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24. ACM Press, 1987.

[17] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[18] Jacob G. Hansen and Eric Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW 2004)*, pages 126–130, 2004.

[19] C. E. Perkins and A. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, pages 415–419, 1997.

[20] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 155–166. ACM Press, 2000.