Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning

Michael R. Hines and Kartik Gopalan
Computer Science, Binghamton University (State University of New York)
{mhines,kartik}@cs.binghamton.edu

Abstract

We present the design, implementation, and evaluation of *post-copy* based live migration for virtual machines (VMs) across a Gigabit LAN. Live migration is an indispensable feature in today's virtualization technologies. Post-copy migration defers the transfer of a VM's memory contents until after its processor state has been sent to the target host. This deferral is in contrast to the traditional *pre-copy* approach, which first copies the memory state over multiple iterations followed by a final transfer of the processor state. The post-copy strategy can provide a "win-win" by reducing total migration time closer to its equivalent time achieved by non-live VM migration. This is done while maintaining the liveness benefits of the pre-copy approach. We compare post-copy extensively against the traditional pre-copy approach on top of the Xen Hypervisor. Using a range of VM workloads we show improvements in several migration metrics including pages transferred, total migration time and network overhead. We facilitate the use of post-copy with adaptive *pre-paging* in order to eliminate all duplicate page transmissions. Our implementation is able to reduce the number of network-bound page faults to within 21% of the VM's working set for large workloads. Finally, we eliminate the transfer of free memory pages in both migration schemes through a *dynamic self-ballooning* (DSB) mechanism. DSB periodically releases free pages in a guest VM back to the hypervisor and significantly speeds up migration with negligible performance degradation.

Categories and Subject Descriptors D.4 [Operating Systems]

General Terms Experimentation, Performance

Keywords Virtual Machines, Operating Systems, Process Migration, Post-Copy, Xen

1. Introduction

This paper addresses the problem of optimizing the live migration of system virtual machines (VMs). Live migration is a key selling point for state-of-the-art virtualization technologies. It allows administrators to consolidate system load, perform maintenance, and flexibly reallocate cluster-wide resources on-the-fly. We focus on VM migration within a cluster environment where physical nodes are interconnected via a high-speed LAN and also employ a network-accessible storage system (such as a SAN or NAS). State-of-the-art live migration techniques (18; 3) use the *pre-copy* approach, which works as follows. The bulk of the VM's memory state is migrated to a target node even as the VM continues to execute at a source node. If a transmitted page is dirtied, it is re-sent to the target in the next round. This iterative copying of dirtied pages continues until either a small, writable working set (WWS) has been identified, or a preset number of iterations is reached, whichever comes first. This constitutes the end of the memory transfer phase and the beginning of *service downtime*. The VM is then suspended and its processor state plus any remaining dirty pages are sent to a target node. Finally, the VM is restarted and the copy at source is destroyed.

Pre-copy's overriding goal is to keep downtime small by minimizing the amount of VM state that needs to be transferred during downtime. Pre-copy will cap the number of copying iterations to a preset limit since the WWS is not guaranteed to converge across successive iterations. On the other hand, if the iterations are terminated too early, then the larger WWS will significantly increase service downtime. This approach minimizes two metrics particularly well – VM downtime and application degradation – when the VM is executing a largely read-intensive workload. However, even moderately write-intensive workloads can reduce pre-copy's effectiveness during migration.

In this paper, we implement and evaluate another strategy for live VM migration, called *post-copy*, previously studied only in the context of process migration. On a high-level, post-copy migration defers the memory transfer phase until *after* the VM's CPU state has already been transferred to the target and resumed there. Post-copy first transmits all processor state to the target, starts the VM at the target, and then actively pushes memory pages from source to target. Concurrently, any memory pages that are faulted on by the VM at target, and not yet pushed, are demand-paged over the network from source. Post-copy thus ensures that each memory page is transferred *at most once*, thus avoiding the duplicate transmission overhead of pre-copy.

We also supplement active push with *adaptive pre-paging*. Pre-paging is a term from earlier literature (21; 31) on optimizing memory-constrained disk-based paging systems. It traditionally refers to a more proactive form of pre-fetching from disk in which the memory subsystem can try to hide the latency of high-locality page faults or cache misses by intelligently sequencing the pre-fetched pages. Modern virtual memory subsystems do not typically employ pre-paging due increasing DRAM capacities. However, pre-paging can still play a critical role in post-copy VM migration by improving the effectiveness of the active push component. Pre-paging adapts the sequence of actively pushed pages by treating network page faults (or major faults) as hints to guess the VM's page access locality at the target. Pre-paging thus increases the likelihood that pages in the neighborhood of a major fault will be available at the target before they are accessed by the VM. We show that our implementation can reduce major network faults to within 21% of the VM's working set for large workloads.

Additionally, we identified deficiencies in both migration techniques with regard to the *handling of free pages* during migration. To avoid transmitting the free pages, we employ a "dynamic self-

VEE'09, March 11–13, 2009, Washington, DC, USA.

ballooning" (DSB) mechanism. Ballooning is an existing technique that allows a guest OS to reduce its memory footprint by releasing its free memory pages back to the hypervisor. DSB automates the ballooning mechanism so it can trigger frequently (every 5 seconds) without degrading application performance. Our DSB implementation responds directly to OS memory allocation requests without the need for kernel patching. It neither requires external introspection by a co-located VM nor extra communication through the hypervisor. DSB thus significantly reduces total migration time by eliminating the transfer of free memory pages.

The original pre-copy algorithm has additional advantages: it employs a relatively self-contained implementation that allows the migration daemon to isolate most of the copying complexity to a single process at each node. Also, pre-copy provides a clean method of *aborting the migration* should the target node ever crash during migration because the VM is still running at the source. Our current post-copy implementation does not handle these kinds of failures. However, we discuss a straightforward approach in Section 3.4 by which post-copy could provide the same level of reliability as pre-copy.

We designed and implemented a prototype of our approach in the Xen VM environment. Through extensive evaluations, we demonstrate situations in which post-copy can significantly improve migration performance in terms of pages transferred and total migration time. Finally, note that post-copy and pre-copy together enhance the toolbox of VM migration techniques available to a cluster administrator. Depending upon the VM workload type and performance goals of migration, an administrator has the flexibility to choose either of the techniques. For interactive VMs on which many users might depend, pre-copy would be the better approach because processor does not wait for major faults. But for large-memory or write-intensive server workloads, post-copy would better suited. Our contributions are to demonstrate that post-copy is a practical VM migration technique and to evaluate its relative merits against pre-copy.

2. Related Work

Process Migration: The post-copy technique has been variously studied in the context of process migration literature: first implemented as "Freeze Free" using a file-server (25), then evaluated via simulations (24), and later via actual Linux implementation (19). There was also a recent implementation of post-copy process migration under openMosix (11). In contrast, our contributions are to develop a viable post-copy technique for live migration of virtual machines. We also evaluate our implementation against an array of application workloads during live VM migration, which the above approaches do not. Process migration techniques have been extensively researched and an excellent survey can be found in (16). Several distributed computing projects incorporate process migration (30; 23; 17; 29; 12; 6). However, these systems have not gained widespread acceptance primarily because of portability and residual dependency limitations. In contrast, VM migration operates on whole operating systems and is naturally free of these problems.

Pre-Paging: Pre-paging is a technique for hiding the latency of page faults (and in general I/O accesses in the critical execution path) by predicting the future working set (5) and loading the required pages before they are accessed. Pre-paging is also known as adaptive prefetching or adaptive remote paging. It has been studied extensively (21; 32; 33; 31) in the context of disk based storage systems, since disk I/O accesses in the critical application execution path can be highly expensive. Traditional pre-paging algorithms use reactive and history based approaches to predict and prefetch the working set of the application. Our system employs pre-paging for the limited duration of live migration to avoid the latency network page faults from the target to the source node. Our implementation employs a reactive approach that uses any network faults as hints about the guest VM's working set with additional optimizations described in Section 3.1 and may also be augmented with history-based approaches.

Live VM Migration. Pre-copy is the predominant approach for live VM migration. These include hypervisor-based approaches from VMware (18), Xen (3), and KVM (13), OS-level approaches that do not use hypervisors from OpenVZ (20), as well as wide-area migration (2). Furthermore, the self-migration of operating systems (which has much in common with process migration) was implemented in (9) building upon prior work (8) atop the L4 Linux microkernel. All of the above systems currently use pre-copy based migration and can potentially benefit from the approach in this paper. The closest work to our technique is SnowFlock (14). This work sets up impromptu clusters to support highly parallel computation tasks across VMs by cloning the source VM on the fly. This is optimized by actively pushing cloned memory via multicast from the source VM. They do not target VM migration in particular, nor present a comprehensive comparison (or optimize upon) the original pre-copy approach. Further, we use transparent ballooning to eliminate free memory transmission whereas SnowFlock requires kernel modifications to do the same.

Non-Live VM Migration. There are several *non-live* approaches to VM migration. Schmidt (28) proposed using capsules, which are groups of related processes along with their IPC/network state, as migration units. Similarly, Zap (22) uses process groups (pods) along with their kernel state as migration units. The Denali project (36; 35) addressed migration of checkpointed VMs. Work in (26) addressed user mobility and system administration by encapsulating the computing environment as capsules to be transferred between distinct hosts. Internet suspend/resume (27) focuses on saving/restoring computing state on anonymous hardware. In all the above systems, the execution of the VM is suspended, during which applications do not make progress.

Dynamic Self-Ballooning (DSB): Ballooning refers to artificially requesting memory within a guest OS and releasing that memory back to the hypervisor. Ballooning is used widely for the purpose of VM memory resizing by both VMWare (34) and Xen (1), and relates to self-paging in Nemesis (7). However, it is not clear how current ballooning mechanisms interact, if at all, with live VM migration techniques. For instance, while Xen is capable of simple one-time ballooning during migration and system boot time, there is no explicit use of dynamic ballooning to reduce the memory footprint before live migration. Additionally, self-ballooning has been recently committed into the Xen source tree (15) by Oracle Corp to enable a guest OS to dynamically return free memory to the hypervisor without explicit human intervention. VMWare ESX server (34) includes dynamic ballooning and idle memory tax, but the focus is not on reducing the VM footprint before migration. Our DSB mechanism is similar in spirit to the above dynamic ballooning approaches. However, to the best of our knowledge, DSB has not been exploited systematically to date for improving the performance of live migration. Our work uses DSB to improve the migration performance of both the pre-copy and post-copy approaches with minimal runtime overhead.

3. Design of Post-Copy Live VM Migration

In this section we present the architectural overview of post-copy live VM migration. The performance of any live VM migration strategy could be gauged by the following metrics.

1. **Preparation Time**: This is the time between initiating migration and transferring the VM's processor state to the target node, during which the VM continues to execute and dirty its mem-

ory. For pre-copy, this time includes the entire iterative memory copying phase, whereas it is negligible for post-copy.

2. **Downtime**: This is time during which the migrating VM's execution is stopped. At the minimum this includes the transfer of processor state. For pre-copy, this transfer also includes any remaining dirty pages. For post-copy this includes other minimal execution state, if any, needed by the VM to start at the target.
3. **Resume Time**: This is the time between resuming the VM's execution at the target and the end of migration altogether, at which point all dependencies on the source must be eliminated. For pre-copy, one needs only to re-schedule the target VM and destroy the source copy. On the other hand, majority of our post-copy approach operates in this period.
4. **Pages Transferred**: This is the total count of memory pages transferred, including duplicates, across all of the above time periods. Pre-copy transfers most of its pages during preparation time, whereas post-copy transfers most during resume time.
5. **Total Migration Time**: This is the sum of all the above times from start to finish. Total time is important because it affects the release of resources on both participating nodes as well as within the VMs on both nodes. Until the completion of migration, we cannot free the source VM's memory.
6. **Application Degradation**: This is the extent to which migration slows down the applications executing within the VM. Pre-copy must track dirtied pages by trapping write accesses to each page, which significantly slows down write-intensive workloads. Similarly, post-copy requires the servicing of major network faults generated at the target, which also slows down VM workloads.

3.1 Post-Copy and its Variants

In the basic approach, post-copy first suspends the migrating VM at the source node, copies minimal processor state to the target node, resumes the virtual machine, and begins fetching memory pages over the network from the source. The manner in which pages are fetched gives rise to different variants of post-copy, each of which provides incremental improvements. We employ a combination of four techniques to fetch memory pages from the source: *demand-paging, active push, pre-paging, and dynamic self-ballooning (DSB)*. Demand paging ensures that each page is sent over the network only once, unlike in pre-copy where repeatedly dirtied pages could be resent multiple times. Similarly, active push ensures that residual dependencies are removed from the source host as quickly as possible, compared to the non-deterministic copying iterations in pre-copy. Pre-paging uses hints from the VM's page access patterns to reduce both the number of major network faults and the duration of the resume phase. DSB reduces the number of free pages transferred during migration, improving the performance of both pre-copy and post-copy. Figure 1 provides a high-level contrast of how different stages of pre-copy and post-copy relate to each other. Table 1 contrasts the different migration techniques, each of which is described below in greater detail.

Post-Copy via Demand Paging: The demand paging variant of post-copy is the simplest and slowest option. Once the VM resumes at the target, its memory accesses result in page faults that can be serviced by requesting the referenced page over the network from the source node. However, servicing each fault will significantly slow down the VM due to the network's round trip latency. Consequently, even though each page is transferred only once, this approach considerably lengthens the resume time and leaves long-term residual dependencies in the form of unfetched pages, possibly for an indeterminate duration. Thus, post-copy performance for this variant by itself would be unacceptable both from the viewpoint of total migration time and application degradation.

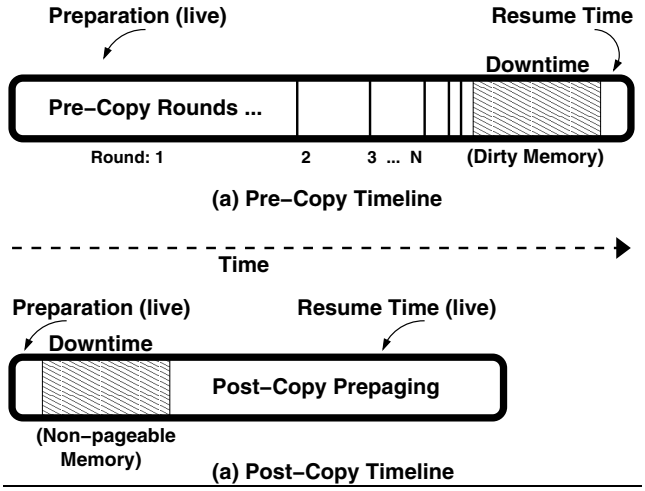

Figure 1. The timeline of (a) Pre-copy vs. (b) Post-copy migration.

	Preparation	Downtime	Resume
1. Pre-copy Only	Iterative mem txfer	CPU + dirty mem txfer	Reschedule VM
2. Demand-Paging	Prep time (if any)	CPU + minimal state	Net. page faults only
3. Pushing + Demand	Prep time (if any)	CPU + minimal state	Active push +page faults
4. Pre-paging + Demand	Prep time (if any)	CPU + minimal state	Bubbling + page faults
5. Hybrid (all)	Single copy round	CPU + minimal state	Bubbling + dirty faults

Table 1. Design choices for live VM migration, in order of their incremental improvements. Method 4 combines methods 2 and 3 with the use of pre-paging. Method 5 combines all of 1 through 4, in which pre-copy only performs a single primer copy round.

Post-Copy via Active Pushing: One way to reduce the duration of residual dependencies on the source node is to proactively "push" the VM's pages from the source to the target even as the VM continues executing at the target. Any major faults incurred by the VM can be serviced concurrently over the network via demand paging. Active push avoids transferring pages that have already been faulted in by the target VM. Thus, each page is transferred only once, either by demand paging or by an active push.

Post-Copy via Pre-paging: The goal of post-copy via pre-paging is to anticipate the occurrence of major faults in advance and adapt the page pushing sequence to better reflect the VM's memory access pattern. While it is impossible to predict the VM's exact faulting behavior, our approach works by using the faulting addresses as hints to estimate the spatial locality of the VM's memory access pattern. The pre-paging component then shifts the transmission window of the pages to be pushed such that the current page fault location falls within the window. This increases the probability that the pushed pages would be the ones accessed by the VM in the near future, reducing the number of major faults. Our pre-paging strategy is described in greater detail in Section 3.2.

Hybrid Pre and Post Copy: The hybrid approach, first described in (19), works by doing a *single* pre-copy round in the preparation phase of the migration. During this time, the guest VM continues

```
1. let N          := total # of guest VM pages
2. let page[N]    := set of all guest VM pages
3. let bitmap[N] := all zeroes
4. let pivot := 0; bubble := 0

5. ActivePush (Guest VM)
6.   while bubble < max (pivot, N-pivot) do
7.      let left := max(0, pivot - bubble)
8.      let right := min(MAX_PAGE_NUM-1, pivot + bubble)
9.      if bitmap[left] == 0 then
10.         set bitmap[left] := 1
11.         queue page[left]  for transmission
12.     if bitmap[right] == 0 then
13.         set bitmap[right] := 1
14.         queue page[right] for transmission
15.     bubble++

16. PageFault (Guest-page X)
17.   if bitmap[X] == 0 then
18.      set bitmap[X] := 1
19.      transmit page[X] immediately
20.      discard pending queue
21.      set pivot := X   // shift pre-paging pivot
22.      set bubble := 1   // new pre-paging window
```

Figure 2. Pseudo-code for the pre-paging algorithm employed by post-copy migration. Synchronization and locking code omitted for clarity of presentation.

running *at the source* while all its memory pages are copied to the target host. After one iteration only, the VM is suspended and its processor state and *dirty* pages are copied to the target. Subsequently, the VM is resumed and the post-copy mechanism described above kicks in immediately, bringing in the remaining dirty pages from the source. As with pre-copy, this scheme can perform well for read-intensive workloads. Yet it also provides deterministic total migration time for write-intensive workloads, as with post-copy.

In the rest of this paper, we describe the design and implementation of *post-copy via pre-paging*. The hybrid approach is currently being implemented and not covered within the scope of this paper.

3.2 Pre-paging Strategy

Figure 2 lists the pseudo-code for the two components of our pre-paging algorithm – active push (lines 5–15) and page fault servicing (lines 16–22) – both of which operate as two concurrent threads of control in kernel space. The active push component starts from a *pivot* page and pushes symmetrically located pages around that pivot to the target in each iteration. We refer to this algorithm as "**bubbling**" since it is akin to a bubble that grows around the pivot as the center. Whenever a page fault occurs, the fault servicing component shifts the pivot to the location of the new page fault and starts the pre-paging bubble from that location. In this manner, the location of the pivot adapts to the occurrence of new page faults in order to exploit spatial locality. Furthermore, when a page-fault needs to be serviced, some pages may have already been queued for transmission by the pre-paging algorithm. In order to reduce the latency of servicing the immediate page fault, we purge the transmission queue and transmit the faulted page immediately. Next, we shift the pivot to the location of the new page fault. (This purge is accomplished with a custom in-kernel network protocol from prior work (10). Our protocol allows us to purge explicitly so that we may also refill the push window queue with new, spatially local pages).

3.3 Dynamic Self-Ballooning

Before migration begins, a guest VM will have an arbitrarily large number of free, unallocated pages. Transferring these pages would be a waste of network and CPU resources, and would increase the total migration time *regardless of which migration algorithm we use*. Further, if a free page is allocated by the guest VM during a post-copy migration and subsequently causes a major fault (due to a copy-on-write by the virtual memory subsystem), fetching that free page over the network, only to be overwritten immediately, will result in unnecessary execution delay for the VM at the target. Thus, it is highly desirable to avoid transmitting these free pages.

Ballooning (34) is a minimally intrusive technique for resizing the memory allocation of a VM (called a reservation). Typical ballooning implementations involve a *balloon driver* in the guest OS. The balloon driver can either reclaim pages considered least valuable by the OS and return them back to the hypervisor (inflating the balloon), or request pages from the hypervisor and return them back to the guest OS (deflating the balloon). As of this writing, Xen-based ballooning is primarily used during the initialization of a new VM. If the hypervisor cannot reserve enough memory for a new VM, it steals unused memory from other VMs by inflating their balloons to accommodate the new guest. The system administrator can re-enlarge those diminished reservations at a later time should more memory become available, such as due to VM destruction or migration.

We extend Xen's ballooning mechanism to avoid transmitting free pages during both pre-copy and post-copy migration. The guest VM performs ballooning *continuously* over its execution lifetime – a technique we call **Dynamic Self-Ballooning** (DSB). DSB reduces the number of free pages without significantly impacting the normal execution of the VM, so that the VM can be migrated quickly with a minimal memory footprint. Our DSB design responds dynamically to VM memory pressure by inflating the balloon under low pressure and deflating under increased pressure. For DSB to be both effective and minimally intrusive, we must choose an appropriate interval between consecutive invocations of ballooning such that DSB's activity does not interfere with the execution of VM's applications. Secondly, DSB must ensure that the balloon can shrink (or altogether pop) when one or more applications becomes memory-intensive. We describe the specific implementation details of DSB in Section 4.2.

3.4 Reliability

Either the source or destination node can fail during migration. In both pre-copy and post-copy, failure of the source node implies permanent loss of the VM itself. Failure of the destination node has different implications in the two cases. For pre-copy, failure of the destination node does not matter because the source node still holds an entire up-to-date copy of the VM's memory and processor state and the VM can be revived if necessary from this copy. However, when post-copy is used, the destination node has more up-to-date copy of the virtual machine and the copy at the source happens to be stale, except for pages not yet modified at the destination. Thus, failure of the destination node during post-copy migration constitutes a critical failure of the VM. Although our current post-copy implementation does not address this drawback, we plan to address this problem by developing a mechanism to incrementally checkpoint the VM state from the destination node *back at the source node*. Our approach is as follows: while the active push of pages is in progress, we also propagate incremental changes to memory and the VM's execution state at the destination back to the source node. We do not need to propagate the changes from the destination on a continuous basis, but only at discrete points such as when interacting with a remote client over the network, or committing an I/O operation to the storage. This mechanism can provide a consistent backup image at the source node that one can fall back upon in case the destination node fails in the middle of post-copy migration. The performance of this mechanism

would depend upon the additional overhead imposed by reverse network traffic from the target to the source and the frequency of incremental checkpointing. Recently, in a different context (4), similar mechanisms have been successfully used to provide high availability.

4. Implementation-specific Details

We implemented post-copy along with all of the optimizations described in Section 3 on Xen 3.2.1 and para-virtualized Linux 2.6.18.8. We begin by first discussing the different ways of trapping page faults at the target within the Xen/Linux architecture and their trade-offs. Then we will discuss our implementation of dynamic self-ballooning (DSB).

4.1 Page Fault Detection

There are three different ways by which the demand-paging component of post-copy can trap page faults at the target VM.

(1) Shadow Paging: Shadow paging refers to a set of read-only page tables for each VM maintained by the hypervisor that maps the VM's pseudo-physical pages to the physical page frames. Shadow paging can be used to trap access to non-existent pages at the target VM. For post-copy, each major fault at the target can be intercepted via these traps and be redirected to the source.

(2) Page Tracking: The idea here is to mark all of the resident pages in the VM at the target as *not present* in their page-table-entries (PTEs) during downtime. This has the effect of forcing a page fault exception when the VM accesses a page. After some third party services the fault, the PTE can be fixed up to reflect accurate mappings. PTEs in x86 carry a few unused bits to potentially support this, but this approach requires significant changes to the guest OS kernel.

(3) Pseudo-paging: The idea here is to swap out all pageable memory in the guest VM to an *in-memory pseudo-paging device* within the guest. This is done with minimal overhead and without any disk I/O. Since the source copy of the VM is suspended at the beginning of post-copy migration, the memory reservation for the VM's source copy can be made to appear as a pseudo-paging device. During resume time, the guest VM then retrieves its "swapped"-out pages through its normal page fault servicing mechanism. In order to service those faults, a modified block driver is inserted to retrieve the pages over the network.

In the end, we chose to implement the pseudo-paging option because it was the quickest to implement. In fact, we attempted page tracking first, but switched to pseudo-paging due to implementation issues. Shadow paging can provide an ideal middle ground, being faster than pseudo-paging but slower (and cleaner) than page tracking. We intend to switch to shadow paging soon. Our prototype doesn't change much except to make a hook available for post-copy to use. Recently, SnowFlock (14) used this method in the context of parallel cloud computing clusters using Xen.

The pseudo-paging approach is illustrated in Figure 3. Page-fault detection and servicing is implemented through the use of two loadable kernel modules, one inside the migrating VM and one inside Domain 0 at the source node. These modules leverage our prior work on a system called MemX (10), which provides transparent remote memory access for both Xen VMs and native Linux systems at the kernel level. As soon as migration is initiated, the memory pages of the migrating VM at the source are swapped out to a pseudo-paging device exposed by the MemX module in the guest VM. This "swap" is performed without copies using a lightweight *MFN exchange* mechanism (described below), after which the pages are mapped to Domain 0 at the source. CPU state and non-pageable memory are then transferred to the target node during downtime. Note the pseudo-paging approach implies that a small amount of non-pageable memory, typically small in-kernel caches and pinned pages, must be transferred during downtime. This increases the downtime in our current post-copy implementation. The non-pageable memory overhead can be significantly reduced via the hybrid migration approach discussed earlier.

MFN Exchanges: Swapping the VM's pages to a pseudo-paging device can be accomplished in two ways: by either transferring ownership of the pages to a co-located VM (like Xen's Domain 0) or by remapping the pseudo-physical address of the pages within the VM itself with zero copying overhead. We chose the latter because of its lower overhead (fewer calls into the hypervisor). We accomplish the remapping by executing an *MFN exchange* (machine frame number exchange) within the VM. The VM's memory reservation is first doubled and all the running processes in the system are suspended. The guest kernel is then instructed to swap out each pageable frame (through the use of existing software suspend code in the Linux kernel). Each time a frame is paged, we rewrite both the VM's PFN (pseudo-physical frame number) to MFN mapping (called a *physmap*) as well as the frame's kernel-level PTE such that we simulate an exchange between the frame's MFN with that of a free page frame. These exchanges are all batched before invoking the hypervisor. Once the exchanges are over, we memory-map the exchanged MFNs into Domain 0. The data structure needed to bootstrap this memory mapping is created within the guest on-demand as a tree that is almost identical to a page-table, from which the root is sent to Domain 0 through the Xen Store.

Once the VM resumes at the target, demand paging begins for missing pages. The MemX "client" module in the VM at the target activates again to service major faults and perform pre-paging by coordinating with the MemX "server" module in Domain 0 at the source. The two modules communicate via a customized and lightweight *remote memory access protocol* (RMAP) which directly operates above the network device driver.

4.2 Dynamic Self Ballooning Implementation

The implementation of DSB has three components. **(1) Inflate the balloon**: A kernel-level DSB thread in the guest VM first allocates as much free memory as possible and hands those pages over to the hypervisor. **(2) Detect memory pressure**: Memory pressure indicates that some entity needs to access a page frame right away. In response, the DSB process must partially deflate the balloon depending on the extent of memory pressure. **(3) Deflate the balloon**: Deflation is the reverse of Step 1. The DSB process re-populates it's memory reservation with free pages from the hypervisor and then releases the list of free pages back to the guest kernel's free pool.

Detecting Memory Pressure: Surprisingly enough, the Linux kernel already provides a transparent mechanism to detect memory pressure: through the kernel's filesystem API. Using a function called "set_shrinker()", one of the function pointers provided as a parameter to this function acts as a callback to some memory-hungry portion of the kernel. This indicates to the virtual memory system that this function can be used to request the deallocation of a requisite amount of memory that it may have pinned – typically things like inode and directory entry caches. These callbacks are indirectly driven by the virtual memory system as a result of copy-on-write faults, satisfied on behalf of an application that has allocated a large amount of memory and is accessing it for the first time. DSB does not register a new filesystem, but rather registers a similar callback function for the same purpose. (It is not necessary to register a new filesystem in order to register this callback). This worked remarkably well and provides very precise feedback to the DSB process about memory pressure. Alternatively, one could manually scan /proc statistics to determine this information, but we found the filesystem API to be more direct reflection of the decisions that the virtual memory system is actually making. Each callback contains a numeric value of exactly how many pages the DSB should re-

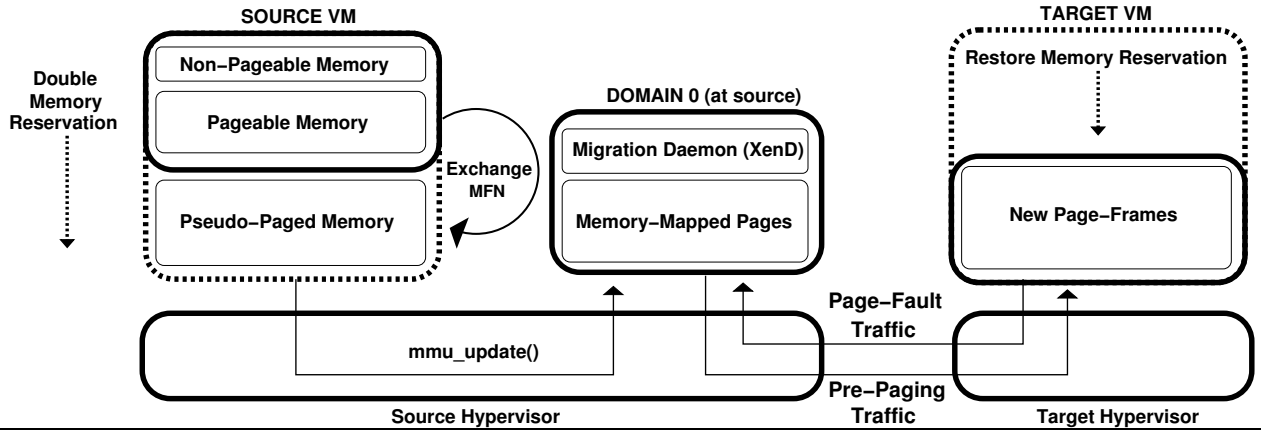

Figure 3. Pseudo-Paging : Pages are swapped out to a pseudo-paging device within the source VM's memory by exchanging MFN identifiers. Domain 0 at the source maps the swapped pages to its memory with the help of the hypervisor. Pre-paging then takes over after downtime.

lease, which typically defaults to 128 pages at a time. When the callback returns, part of the return value is used to indicate to the virtual memory system how much "pressure" is still available to be relieved. Filesystems typically return the sum totals of their caches, whereas the DSB process will return the size of the balloon itself.

Also, the DSB must periodically reclaim free pages that may have been released over time. The DSB process performs this sort of "garbage collection" by periodically waking up and re-inflating the balloon as much as possible. Currently, we will inflate to 95% of all of the available free memory. (Inflating to 100% would trigger the "out-of-memory" killer, hence the 5% buffer). If memory pressure is detected during this time, the thread preempts any attempts to do a balloon inflation and will maintain the size of the balloon for a backoff period of about 10 intervals. As we show later, an empirical analysis has shown that a ballooning interval size of about 5 seconds has proven effective.

Lines of Code. Most of the post-copy implementation is about 7000 lines within pluggable kernel modules. 4000 lines of that are part of the MemX system that is invoked during resume time. 3000 lines contribute to the pre-paging component, the pushing component, and the DSB component combined. A 200 line patch is applied to the migration daemon to support ballooning and a 300-line patch is applied to the guest kernel so as to initiate pseudo-paging. In the end, the system remains completely transparent to applications and approaches about 7500 lines. Neither the original pre-copy algorithm, nor the hypervisor is modified in any manner.

5. Evaluation

In this section, we present a detailed evaluation of our post-copy implementation and compare it against Xen's pre-copy migration. Our test environment consists of two 2.8 GHz dual core Intel machines connected via a Gigabit Ethernet switch. Each machine has 4 GB of memory. Both the guest VM in each experiment and the Domain 0 are configured to use two virtual CPUs. Guest VM sizes range from 128 MB to 1024 MB. Unless otherwise specified, the default guest VM size is 512 MB. In addition to the performance metrics mentioned in Section 3, we evaluate post-copy against an additional metric. Recall that post-copy is effective only when a large majority of the pages reach the target before they are faulted upon by the VM at the target, in which case they become *minor page faults* rather than *major network page faults*. Thus the fraction of major faults compared to minor page faults is another indication of the effectiveness of our post-copy approach.

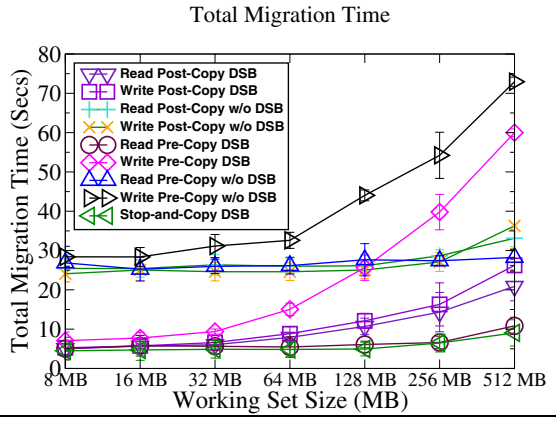

Figure 4. Comparison of total migration times.

5.1 Stress Testing

We start by first doing a stress test for both migration schemes with the use of a simple, highly sequential memory-intensive C program. This program accepts a parameter to change the working set of memory accesses and a second parameter to control whether it performs memory reads or writes during the test. The experiment is performed in a 2048 MB VM with its working set ranging from 8 MB to 512 MB. The rest is simply free memory. We perform the experiment with different test configurations:

1. **Stop-and-copy Migration**: This is a *non-live* migration which provides a baseline to compare the total migration time and number of pages transferred by post-copy.
2. **Read-intensive Pre-Copy with DSB**: This configuration provides the best-case workload for pre-copy. The performance is expected to be roughly similar to pure stop-and-copy migration.
3. **Write-intensive Pre-Copy with DSB**: This configuration provides the worst-case workload for pre-copy.
4. **Read-intensive Pre-Copy without DSB**:
5. **Write-intensive Pre-Copy without DSB**: These two configurations test the default implementation of pre-copy in Xen.
6. **Read-intensive Post-Copy with and without DSB**:
7. **Write-intensive Post-Copy with and without DSB**: These four configurations will stress our pre-paging algorithm. Both reads and writes are expected to perform almost identically. DSB is expected to minimize the transmission of free pages.

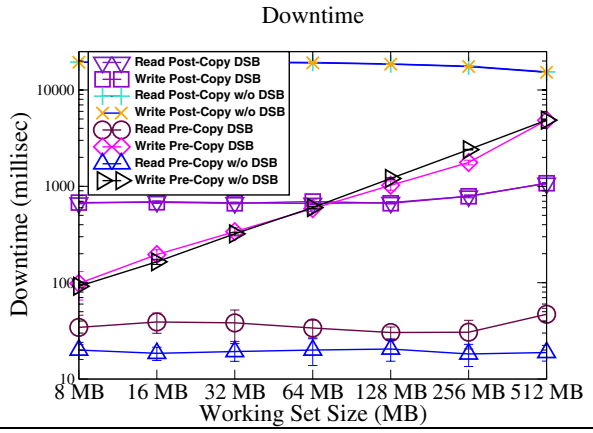

Figure 5. Comparison of downtimes. (Y-axis in log scale).

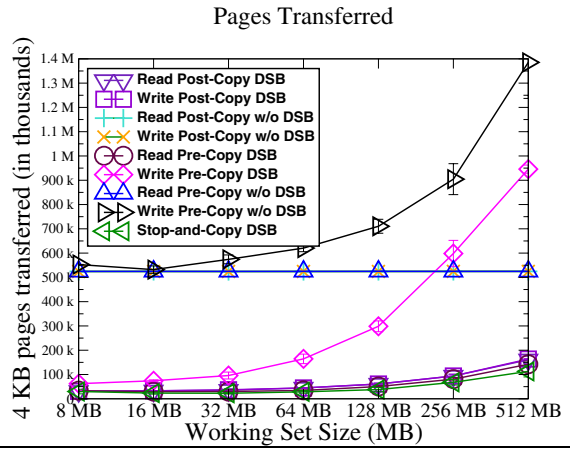

Figure 6. Comparison of the number of pages transferred during a single migration.

Total Migration Time: Figure 4 shows the variation of total migration time with increasing working set size. Notice that both post-copy plots with DSB are at the bottom, surpassed only by read-intensive pre-copy with DSB. Both the read and write intensive tests of post-copy perform very similarly. *Thus our post-copy algorithm's performance is agnostic to the read or write-intensive nature of the application workload.* Furthermore, we observe that without DSB activated, the total migration times are high for all migration schemes due to unnecessary transmission of free pages over the network.

Downtime: Figure 5 compares the metric of downtime as the working set size increases. As expected, read-intensive pre-copy gives the lowest downtime, whereas that for write-intensive pre-copy increases as the size of the writable working set increases. For post-copy, recall that our choice of pseudo-paging for page fault detection (in Section 4) increases the downtime since all non-pageable memory pages are transmitted during downtime. With DSB, post-copy achieves a downtime that ranges between 600 milliseconds to just over one second. However, without DSB, our post-copy implementation experiences a large downtime of around 20 seconds because *all* free pages in the guest kernel are treated as non-pageable pages and transferred during downtime. Hence the use of DSB is essential to maintain a low downtime with our current implementation of post-copy. This limitation can be overcome by the use of shadow paging to track page-faults.

Working Set Size	Pre-Paging		Pushing	
	Net	Minor	Net	Minor
8 MB	2%	98%	15%	85%
16 MB	4%	96%	13%	87%
32 MB	4%	96%	13%	87%
64 MB	3%	97%	10%	90%
128 MB	3%	97%	9%	91%
256 MB	3%	98%	10%	90%

Table 2. Percent of minor and network faults for pushing vs. pre-paging. Pre-paging greatly reduces the fraction of network faults.

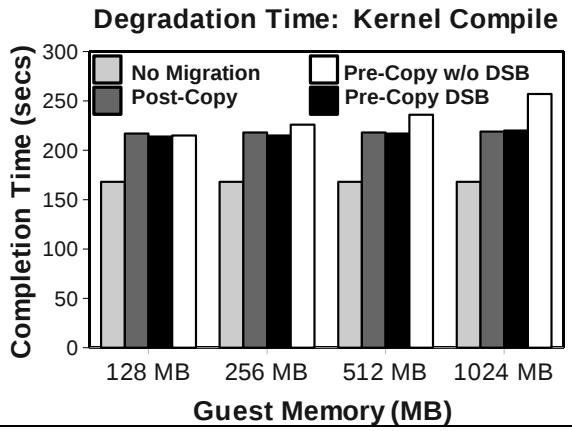

Figure 7. Kernel compile with back-to-back migrations using 5 seconds pauses.

Pages Transferred and Page Faults: Figure 6 and Table 2 illustrate the utility of our pre-paging algorithm in post-copy across increasingly large working set sizes. Figure 6 plots the total number of pages transferred. As expected, post-copy transfers far fewer pages than write-intensive pre-copy. It performs on par with read-intensive post-copy and stop-and-copy. Without DSB, the number of pages transferred increase significantly for both pre-copy and post-copy. Table 13 compares the fraction of network and minor faults in post-copy. We see that pre-paging reduces the fraction of network faults from 7% to 13%. To be fair, the stress-test is highly sequential in nature and consequently, pre-paging predicts this behavior almost perfectly. We expect the real applications in the next section to do worse than this ideal case.

5.2 Degradation, Bandwidth, and Ballooning

Next, we quantify the side effects of migration on a couple of sample applications. We want to answer the following questions: What kinds of slow-downs do VM workloads experience during pre-copy versus post-copy migration? What is the impact on network bandwidth received by applications? And finally, what kind of balloon inflation interval should we choose to minimize the impact of DSB on running applications? For application degradation and the DSB interval, we use Linux kernel compilation. For bandwidth testing we use the NetPerf TCP benchmark.

Degradation Time: Figure 7 depicts a repeat of an interesting experiment from (18). We initiate a kernel compile inside the VM and then migrate the VM repeatedly between two hosts. We script the migrations to pause for 5 seconds each time. Although there is no exact way to quantify degradation time (due to scheduling and context switching), this experiment provides an approximate measure. As far as memory is concerned, we observe that kernel compilation tends not to exhibit too many memory writes. (Once

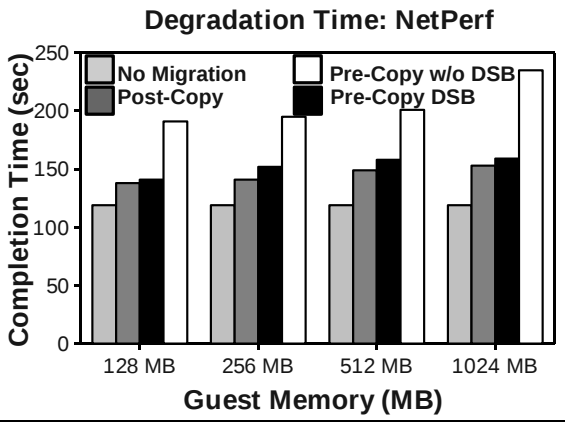

Figure 8. NetPerf run with back-to-back migrations using 5 seconds pauses.

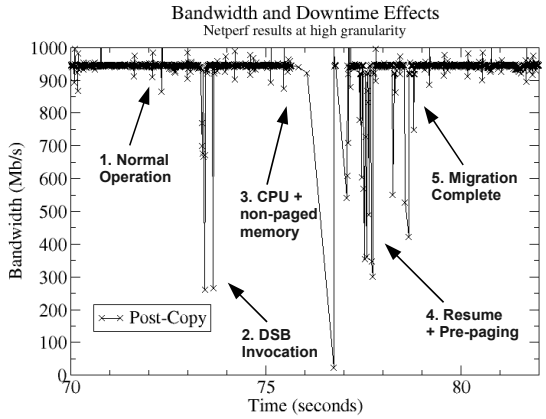

Figure 9. Impact of post-copy on NetPerf bandwidth.

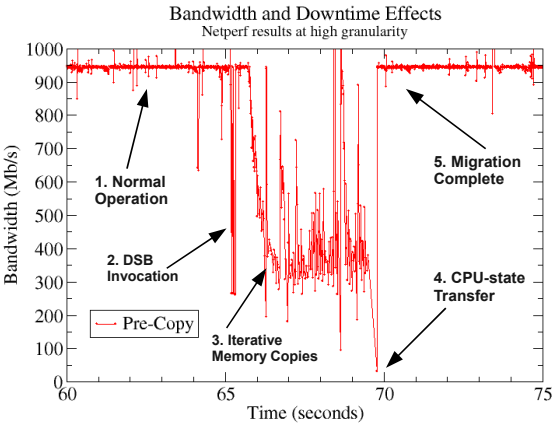

Figure 10. Impact of pre-copy on NetPerf bandwidth.

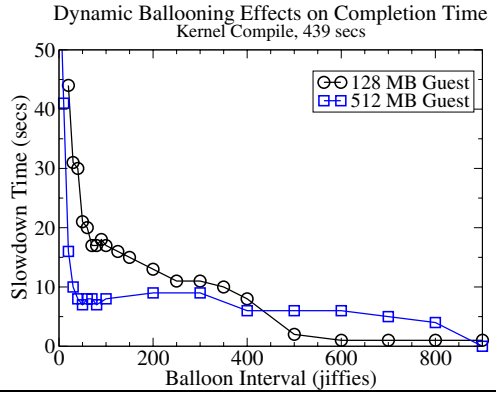

Figure 11. Application degradation is inversely proportional to the ballooning interval.

gcc forks and compiles, the OS page cache will only be used once more at the end to link the kernel object files together). As a result, the experiment represents the *best case* for the original pre-copy approach when there is not much repeated dirtying of pages. This experiment is also a good worst-case test for our implementation of Dynamic Self Ballooning due to the repeated fork-and-exit behavior of the kernel compile as each object file is created over time. (Interestingly enough, this experiment also gave us a headache, because it exposed the bugs in our code!) We were surprised to see how many additional seconds were added to the kernel compilation in Figure 7 just by executing back to back invocations of pre-copy migration. Nevertheless, we observe that post-copy closely matches pre-copy in the amount of degradation. This is in line with the competitive performance of post-copy with read-intensive pre-copy tests in Figures 4 and 6. We suspect that a shadow-paging based implementation of post-copy would perform much better due to the significantly reduced downtime it would provide. Figure 8 shows the same experiment using NetPerf. A sustained, high-bandwidth stream of network traffic causes slightly more page-dirtying than the compilation does. The setup involves placing the NetPerf sender inside the guest VM and the receiver on an external node on the same switch. Consequently, regardless of VM size, post-copy actually does perform slightly better and reduce the degradation time experienced by NetPerf.

Effect on Bandwidth: In their paper (3), the Xen project proposed a solution called "adaptive rate limiting" to control the bandwidth overhead due to migration. However, this feature is not enabled in the currently released version of Xen. In fact it is compiled out without any runtime options or any pre-processor directives. This could likely be because rate-limiting increases the total migration time, or even because it is difficult, if not impossible, to predict beforehand the bandwidth requirements of any single guest VM, on the basis of which to guide adaptive rate limiting. We do not activate rate limiting for our post-copy implementation either so as to normalize the comparison of the two migration techniques. We believe the omission of rate limiting makes sense: If the guest is hosting, say, a webserver, then the webserver will take whatever size network pipe in can get its hands on. This suggests that the migration daemon should just let TCP's own fairness policies act normally across all virtual machines.

With that in mind, Figures 9 and 10 show a visual representation of the reduction in bandwidth experienced by a high-throughput NetPerf session. We conduct this experiment by invoking VM migration in the middle of a NetPerf session and measuring bandwidth values rapidly throughout. The impact of migration can be seen in both figures by a sudden reduction in the observed bandwidth during migration. This reduction is more sustained, and greater, for pre-copy than for post-copy due to the fact that the total number of pages transferred in pre-copy is much higher. This is exactly the bottom line that we were targeting for improvement.

Dynamic Ballooning Interval: Figure 11 shows how we chose the DSB interval, by which the DSB process wakes up to reclaim

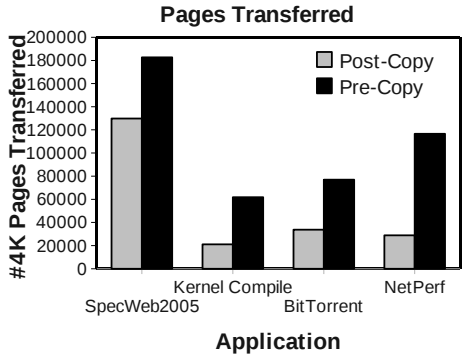

Figure 12. Total pages transferred for both migration schemes.

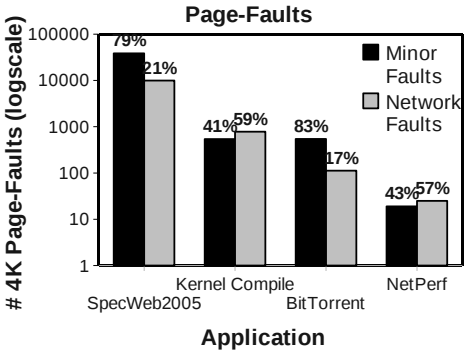

Figure 13. Page-fault comparisons: Pre-paging lowers the network page faults to 17% and 21%, even for the heaviest applications.

available free memory. With the kernel compile as a test application, we execute the DSB process at intervals from 10ms to 10s. At every interval, we script the kernel compile to run multiple times and output the average completion time. The difference in that number from the base case is the *degradation time* added to the application by the DSB process due to its CPU usage. As expected, the choice of ballooning interval is inversely proportional to the application degradation. The more often you balloon, the more it affects the VM workload. The graph indicates that we should choose an interval between 4 and 10 seconds to balance between frequently reclaiming free pages and application CPU activity.

5.3 Application Scenarios

The last part of our evaluation is to re-visit the aforementioned performance metrics across four real applications:

1. **SPECWeb 2005**: This is our largest application. It is a well-known webserver benchmark involving at least 2 or more physical hosts. We place the system under test within the guest VM, while six separate client nodes bombard the VM with connections.
2. **Bit Torrent Client**: Although this is not a typical server application, we chose it because it is a simple representative of a multi-peer distributed application. It is easy to initiate and does not immediately saturate a Gigabit Ethernet pipe. Instead, it fills up the network pipe gradually, is slightly CPU intensive, and involves a somewhat more complex mix of page-dirtying and disk I/O than just a kernel compile.
3. **Linux Kernel Compile**: We consider this again for consistency.
4. **NetPerf**: Once more, as in the previous experiments, the NetPerf sender is placed inside the guest VM.

Using these applications, we evaluate the same four primary metrics that we covered in Section 5.1: downtime, total migration time, pages transferred, and page faults. Each figure for these applications represents one of the four metrics and contains results for a constant, 512 MB virtual machine in the form of a bar graph for both migration schemes across each application. Each data point is the average of 20 samples. And just as before, the guest VM is configured to have two virtual CPUs. All of these experiments have the DSB process activated.

Pages Transferred and Page Faults.. The experiments in Figures 12 and 13 illustrate these results. For all of the applications except the SPECWeb, post-copy reduces the total pages transferred by more than half. The most significant result we've seen so far is in Figure 13 where post-copy's pre-paging algorithm is able to avoid 79% and 83% of the network page faults (which become minor faults) for the largest applications (SPECWeb, Bittorrent). For the smaller applications (Kernel, NetPerf), we still manage to save 41% and 43% of network page faults. There is a significant amount of

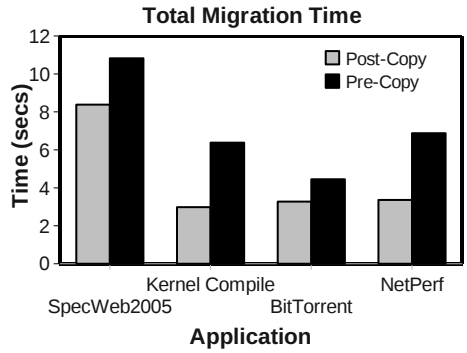

Figure 14. Total migration time for both migration schemes.

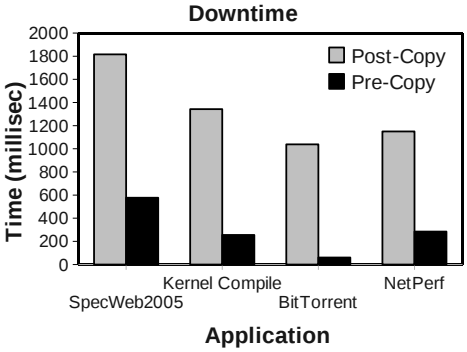

Figure 15. Downtime for post-copy vs. pre-copy. Post-copy downtime can improve with better page-fault detection.

additional prior work in the literature aimed at working-set identification, and we believe that these improvements can be even better if we employ both knowledge-based and history-based predictors in our pre-paging algorithm. But even with a reactive approach, post-copy appears to be a strong competitor.

Total Time and Downtime. Figure 14 shows that post-copy reduces the total migration time for all applications, when compared to pre-copy, in some cases by more than 50%. However, the downtime in Figure 15 is currently much higher for post-copy than for pre-copy. As we explained earlier, the relatively high downtime is due to our speedy choice of pseudo-paging for page fault detection, which we plan to reduce through the use of shadow paging. Nevertheless, this tradeoff between total migration time and downtime

may be acceptable in situations where network overhead needs to be kept low and the entire migration needs to be completed quickly.

6. Conclusions

We have presented the design and implementation of a post-copy technique for live migration of virtual machines. Post-copy is a combination of four key components: demand paging, active pushing, pre-paging, and dynamic self-ballooning. We have implemented and evaluated post-copy on a Xen and Linux based platform and shown that it is able to achieve significant performance improvements over pre-copy based live migration by reducing the number of pages transferred over the network and the total migration time. In future work, we plan to investigate an alternative to pseudo-paging, namely shadow paging based page fault detection. We are also developing techniques to handle destination node failure during post-copy migration, so that post-copy can provide at least the same level of reliability as pre-copy. Finally, we are implementing a hybrid pre/post copy approach where a single round of pre-copy precedes the CPU state transfer, followed by a post-copy of the remaining dirty pages from the source.

7. Acknowledgments

We'd like to thank AT&T Labs Research at Florham Park, New Jersey, for providing the fellowship funding to Michael Hines for this research. This work is also supported in part by the National Science Foundation through grants CNS-0509131 and CCF-0541096.

References

[1] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proc. of ACM SOSP 2003* (Oct. 2003).

[2] BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live wide-area migration of virtual machines including local persistent state. In *Proc. of the International Conference on Virtual Execution Environments* (2007), pp. 169–179.

[3] CLARK, C., FRASER, K., HAND, S., HANSEN, J., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Network System Design and Implementation* (2005).

[4] CULLY, B., LEFEBVRE, G., AND MEYER, D. Remus: High availability via asynchronous virtual machine replication. In *NSDI '07: Networked Systems Design and Implementation* (2008).

[5] DENNING, P. J. The working set model for program behavior. *Communications of the ACM 11*, 5 (1968), 323–333.

[6] DOUGLIS, F. Transparent process migration in the Sprite operating system. Tech. rep., University of California at Berkeley, Berkeley, CA, USA, 1990.

[7] HAND, S. M. Self-paging in the nemesis operating system. In *OSDI'99, New Orleans, Louisiana, USA* (1999), pp. 73–86.

[8] HANSEN, J., AND HENRIKSEN, A. Nomadic operating systems. In *Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark* (2002).

[9] HANSEN, J., AND JUL, E. Self-migration of operating systems. In *Proc. of ACM SIGOPS Europen Workshop, Leuven, Belgium* (2004).

[10] HINES, M., AND GOPALAN, K. MemX: Supporting large memory applications in Xen virtual machines. In *Second International Workshop on Virtualization Technology in Distributed Computing (VTDC07), Reno, Nevada* (2007).

[11] HO, R. S., WANG, C.-L., AND LAU, F. C. Lightweight process migration and memory prefetching in OpenMosix. In *Proc. of IPDPS* (2008).

[12] KERRIGHED. *http://www.kerrighed.org*.

[13] KIVITY, A., KAMAY, Y., AND LAOR, D. kvm: the linux virtual machine monitor. In *Proc. of Ottawa Linux Symposium* (2007).

[14] LAGAR-CAVILLA, H. A., WHITNEY, J., SCANNEL, A., RUMBLE, S., BRUDNO, M., DE LARA, E., AND SATYANARAYANAN, M. Impromptu clusters for near-interactive cloud-based services. Tech. rep., CSRG-578, University of Toronto, June 2008.

[15] MAGENHEIMER, D. *Add self-ballooning to balloon driver*. Discussion on Xen Development mailing list and personal communication, April 2008.

[16] MILOJICIC, D., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. Process migration survey. *ACM Computing Surveys 32(3)* (Sep. 2000), 241–299.

[17] MOSIX. *http://www.mosix.org*.

[18] NELSON, M., LIM, B.-H., AND HUTCHINS, G. Fast transparent migration for virtual machines. In *Usenix, Anaheim, CA* (2005), pp. 25–25.

[19] NOACK, M. Comparative evaluation of process migration algorithms. Master's thesis, Dresden University of Technology - Operating Systems Group, 2003.

[20] OPENVZ. *Container-based Virtualization for Linux, http://www.openvz.com/*.

[21] OPPENHEIMER, G., AND WEIZER, N. Resource management for a medium scale time-sharing operating system. *Commun. ACM 11*, 5 (1968), 313–322.

[22] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of Zap: A system for migrating computing environments. In *Proc. of OSDI* (2002), pp. 361–376.

[23] PLANK, J., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: transparent checkpointing under unix. In *Proc. of Usenix Annual Technical Conference, New Orleans, Louisiana* (1998).

[24] RICHMOND, M., AND HITCHENS, M. A new process migration algorithm. *SIGOPS Oper. Syst. Rev. 31*, 1 (1997), 31–42.

[25] ROUSH, E. T. Fast dynamic process migration. In *Intl. Conference on Distributed Computing Systems (ICDCS)* (1996), p. 637.

[26] SAPUNTZAKIS, C., CHANDRA, R., AND PFAFF, B. Optimizing the migration of virtual computers. In *Proc. of OSDI* (2002).

[27] SATYANARAYANAN, M., AND GILBERT, B. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing 11*, 2 (2007), 16–25.

[28] SCHMIDT, B. K. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Computer Science Dept., Stanford University, 2000.

[29] STELLNER, G. Cocheck: Checkpointing and process migration for mpi. In *IPPS '1996* (Washington, DC, USA), pp. 526–531.

[30] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed computing in practice: the condor experience. *Concurr. Comput. : Pract. Exper. 17* (2005), 323–356.

[31] TRIVEDI, K. An analysis of prepaging. *Journal of Computing 22* (1979), 191–210.

[32] TRIVEDI, K. On the paging performance of array algorithms. *IEEE Transactions on Computers C-26*, 10 (Oct. 1977), 938–947.

[33] TRIVEDI, K. Prepaging and applications to array algorithms. *IEEE Transactions on Computers C-25*, 9 (Sept. 1976), 915–921.

[34] WALDSPURGER, C. Memory resource management in vmware esx server. In *Operating System Design and Implementation (OSDI 02), Boston, MA* (Dec 2002).

[35] WHITAKER, A., COX, R., AND SHAW, M. Constructing services with interposable virtual hardware. In *NSDI 2004* (2004), pp. 13–13.

[36] WHITAKER, A., SHAW, M., AND GRIBBLE, S. Scale and performance in the denali isolation kernel. In *OSDI 2002, New York, NY, USA* (2002), pp. 195–209.