Post-Copy Live Migration of Virtual Machines

Michael R. Hines, Umesh Deshpande, and Kartik Gopalan
Computer Science, Binghamton University (SUNY)
{mhines,udeshpa1,kartik}@cs.binghamton.edu

ABSTRACT [1]

We present the design, implementation, and evaluation of *post-copy* based live migration for virtual machines (VMs) across a Gigabit LAN. 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 while maintaining the liveness of the VM during migration. We compare post-copy extensively against the traditional pre-copy approach on the Xen Hypervisor. Using a range of VM workloads we show that post-copy improves several metrics including pages transferred, total migration time, and network overhead. We facilitate the use of post-copy with adaptive *prepaging* techniques to minimize the number of page faults across the network. We propose different prepaging strategies and quantitatively compare their effectiveness in reducing network-bound page faults. Finally, we eliminate the transfer of free memory pages in both pre-copy and post-copy through a *dynamic self-ballooning* (DSB) mechanism. DSB periodically reclaims free pages from a VM and significantly speeds up migration with negligible performance impact on VM workload.

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. State-of-the-art live migration techniques [19, 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, where the VM is restarted.

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. Pre-copy 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 because pages that are repeatedly dirtied may have to be transmitted multiple times.

In this paper, we propose and evaluate the *postcopy* strategy for live VM migration, previously studied only in the context of process migration. At 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 the VM's 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.

Effectiveness of post-copy depends on the ability to minimize the number of network-bound page-faults (or network faults), by pushing the pages from source *before* they are faulted upon by the VM at target. To reduce network faults, we supplement the active push component of post-copy with *adaptive prepaging*. Prepaging is a term borrowed from earlier literature [22, 32] on optimizing memory-constrained

[1] A shorter version of this paper appeared in the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), March 2009 [11]. The additional contributions of this paper are new prepaging strategies including dual-direction and multi-pivot bubbling (Section 3.2), proactive LRU ordering of pages (Section 4.3), and their evaluation (Section 5.4).

disk-based paging systems. It traditionally refers to a more proactive form of pre-fetching from storage devices (such as hard disks) in which the memory subsystem can try to hide the latency of high-locality page faults by intelligently sequencing the pre-fetched pages. Modern virtual memory subsystems do not typically employ prepaging due increasing DRAM capacities. Although post-copy doesn't deal with disk-based paging, the prepaging algorithms themselves can still play a helpful role in reducing the number of network faults in post-copy. Prepaging adapts the sequence of actively pushed pages by using network faults as hints to predict the VM's page access locality at the target and actively push the pages in the neighborhood of a network fault before they are accessed by the VM. We propose and compare a number of prepaging strategies for post-copy, which we call *bubbling*, that reduce the number of network faults to varying degrees.

Additionally, we identified a deficiency in both pre-copy and post-copy migration due to which free pages in the VM are also transmitted during migration, increasing the total migration time. To avoid transmitting the free pages, we develop a "dynamic self-ballooning" (DSB) mechanism. Ballooning is an existing technique that allows a guest kernel to reduce its memory footprint by releasing its free memory pages back to the hypervisor. DSB automates the ballooning mechanism so it can trigger periodically (say every 5 seconds) without degrading application performance. Our DSB implementation reacts directly to kernel memory allocation requests without the need for guest kernel modifications. It neither requires external introspection by a co-located VM nor excessive communication with the hypervisor. We show that DSB significantly reduces total migration time by eliminating the transfer of free memory pages in both pre-copy and post-copy.

The original pre-copy algorithm has advantages of its own. It can be implemented in a relatively self-contained external migration daemon to isolate most of the copying complexity to a single process at each node. Further, pre-copy also provides a clean way to *abort the migration* should the target node ever crash during migration because the VM is still running at the source. (Source node failure is fatal to both migration schemes.) Although our current post-copy implementation cannot recover from failure of the target node during migration, we discuss approaches in Section 3.4 by which post-copy could potentially provide the same level of reliability as pre-copy.

We designed and implemented a prototype of the post-copy live VM migration in the Xen VM environment. Through extensive evaluations, we demonstrate situations in which post-copy can significantly improve performance in terms of total migration time and pages transferred. We note that post-copy and pre-copy complement each other in 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 VMs with read-intensive workloads, pre-copy would be the better approach whereas for large-memory or write-intensive workloads, post-copy would better suited. Our main contribution is in demonstrating that a post-copy based approach is practical for live VM migration and to evaluate its merits and drawbacks against the pre-copy approach.

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 [26], then evaluated via simulations [25], and later via actual Linux implementation [20]. There was also a recent implementation of post-copy process migration under openMosix [12]. In contrast, our contributions are to develop a viable post-copy technique for live migration of virtual machines. Process migration techniques in general have been extensively researched and an excellent survey can be found in [17]. Several distributed computing projects incorporate process migration [31, 24, 18, 30, 13, 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.

PrePaging: Prepaging 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. Prepaging is also known as adaptive prefetching or adaptive remote paging. It has been studied extensively [22, 33, 34, 32] in the context of disk based storage systems, since disk I/O accesses in the critical application execution path can be highly expensive. Traditional prepaging algorithms use reactive and history based approaches to predict and prefetch the working set of the application. Our system employs prepaging, not in the context of disk prefetching, but for the limited duration of live VM migration to avoid the latency of network page faults from target to source. Our implementation employs a reactive approach that uses any network faults as hints about the VM's working set with additional optimizations described in Section 3.1.

Live VM Migration. Pre-copy is the predominant approach for live VM migration. These include hypervisor-based approaches from VMware [19], Xen [3], and KVM [14], OS-level approaches that do not use hypervisors from OpenVZ [21], as well as wide-area migration [2]. 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 [15]. 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 against (or optimize upon) the original pre-copy approach.

Non-Live VM Migration. There are several *non-live* approaches to VM migration. Schmidt [29] proposed using capsules, which are groups of related processes along with their IPC/network state, as migration units. Similarly, Zap [23] uses process groups (pods) along with their kernel state as migration units. The Denali project [37, 36] addressed migration of checkpointed VMs. Work in [27] addressed user mobility and system administration by encapsulating the computing environment as capsules to be transferred between distinct hosts. Internet suspend/resume [28] focuses on saving/restoring computing state on anonymous hardware. In all the above systems, the VM execution sus-

pended and applications do not make progress.

Dynamic Self-Ballooning (DSB): Ballooning refers to artificially requesting memory within a guest kernel and releasing that memory back to the hypervisor. Ballooning is used widely for the purpose of VM memory resizing by both VMWare [35] 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 [16] to enable a guest kernel to dynamically return free memory to the hypervisor without explicit human intervention. VMWare ESX server [35] 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

In this section we present the design 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 memory. 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 running in 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 needs to service 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, prepaging, 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. Prepaging 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 different migration

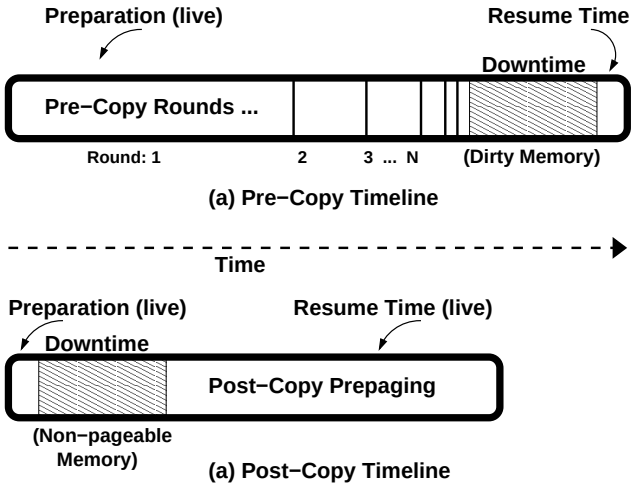

Figure 1: Timeline for Pre-copy vs. Post-copy.

techniques, each of which is described below in 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 from the viewpoint of total migration time and application degradation.

	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. Prepaging + 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 the order of their incremental improvements. Method 4 combines methods 2 and 3 with the use of prepaging. Method 5 combines all of 1 through 4, with pre-copy only performing a single 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 Prepaging: The goal of post-copy via prepaging 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 prepaging 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 pushed pages would be the ones accessed by the VM in the near future, reducing the number of major faults. Various prepaging strategies are described in Section 3.2.

Hybrid Live Migration: The hybrid approach was first described in [20] for process migration. It works by doing a *single* pre-copy round in the preparation phase of the migration. During this time, the VM continues running *at the source* while all its memory pages are copied to the target host. After just one iteration, the VM is suspended and its processor state and *dirty* non-pageable pages are copied to the target. Subsequently, the VM is resumed at target and post-copy described above kicks in, pushing 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. This hybrid approach is currently being implemented and not covered within the scope of this paper. Rest of this paper describes the design and implementation of *post-copy via prepaging*.

3.2 Prepaging Strategy

Prepaging refers to actively pushing the VM's pages from the source to the target. The goal is to make pages available at the target *before* they are faulted on by the running VM.

```
1. let N         := total # of pages in VM
2. let page[N]   := set of all VM pages
3. let bitmap[N] := all zeroes
4. let pivot := 0; bubble := 0 // place pivot at the start

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.      set pivot := X   // shift prepaging pivot
21.      set bubble := 1   // new prepaging window
```

Figure 2: Pseudo-code for the Bubbling algorithm with a single pivot. Synchronization and locking code are omitted for clarity.

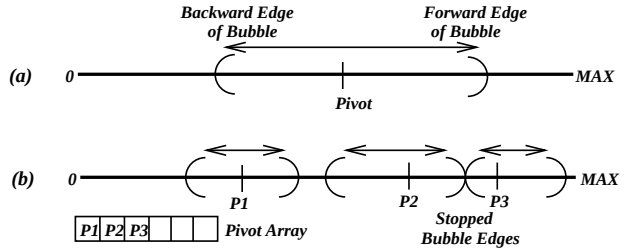

Figure 3: Prepaging strategies: (a) Bubbling with single pivot and (b) Bubbling with multiple pivots. Each pivot represents the location of a network fault on the in-memory pseudo-paging device. Pages around the pivot are actively pushed to target.

The effectiveness of prepaging is measured by the percentage of VM's page faults at the target that require an explicit page request to be sent over the network to the source node – also called *network page faults*. The smaller the percentage of network page faults, the better the prepaging algorithm. The challenge in designing an effective prepaging strategy is to accurately predict the pages that might be accessed by the VM in the near future, and to push those pages before the VM faults upon them. Below we describe different design options for prepaging strategies.

(A) Bubbling with a Single Pivot: Figure 2 lists the pseudo-code for the two components of bubbling with a single pivot – active push (lines 5–15), which executes in a kernel thread, and page fault servicing (lines 16–21), which executes in the interrupt context whenever a page-fault occurs. Figure 3(a) illustrates this algorithm graphically. The VM's pages at source are kept in an *in-memory pseudo-paging device*, which is similar to a traditional swap device except that it resides completely in memory (see Section 4 for details). The active push component starts from a *pivot* page in the pseudo-paging device and transmits symmetrically located

pages around that pivot 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. Even if one edge of the bubble reaches the boundary of pseudo-paging device (0 or MAX), the other edge continues expanding in the opposite direction. To start with, the pivot is initialized to the first page in the in-memory pseudo-paging device, which means that initially the bubble expands only in the forward direction. Subsequently, whenever a network page fault occurs, the fault servicing component shifts the pivot to the location of the new fault and starts a new bubble around this new location. In this manner, the location of the pivot adapts to new network faults in order to exploit the spatial locality of reference. Pages that have already been transmitted (as recorded in a bitmap) are skipped over by the edge of the bubble. Network faults that arrive at the source for a page that is in-flight (or just been pushed) to the target are ignored to avoid duplicate page transmissions.

(B) Bubbling with Multiple Pivots: Consider the situation where a VM has multiple processes executing concurrently. Here, a newly migrated VM would fault on pages at multiple locations in the pseudo-paging device. Consequently, a single pivot would be insufficient to capture the locality of reference across multiple processes in the VM. To address this situation, we extend the bubbling algorithm described above to operate on *multiple pivots*. Figure 3(b) illustrates this algorithm graphically. The algorithm is similar to the one outlined in Figure 2, except that the active push component pushes pages from multiple "bubbles" concurrently. (We omit the pseudo-code for space constraints, since it is a straightforward extension of single pivot case.)

Each bubble expands around an independent pivot. Whenever a new network fault occurs, the faulting location is recorded as one more pivot and a new bubble is started around that location. To save on unnecessary page transmissions, if the edge of a bubble comes across a page that is already transmitted, that edge stops progressing in the corresponding direction. For example, the edges between bubbles around pivots P2 and P3 stop progressing when they meet, although the opposite edges continue making progress. In practice, it is sufficient to limit the number of concurrent bubbles to those around k *most recent* pivots. When new network faults arrives, we replace the oldest pivot in a pivot array with the new network fault location. For the workloads tested in our experiments in Section 5, we found that around $k = 7$ pivots provided the best performance.

(C) Direction of Bubble Expansion: We also wanted to examine whether the pattern in which the source node pushes the pages located around the pivot made a significant difference in performance. In other words, is it better to expand the bubble around a pivot in both directions, or only the forward direction, or only the backward direction? To examine this we included an option of turning off the bubble expansion in either the forward or the backward direction. Our results, detailed in Section 5.4, indicate that *forward* bubble expansion is essential, *dual* (bi-directional) bubble expansion performs slightly better in most cases, and backwards-only bubble expansion is counter-productive.

When expanding bubbles with multiple pivots in only a single direction (forward-only or backward-only), there is a possibility that the entire active push component could stall before transmitting all pages in the pseudo-paging device. This happens when all active bubble edges encounter already sent-pages at their edges and stop progressing. (A simple thought exercise can show that stalling of active push is not a problem for dual-direction multi-pivot bubbling.) While there are multiple ways to solve this problem, we chose a simple approach of designating the initial pivot (at the first page in pseudo-paging device) as a *sticky* pivot. Unlike other pivots, this sticky pivot is never replaced by another pivot. Further, the bubble around sticky pivot does not stall when it encounters an already transmitted page; rather it skips such a page and keeps progressing, ensuring that the active push component never stalls.

3.3 Dynamic Self-Ballooning

Before migration begins, a 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 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 free pages. *Ballooning* [35] 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 kernel. 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 kernel (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 VM. The system administrator can re-enlarge those diminished reservations at a later time should more memory become available. We extend Xen's ballooning mechanism to avoid transmitting free pages during both pre-copy and post-copy migration. The 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 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 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 first discuss 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 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 it requires significant changes to the guest kernel.

(3) Pseudo-paging: The idea here is to swap out all pageable memory in the VM to an *in-memory pseudo-paging device* within the guest kernel. 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 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 [15] used this method in the context of parallel cloud computing clusters using Xen.

The pseudo-paging approach is illustrated in Figure 4. 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 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 re-write 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 VM 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 prepaging by coordinating with the MemX "server" module in Domain 0 at the source. The two modules communicate via a customized and lightweight *remote memory*

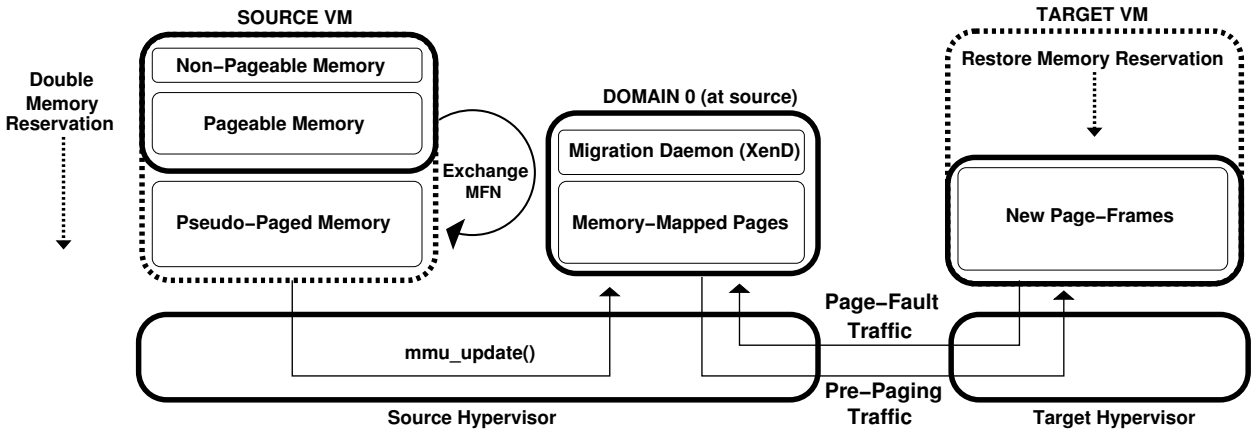

Figure 4: Pseudo-Paging : Pages are swapped out to a pseudo-paging device within the source VM's memory by exchanging MFN identifiers. Domain 0 at the source maps the swapped pages to its memory with the help of the hypervisor. Prepaging then takes over after downtime.

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 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: In order to detect memory pressure in a completely transparent manner, we begin by employing one of two mechanisms in the Linux Kernel. First, we depend on the kernel's existing ability to perform *physical memory overcommitment*. Memory overcommitment within an individual OS allows the virtual memory subsystem to provide the illusion of infinite physical memory (as opposed to just infinite virtual memory only, depending on the number of bits in the architecture of the CPU). Linux, however, offers *multiple operating modes* of physical memory overcommitment - and these modes can be changed during runtime. By default, Linux disables this feature, which precludes application memory allocations in advance by returning an error. However if you enable overcommitment, the kernel will return success. Without this, we cannot detect memory pressure transparently.

Second, coupled with over-commitment, the kernel also 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 release, which typically defaults to 128 pages at a time. When the callback returns, part of the return value indicates 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 prepaging 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 transparent to applications and approaches about 7500 lines. Neither original pre-copy implementation, nor the hypervisor is modified in any way.

4.3 Proactive LRU Ordering to Improve Reference Locality

During normal operation, the guest kernel maintains the age of each allocated page frame in its page cache. Linux, for example, maintains two linked lists in which pages are maintained in Least Recently Used (LRU) order: one for active pages and one for inactive pages. A kernel daemon periodically ages and transfers these pages between the two lists. The inactive list is subsequently used by the paging system to reclaim pages and write to the swap device. As a result, the order in which pages are written to the swap device reflects the historical locality of access by processes in the VM. Ideally, the active push component of post-copy could simply use this ordering of pages in its pseudo-paging device to predict the page access pattern in the migrated VM and push pages just in time to avoid network faults. However, Linux does not actively maintain the LRU ordering in these lists until a swap device is enabled. Since a pseudo-paging device is enabled just before migration, post-copy would not automatically see pages in the swap device ordered in the LRU order. To address this problem, we implemented a kernel thread which periodically scans and reorders the active and inactive lists in LRU order, without modifying the core kernel itself. In each scan, the thread examines the *referenced bit* of each page. Pages with their referenced bit set are moved to the most recently used end of the list and their referenced bit is reset. This mechanism supplements the kernel's existing aging support without the requirement that a real paging device be turned on. Section 5.4 shows that such a proactive LRU ordering plays a positive role in reducing network faults.

5. EVALUATION

In this section, we present a detailed evaluation of post-copy implementation and compare it against Xen's pre-copy migration. Our test environment consists of 2.8 GHz multi-core Intel machines connected via a Gigabit Ethernet switch and having between 4 to 16 GB of memory. Both the 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 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 post-copy.

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.
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 prepaging algorithm. Both reads and writes are expected to perform almost identically. DSB is expected to minimize the transmission of free pages.

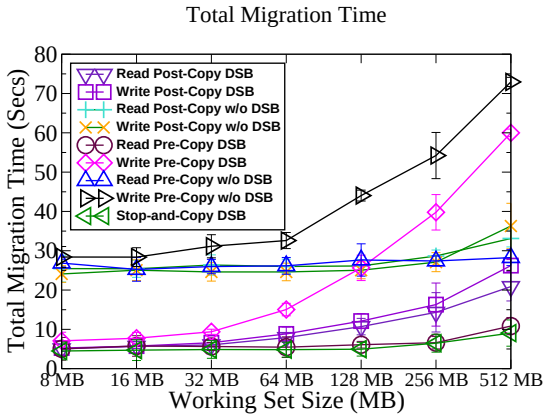

Figure 5: Comparison of total migration times.

Total Migration Time: Figure 5 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, without DSB activated, the total migration times are high for all migration schemes due to unnecessary transmission of free pages.

Downtime: Figure 6 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.

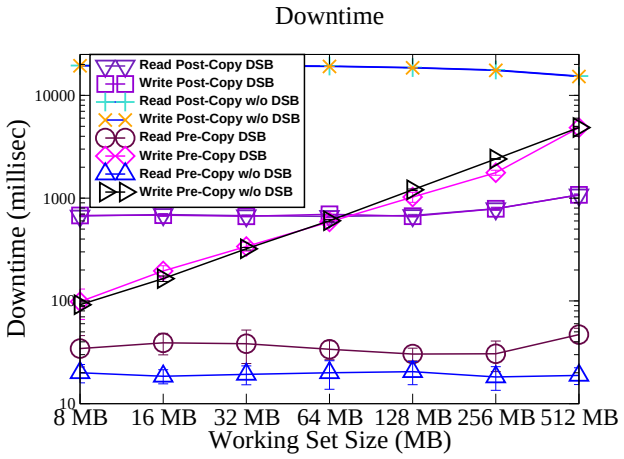

Figure 6: Downtime comparison. (Log scale y-axis).

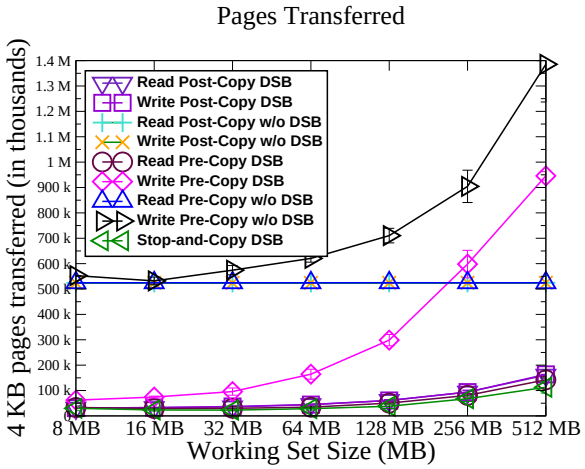

Figure 7: Comparison of the number of pages transferred during a single migration.

Working Set Size	Prepaging		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. prepaging.

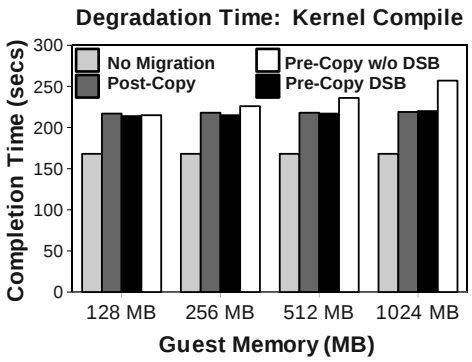

Figure 8: Degradation time with kernel compile.

Pages Transferred and Page Faults: Figure 7 and Table 2 illustrate the utility of our prepaging algorithm in post-copy across increasingly large working set sizes. Figure 7 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 14 compares the fraction of network and minor faults in post-copy. We see that prepaging reduces the fraction of network faults from 7% to 13%. To be fair, the stress-test is highly sequential in nature and consequently, prepaging predicts this behavior almost perfectly. The real applications in the next section 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 8 depicts a repeat of an interesting experiment from [19]. 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 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 8 just by executing back to back invocations of

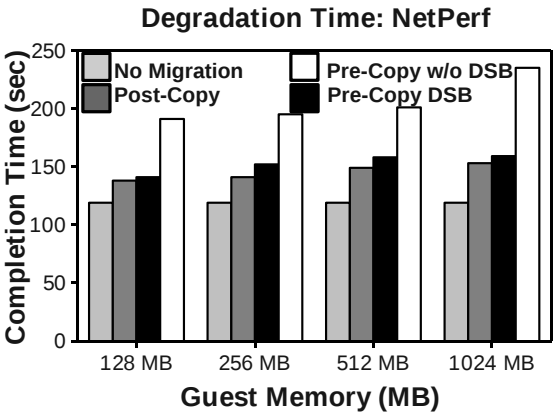

Figure 9: NetPerf run with back-to-back migrations.

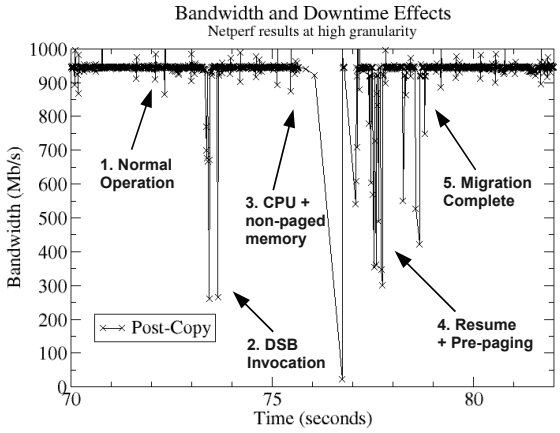

Figure 10: Post-copy impact on NetPerf bandwidth.

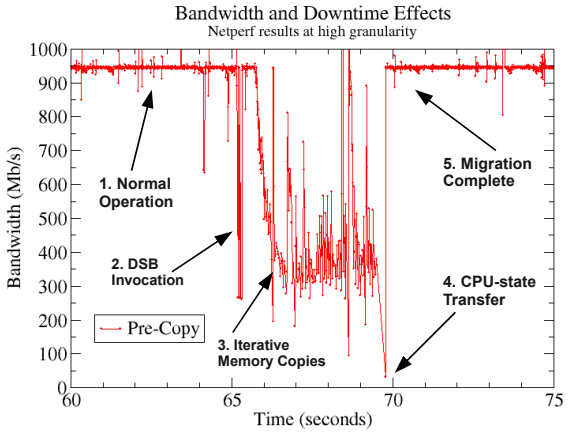

Figure 11: Pre-copy impact on NetPerf bandwidth.

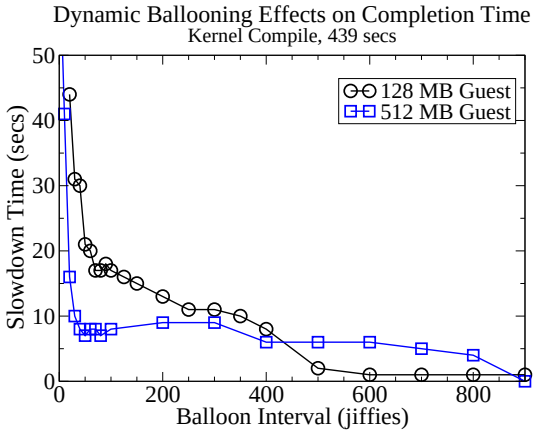

Figure 12: Application degradation is inversely proportional to the ballooning interval.

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 5 and 7. 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 9 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 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 the original 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 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 techniques.

With that in mind, Figures 10 and 11 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 pages transferred in pre-copy is much higher. This is exactly the bottom line that we were targeting for improvement.

Dynamic Ballooning Interval: Figure 12 shows how we chose the DSB interval, by which the DSB process wakes up to reclaim 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

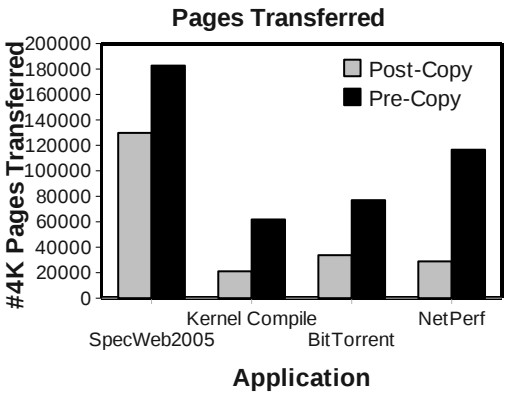

Figure 13: Total pages transferred.

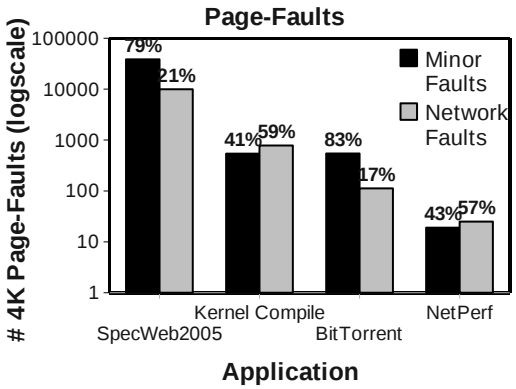

Figure 14: Network faults versus minor faults.

base case is the *degradation time* added to the application by the DSB process due to its CPU usage. As expected, the ballooning interval is inversely proportional to the application degradation. The more often you balloon, the more it affects the VM workload. The graph shows that an interval between 4 and 10 seconds is good enough to frequently reclaim free pages without impacting application performance.

5.3 Application Scenarios

We re-visit the aforementioned performance metrics across four applications: (a) **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 VM, while six separate client nodes bombard the VM with connections; (b) **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. (c) **Linux Kernel Compile**: We consider this application again for consistency. (d) **NetPerf**: Once more, as in the previous experiments, the NetPerf sender is placed inside the 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 VM is configured to have two virtual CPUs. All these experiments have DSB enabled.

Pages Transferred and Page Faults.. The experiments in Figures 13 and 14 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 14 where post-copy's prepaging 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.

Total Time and Downtime. Figure 15 shows that post-

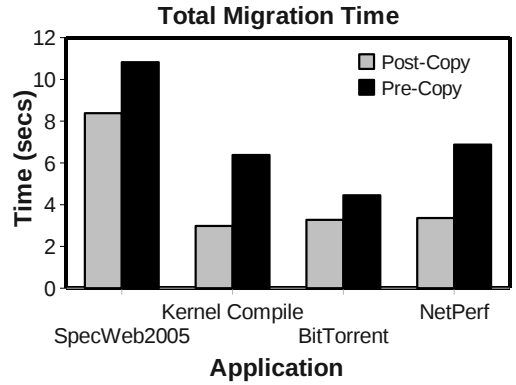

Figure 15: Total migration time.

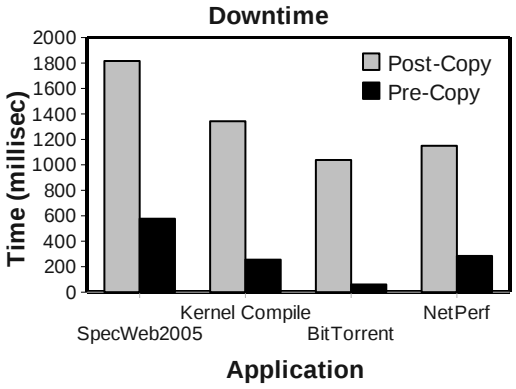

Figure 16: Downtime comparison: Post-copy downtime can improve with better page-fault detection.

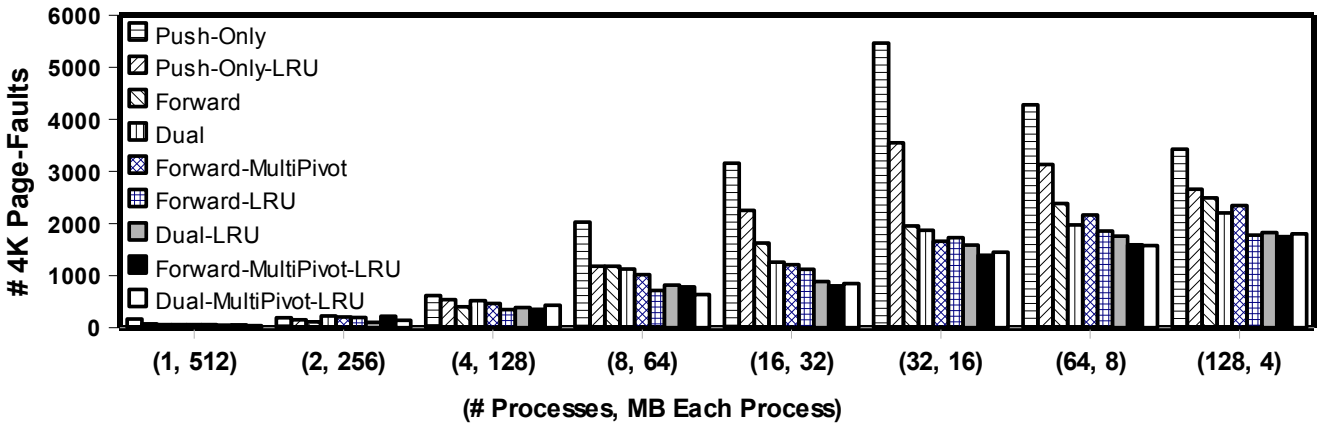

Figure 17: Comparison of prepaging strategies using multi-process Quicksort workloads.

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 16 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 where network overhead needs to be kept low and the entire migration needs to be completed quickly.

5.4 Comparison of Prepaging Strategies

This section compares the effectiveness of different prepaging strategies. The VM workload is a Quicksort application that sorts a randomly populated array of user-defined size. We vary the number of processes running Quicksort from 1 to 128, such that 512MB of memory is collectively used among all processes. We migrate the VM in the middle of its workload execution and measure the number of network faults during migration. A smaller the network fault count indicates better prepaging performance. We compare a number of prepaging combinations by varying the following factors: (a) whether or not some form of bubbling is used; (b) whether the bubbling occurs in forward-only or dual directions; (c) whether single or multiple pivots are used; and (d) whether the page-cache is maintained in LRU order.

Figure 17 shows the results. Each vertical bar represents an average over 20 experimental runs. First observation is that bubbling, in any form, performs better than push-only prepaging. Secondly, sorting the page-cache in LRU order performs better than non-LRU cases by improving the locality of reference of neighboring pages in the pseudo-paging device. Thirdly, dual directional bubbling improves performance over forward-only bubbling in most cases, but never performs significantly worse. This indicates that it is always preferable to use dual directional bubbling. (The performance of reverse-only bubbling was found to be much worse than even push-only prepaging, hence its results are omitted). Finally, dual multi-pivot bubbling is found to consistently improve the performance over single-pivot bubbling since it exploits locality of reference at multiple locations in the pseudo-paging device.

6. CONCLUSIONS

This paper 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, prepaging, and dynamic self-ballooning. We implemented and evaluated post-copy on Xen and Linux based platform. Our evaluations show that post-copy significantly reduces the total migration time and the number of pages transferred compared to pre-copy. Further, the bubbling algorithm for prepaging is able to significantly reduce the number network faults incurred during post-copy migration. Finally, dynamic self-ballooning improves the performance of both pre-copy and post-copy by eliminating the transmission of free pages during migration. In future work, we plan to investigate an alternative to pseudo-paging, namely shadow paging based page fault detection. We are also investigating techniques to handle target 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-0845832, CNS-0509131, and CCF-0541096.

8. 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] Hines, M., and Gopalan, K. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Washington, DC* (March 2009).

[12] Ho, R. S., Wang, C.-L., and Lau, F. C. Lightweight process migration and memory prefetching in OpenMosix. In *Proc. of IPDPS* (2008).

[13] Kerrighed. *http://www.kerrighed.org*.

[14] Kivity, A., Kamay, Y., and Laor, D. KVM: the linux virtual machine monitor. In *Proc. of Ottawa Linux Symposium* (2007).

[15] 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.

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

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

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

[19] Nelson, M., Lim, B.-H., and Hutchins, G. Fast transparent migration for virtual machines. In *Usenix, Anaheim, CA* (2005), pp. 25–25.

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

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

[22] Oppenheimer, G., and Weizer, N. Resource management for a medium scale time-sharing operating system. *Commun. ACM 11*, 5 (1968), 313–322.

[23] 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.

[24] 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).

[25] Richmond, M., and Hitchens, M. A new process migration algorithm. *SIGOPS Oper. Syst. Rev. 31*, 1 (1997), 31–42.

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

[27] Sapuntzakis, C., Chandra, R., and Pfaff, B. Optimizing the migration of virtual computers. In *Proc. of OSDI* (2002).

[28] Satyanarayanan, M., and Gilbert, B. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing 11*, 2 (2007), 16–25.

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

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

[31] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the Condor experience. *Concurr. Comput. : Pract. Exper. 17* (2005), 323–356.

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

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

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

[35] Waldspurger, C. Memory resource management in VMWare ESX server. In *Operating System Design and Implementation (OSDI 02), Boston, MA* (Dec 2002).

[36] Whitaker, A., Cox, R., and Shaw, M. Constructing services with interposable virtual hardware. In *NSDI 2004* (2004), pp. 13–13.

[37] 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.