MiG: Efficient Migration of Desktop VMs using Semantic Compression

Anshul Rai[†], Ramachandran Ramjee[†], Ashok Anand[‡,*],
Venkata N. Padmanabhan[†], and George Varghese[§]
[†]Microsoft Research India [‡]Bell Labs India [§]Microsoft Research US

ABSTRACT

We consider the problem of efficiently migrating desktop virtual machines. The key challenge is to migrate the desktop VM quickly and in a bandwidth-efficient manner. The idea of replaying computation to reconstruct state seems appealing. However, our detailed analysis shows that the match between the source memory and the memory reconstructed via replay at the destination is poor, even at the sub-page level; the ability to reconstruct memory state is stymied because modern OSes use address space layout randomization (ASLR) to improve security, and page prefetching to improve performance.

Despite these challenges, we show that desktop VM memory state can be efficiently compressed for transfer without relying on replay, using a suite of *semantic* techniques – collectively dubbed as MiG – that are tailored to the type of each memory page. Our evaluation on Windows and Linux desktop VMs shows that MiG is able to compress the VM state effectively, requiring on average 51-65% fewer bytes to be transferred during migration compared to standard compression, and halving the migration time in a typical setting.

1. INTRODUCTION

Efficient migration of desktop virtual machines (VM) is important in a variety of scenarios. First, consider the vision of a desktop PC environment that is always available and local to the user [15, 16, 25, 27]. In these systems, the user's desktop environment is encapsulated in a VM, so that it can be moved flexibly between, say, the user's office workstation, home PC, and laptop, providing a *seamless computing experience, without sacrificing interactive responsiveness of local execution*. Second, consider the desktop as a service model where desktop VMs execute in the cloud and are accessible from any local device. A key requirement in this scenario is ensuring low response times [24]. This necessitates migrating the VM over WAN links so that the VM executes in a data center that is always close to the user. Finally, desktop VM migration has also been utilized for saving energy [20]. In these systems, when the user is not engaged in computing, the VM is migrated to a server in the cloud so that the local machine can go to sleep and save energy.

A key challenge common to the above scenarios is efficient migration of VMs, both in terms of migration time and the amount of data transferred, especially over links of modest bandwidth. For instance, transferring a 4 GB VM over a 10 Mbps connection would take nearly an hour, which can be frustrating for a user who wants to transfer the VM from workplace or cloud to her home for better interactivity. Further, many ISPs worldwide offer tiered service plans with bandwidth caps ranging from 1GB to 250GB per month, with higher cost for higher limits [4]; apart from *transfer time*, a home user would also care equally about the *amount of bytes* transferred.

In this paper, we consider the problem of efficiently migrating desktop VMs. We start by revisiting the idea of replaying input to speed up migration [28] and show its limitations in practice. We then present *MiG*, which categorizes memory pages based on type (e.g., free page, code (i.e., image) page, heap page, etc.) and then employs a page-type-specific technique to perform effective compression. This paper only considers migration of memory state; while migration of disk state could be important in certain settings, measurements presented in prior work show that the amount of dirty disk state to be migrated is an order or magnitude smaller than the VM's memory size (e.g., [20] reports dirtying of disk blocks at an uncompressed rate of 40-100 MB per hour).

Input replay has been proposed as a technique to speed up desktop VM migration [28], by trading computation for byte savings. By replaying user input (e.g., keyboard/mouse events), the "same" computation is performed on the destination machine. The hope is to recreate much of the source's memory state on the destination, thereby reducing the state to be transferred during VM migration. Our study reveals that mechanisms employed by modern OSes pose many practical difficulties in benefiting from input replay.

First, for improving interactive performance, modern desktop OSes prefetch pages into memory based on user actions, application behavior, etc. (e.g., SuperFetch [9] in Windows and preload in Linux). Thus, a long running workload might result in a certain set of pages prefetched into memory while the same workload, replayed in an accelerated fashion for fast migration, might result in a different set of prefetched pages at the replayed VM.

[*]The author was an intern at MSR India during part of this work.

Second, for security reasons, modern OSes (e.g., Windows Vista/7 and recent versions of Linux) employ Address Space Layout Randomization (ASLR), wherein the layout of code segments is randomized, which in turn impacts the values of the embedded pointers. Therefore, the "same" pages, or even sub-pages, at the source and the destination will not match with input replay.

Third, managed code runtimes (e.g., the .NET Common Language Runtime) actively manage memory using mechanisms such as garbage collection. The invocation of these mechanisms during replay on the destination machine will typically *not* match that on the source machine, resulting in poor matches for heap pages. We find that even matching at the level of heap allocation units yields little benefit.

Our first contribution in this paper is an evaluation of the impact of the above mechanisms through extensive measurements of VMs running multiple flavours of Windows and Linux, spanning the evolution in the prevalence of ASLR and page prefetching. Our findings go beyond a recent study of memory similarity in VMs [13] by showing that even identical VMs with identical input can have dramatic differences in memory, even at the sub-page level. For example, while zero pages account for 72% of the pages in a Windows XP VM that is left running for several hours, the corresponding figure for the newer Windows 7 OS is only 4%, because of SuperFetch implemented by the latter. Likewise, the fraction of non-zero pages that match across two freshly booted VMs goes from 66% in the case of Windows XP to 33% with Windows 7, on account of ASLR. We also see a corresponding, though less pronounced, trend with Linux.

Despite the above findings, we show that we do not have to turn off prefetching and ASLR (which could have undesirable performance and security implications) for efficient VM migration. *We present MiG, our second contribution, which leverages observations from our measurement study to tailor compression to the semantics of memory pages, thereby obtaining significant gains in the context of VM migration.* The page-semantics-dependent techniques including identifying and suppressing free pages, eliminating significant intra-VM redundancy in heap pages and compressing image and SuperFetch pages using a novel approach that uses file system data as a primer dictionary for a dictionary-based redundancy elimination [12]. Our experiments bear out the effectiveness of MiG, which yields average byte savings of 51% and 65% over a `gzip`-compressed VM image, for Windows and Linux desktop VMs, respectively. These byte savings translate into a significant speedup in migration time; e.g., for a 2GB Windows 7 VM being migrated over a 10Mbps link, MiG halves the migration time (including computing overhead) to 275s from 558s with `gzip`-only compression.

Our third contribution is a reality check on the gains achievable through replay. To this end, we develop MiG-Replay, which uses MiG as the starting point but additionally exploits full page and heap matches with respect to the memory state of a replayed VM. We find that MiG-Replay can provide 15% gains over MiG but only in specific cases where either the workload is short or the pace of replay is identical to the original; for long workloads and where replay is accelerated in time to be practical, MiG-Replay even underperforms MiG, because of ASLR and SuperFetch.

Type	WinXP	Win7	Debian 6d	Debian 6
Blank VM	85%	66%	89%	80%
Short workload	72%	47%	68%	56%
Long Workload	72%	4%	63%	55%

Table 1: **Zero pages in Win XP, Win 7, Debian 6 with preload/ASLR disabled (Debian 6d) and Debian 6**

Type	WinXP	Win7	Debian 6d	Debian 6
Blank VM	66%	33%	76%	61%
Short+Paced	42%	34%	66%	48%
Long+Accelerated	41%	14%	62%	43%

Table 2: **Identical non-zero pages in OSes with replay**

2. MEMORY SIMILARITY

A high degree of full and partial page similarity were reported [23] in Windows XP and older Linux VMs (Debian 3.1/Slackware 10.2). A recent study [13] of memory similarity among VMs shows that page similarity has reduced to 15%. However, these studies [13, 23] were in the context of a server hosting disparate VMs. In this section, we characterize memory similarity between *two identical VMs provided with identical input*. In particular, we seek to answer the following questions:

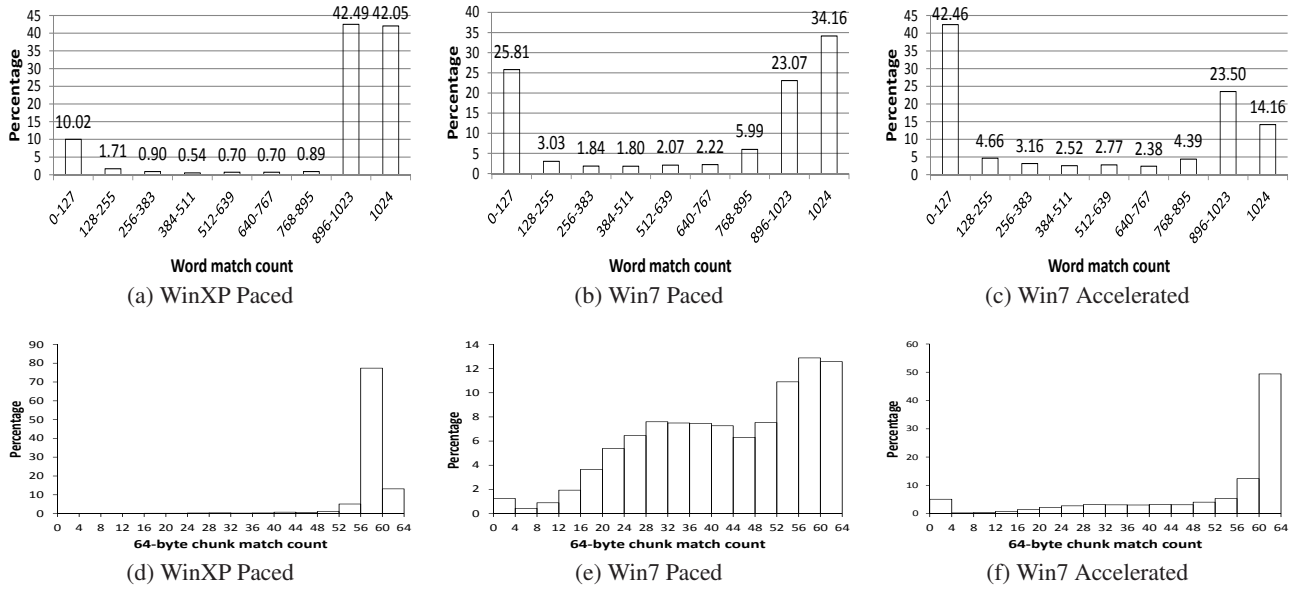
- How similar are two VMs at the full page level? At the sub-page level?
- How effective are existing techniques, like `rsync` [29], in leveraging inter-VM similarity?
- How do these similarities vary for different page types (e.g., Heap, Image, etc.)?
- How much redundancy exists intra-VM? How effective are existing compression techniques (`gzip`, `bzip2`, `7zip`) on intra-VM redundancy?

Understanding these issues is crucial for designing an efficient migration scheme. We now briefly describe the workload and the replay techniques used in our experiments before presenting the results of our analysis.

2.1 VM Workloads and Input Replay

Workload. Our workload consists of VMs running various versions of Windows and Linux for three cases: i) Freshly booted blank VM, ii) short workload of running applications for 30 minutes and iii) long workload of running applications over several hours (workload is typical desktop office applications, detailed in Section 5).

Replay. Replay on VMs can be accomplished in a number of ways. Instruction-level replay with strict adherence to timing as in the ReVirt system [21] will ensure that the destination VM is identical to the source VM in all respects. However, accomplishing instruction-level replay on a multiprocessor system has large overheads [22]. In this paper, we



(a) WinXP Paced (b) Win7 Paced (c) Win7 Accelerated

(d) WinXP Paced (e) Win7 Paced (f) Win7 Accelerated

Figure 1: **Distribution of sub-page matches**

consider input replay, which involves simply replaying the user inputs to the system (e.g., keyboard and mouse events), detailed in Section 5). While input replay cannot guarantee that the destination VM memory is identical to the source VM due to non-determinism and network interactions, the hope is to recreate similar memory so that creating an identical version is efficient. In this section, we evaluate the *similarity* of VMs that are created using this input replay mechanism.

We consider 3 scenarios: i) *Blank VM:* two freshly booted VMs, ii) *short workload, paced replay:* two VMs with identical apps executing for 30 minutes with paced input replay (same keyboard and mouse events paced identically at both VMs), iii) *long workload, accelerated replay:* two VMs with identical apps/replay but one is a long running VM where input is spread over a period of several hours representing typical usage, while, in the other VM the input is accelerated in time (e.g., ten minutes), representing a practical scenario of using replay for fast migration.

2.2 Page-level Similarity

We start with Table 1 that lists the percentage of zero pages in VMs running various OSes that have each been allocated 2 GB of memory. We see that the fraction of zero pages in Windows XP starts at 85% and reduces to 72%; *in the case of Windows 7, the fraction of zero pages starts at 66% but goes down to 4% for the long workload case*.

The dramatic reduction in zero pages in Windows 7 is due to a new feature that was first introduced in Windows Vista called SuperFetch [9]. SuperFetch is a user-customized prefetching technique that tracks application usage and selectively preloads applications or data into memory in order to improve interactive responsiveness. Linux has a similar feature called *preload* available in Debian 6 that preloads pages to improve performance. While it does not appear to be as aggressive as SuperFetch, it also reduces the number of zero pages. For a blank VM, Debian with preload disabled had 89% zero pages which reduces to 80% with preload enabled, and for a long workload, the corresponding numbers are 63% and 55%, respectively.

Next, in Table 2, we consider the number of identical non-zero pages when replay is used. Consider the case of freshly booted Windows XP and Windows 7 VMs. While 66% of the non-zero pages are identical in two XP VMs, only 33% are identical in Windows 7. Next consider paced replay which represents an ideal scenario for recreating similar memory; the percentage of identical non-zero pages in Windows XP reduces to 42%, while, for Windows 7 it is 34%.

However, in the more practical accelerated replay scenario, we notice an interesting divergence. While the numbers for Windows XP do not change significantly, we notice *a drastic reduction in the percentage of identical non-zero pages in Windows 7 to 14%*. This reduction is due to a combination of SuperFetch (acceleration of input has significant impact on SuperFetch's pre-fetching) and ASLR, that we will discuss in the next sub-section.

In the case of Linux, we verified that Debian 3.1 used in Difference Engine [23] does not have ASLR while Debian 6 has a weaker form of ASLR with less randomization, resulting in higher non-zero page matches than Windows 7. For the long workload, accelerated replay scenario, Debian 6 with ASLR and preload disabled had 62% non-zero identical page matches which reduced to 43% when ASLR and preload was enabled.

Two observations follow from these results:

- **O1:** Fraction of zero pages is dramatically reduced to 4% in Windows 7 and significantly reduced to 55% in Debian 6 compared to 70+% in Windows XP due to prefetching.

- **O2:** Fraction of identical non-zero pages between two Windows 7 VMs running identical applications with identical input ranges between 14-33% compared to 41-66% for Windows XP and 43-61% for Debian 6.

2.3 Sub-page-level Similarity

We now investigate partial-page similarity between two 2GB identical VMs provided with same input. A brute-force way to identify a page in the second VM most similar to a page in the source VM would entail 2GB*2GB or 10^{18} comparisons! Instead, we adapt Min-wise hashing [14] and compute hashes of each 4-byte word using 16 hash functions. For each hash function, we store the minimum hash value of all words in the page as a 16-tuple that succinctly represents the page. For each page in the source VM, we find the page in the second VM with the largest number of matching hash values in the 16-tuple. Tuple similarity is an unbiased estimator [14] of page similarity, a fact we verified by brute-force calculation on a small sample of source VM. After finding the most similar page in the destination VM, we compute a similarity measure as the number of corresponding words that match between the two similar pages

The distribution of word match count between non-zero pages of two identical Windows XP and Windows 7 VMs are shown in Figures 1a and 1b, respectively. Note that a page is 4KB in size and thus there are 1024 4-byte words in a page. Consider Windows XP with paced replay (accelerated replay is similar). We find that, in 42% of pages, the word-level match count is 1024, i.e., these are identical pages. We also find that, for another 42.5% of the pages, there is a very high degree of similarity (896-1023 word matches). Now consider Windows 7 with paced replay. While 34% of pages are identical, only 23% of pages have a high degree of similarity.

The presence of highly similar pages is not sufficient for reducing the amount of bytes transferred; the word differences in these similar pages must also be clustered to have long sequences of continuous word matches, which can be efficiently removed. To examine this issue, we segment each 4 KB page into sixty four 64-byte chunks and study the distribution of differences between the highly similar pages. Figure 1d shows that the differences are indeed clustered in the case of Windows XP (56 or more out of 64 chunks match in over 90% of the cases) while Figures 1e and 1f shows that the differences are spread throughout the page in the case of Windows 7, resulting in far fewer chunk-level matches.

The difference between Windows XP and Windows 7 is due to Address Space Layout Randomization (ASLR) [2], a security feature where the start addresses of executables, the heap, etc. are placed at random locations to make it difficult for an attacker to guess. The randomization, performed at the granularity of 64 KB chunks, can result in pointer references in code/heap pages being different in executions in two VMs. This results in differences between similar Windows 7 pages being spread throughout the page. For Linux, a minimal version of ASLR was enabled only in 2.6.12 while both Linux versions studied in [23] used older kernels. Thus, while the authors in [23] found a high degree of partial page matches ($>$ 2KB) across VMs, our findings corroborate the diminished page sharing found in [13].

Finally, Figure 1c shows the distribution of word match count for the Windows 7 VM with accelerated replay. The fraction of pages that have very low match (0-127) has increased to 42.5% for the Windows 7 VM with accelerated replay, up from 26% in the case of Windows 7 VM with paced replay and just 10% in the case of Windows XP. SuperFetch is the primary reason for this increase in the prevalence of low matches (as elaborated further in Section 2.5), since SuperFetch customized to the first VM is unlikely to make matching decisions regarding prefetching in the second VM, where the input replay is accelerated in time.

Summarizing sub-page-level similarity results:

- **O3:** Even among pages that are highly similar (896-1023 word matches), the locations of differences in the page are *not* clustered in Windows 7 due to ASLR. Thus, partial page sharing opportunities, as identified in [23], are significantly diminished.
- **O4:** Fraction of pages with little match is significant (42%) in Windows 7, primarily, due to SuperFetch.

2.4 Chunk-level Matches using rsync

While pages are a natural way of segmenting physical memory, an alternative is finer-grained chunk-level matching between two VMs. In this section, we consider synchronizing two VM memory dumps using `rsync`[29], a file synchronization application that leverages a similar, remote version of the file for compression. `rsync` computes sliding window chunk hashes over the remote version (replayed VM in our case) and uses these hashes to identify and compress identical chunks in the local version (current VM) for efficient migration.

We perform a parameter sweep, in steps of 32 bytes, to determine the optimal chunk-size for `rsync` that maximizes compression for the VM dumps. Using this optimal chunk size (128 bytes), `rsync` yields compression savings of 69.7% and 40.4%, respectively, for the Windows 7 with paced and accelerated replay.. These savings correspond roughly to the sum of the last three bars in Figures 1b and 1c, respectively. Applying `gzip` in addition to `rsync` yields a total savings of 72.5% with accelerated replay.

In the context of VM migration, the on-the-wire traffic goes from 100-66.5 = 33.5% with gzip compression alone to 100-72.5 = 27.5% of the VM size with rsync (plus gzip). Thus, `rsync`, which relies on replay, provides only a modest 18% relative byte savings over gzip. Furthermore, it takes 840s, 10X slower than gzip.

- **O5:** Applying a fine-grained chunk-matching technique like `rsync` on two VMs with identical applications and identical replay, only yields about an 18% reduction over conventional gzip compression.

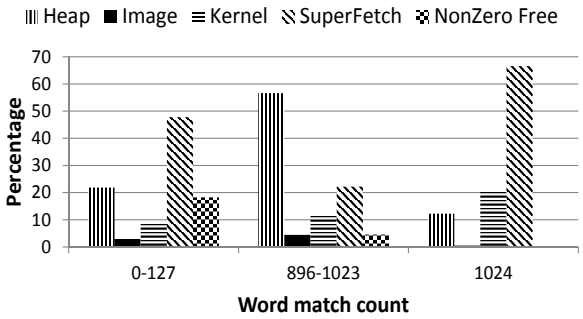


Figure 2: **Page Type Distribution**

2.5 Semantic Analysis

To gain a deeper understanding, we now parse the page similarity results by page type. We consider the accelerated replay case. We classify the pages into five categories: heap, image (i.e., code), kernel, SuperFetch, and free. Figure 2 shows the relative distribution of different page types, corresponding to a few cases in Figure 1c, namely, pages with very low matches (0-127), highly similar pages (896-1023), and identical pages (1024).

For pages with very low match (0-127), SuperFetch pages are dominant, (50%); this is caused due to the difference in the prefetching decisions in the long-running VM and the replayed VM where input is accelerated in time. In contrast, for pages with high similarity (896-1023), heap is dominant (over 50%) while SuperFetch is second (over 20%). ASLR converts what might have been identical pages in Windows XP into pages that are highly similar in Windows 7. Finally, for identical page matches, SuperFetch constitutes over 60%, followed by kernel pages at 20%.

Full VM	Heap	Image	Kernel	SFetch	Free
66.5%	80.0%	69.2%	67.6%	47.3%	81.4%

Table 3: **Savings by page type using gzip**

Intra-VM redundancy: We applied `gzip` on the entire VM and also on different collection of pages collated by their type (Table 3). While the entire VM can be reduced by 66.5% using `gzip`, we see that compression savings vary significantly across page types. Heap and free pages can be reduced by 80% (due to a predominance of zero bytes; e.g. 66% of bytes in heap pages were zeros compared to 45% for the entire VM), while SuperFetch pages can be reduced by only 47%.

gzip vs bzip2 vs 7zip: We also examined other well-known compression utilities such as `bzip2` and `7zip` that have been shown to be better than gzip in other contexts [18]. However, these utilities were all *significantly* slower than `gzip`, significantly inflating overall VM migration time, another metric of interest in our setting. For example, on a 2.2 GHz CPU core, `gzip` takes 65s when optimized for speed (with compression savings of 66.5%) and 117s when optimized for compression (savings increases to 68.5%). In contrast, using default settings, `bzip2` [5] takes 390s to reduce the VM by 68.4% and `7zip` [1] takes 810s to reduce the VM by 77.5%.

Summarizing, semantic analysis of memory pages helps inform our design of efficient migration:

- **O6:** Free pages can be compressed by almost 100% since these need not be transferred.
- **O7:** Heap pages are highly compressible using gzip.
- **O8:** SuperFetch pages constitute a significant fraction of low-match pages and are not highly compressible using gzip. Hence, we need an efficient technique for transferring these pages. Many SuperFetch pages are also image pages; a technique that works for these SuperFetch pages will also work for image pages.
- **O9:** Full page matches can benefit kernel and SuperFetch pages.

3. MIG DESIGN

We now present the design of MiG, our solution for efficient migration of desktop VMs, which does *not* resort to input replay. As a point of comparison, we also design MiG-Replay, which leverages a replayed VM's memory state where appropriate.

Design Constraints: In order to ensure correct operation post migration, we need to ensure that source memory is replicated fully and identically at destination. Even the goal of input replay is only to create similar memory at the destination so that creating an identical version is efficient, since as mentioned earlier, due to non-determinism and network interactions, input replay cannot guarantee a semantically identical VM.

The only exception to the above is that Free pages need not be identical since the OS does not rely on its contents. Note that even SuperFetch pages have to be identical at the two ends. This is because the SuperFetch service may "activate" these pages at any time based on its internal representation (e.g, SuperFetch page 1 is image page for process x), without the knowledge of the hypervisor.

Another constraint we impose is that the design should not require changes to the guest OS. For example, requiring that the guest OS implement a mechanism to get/set the ASLR random seed via the hypervisor is out of scope. This is to ensure that the migration solution will work for existing versions of guest OSes that are deployed today. Note that this constraint does not preclude the design from using any publicly documented information of the guest OSes for its operation, since this does not affect its deployability.

3.1 Overview

At a high level, MiG and MiG-Replay operate as follows. When a desktop machine is to be migrated from a source machine *S* to a destination machine *D*, we examine each memory page on *S* and apply techniques tailored to the type of the page. MiG relies only on local state at *S*, including disk state that has been synced previously, MiG-Replay, in addition, also leverages the memory state of the VM at *D* that has been constructed via input replay. The set of techniques applied derives from the observations **O6** through **O9**:

- **Free/Zero pages**: In both MiG and MiG-Replay, these pages are identified on *S* and not transferred.
- **Full-page matches:** In MiG, memory image of a freshly booted VM is pre-provisioned at both *S* and *D*. Full-page matches with respect to this "blank VM" helps reduce the bytes transferred. In MiG-Replay, full-page matches are computed against the replayed VM at *D* instead of a blank VM.
- **Image/SuperFetch pages:** For image pages, whether active or prefetched, both MiG and MiG-Replay employ a novel approach involving statically precomputing a common, primer dictionary at both *S* and *D*. This dictionary comprises the contents of commonly-accessed executable and library files, and is used as a reference for computing a diff of the memory state.
- **Heap pages:** In MiG, we employ a combination of history-based redundancy elimination [12] and gzip to identify and eliminate redundancy within heap pages. In MiG-Replay, where possible, we parse the heap to identify the chunks that match between *S* and *D*.
- **Other:** For both MiG and MiG-Replay, the remaining pages are compressed using a combination of dictionary-based redundancy elimination [12] and gzip.

Next, we discuss each technique in greater detail.

3.2 Free/Zero Pages

MiG and MiG-Replay identify free/zero pages by parsing the page allocation table on *S* (Section 4) and only convey their indices to *D*, thereby achieving nearly 100% compression for these pages. Since free pages can have non-zero content, conventional compression schemes achieve less savings on these pages (e.g., only 81% savings when gzip is applied on free pages – Table 3).

3.3 Full-page Matches

As seen in Figure 2, kernel pages constitute a good percentage of full-page matches, in large part because ASLR is typically not applied to kernel pages. Thus, MiG preprovisions the memory state of a freshly booted "blank VM" and the corresponding page hashes at both *S* and *D*. At transfer time, MiG simply computes a fast 4-byte hash [6] for each page at *S*, matches it against the hash list of the blank VM, verifies using a byte-by-byte comparison with the local copy (to neutralize hash collision risk), and sends across the index and location of the matched page to *D*, which then reads in the corresponding page from its local copy to reconstruct the memory state.

In MiG-Replay, we look for full-page matches between *S* and the replayed VM, *D*. A 4-byte hash is computed for each page at *S* and these are sent across to *D* as a list of *(page index, hash)* pairs. *D* then compares these hashes from *S* with those computed locally on its own pages. When a hash from *S* matches one at *D*, the corresponding page need not be transferred from *S* to *D*. To reduce hash collision risk in MiG-Replay, we also send a full 20-byte SHA1 hash [11] of just the matched pages.

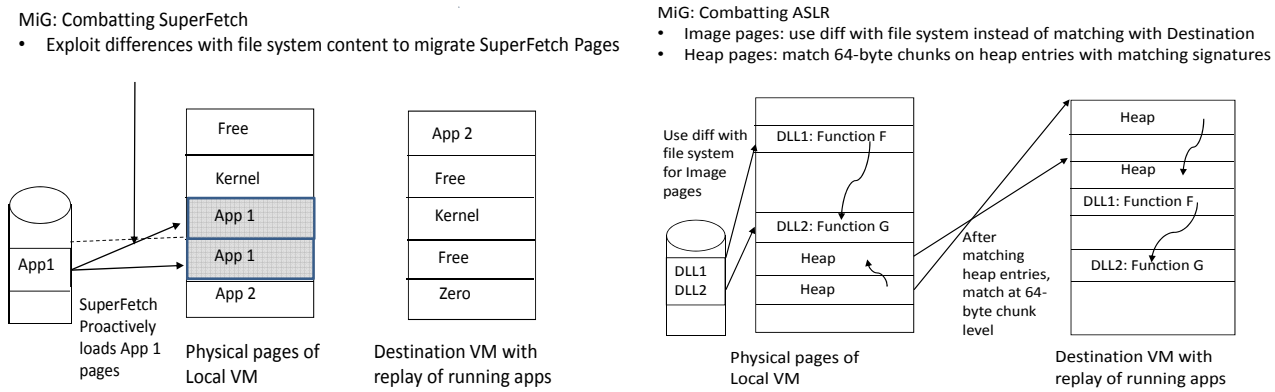
3.4 Image/SuperFetch Pages

Image pages comprise active pages that are in the address space of a process as well as SuperFetch pages that are prefetched in anticipation of future use. ASLR impacts both active and SuperFetch pages by impeding even sub-page level matching. Indeed, as reported in Section 2, even if the page were divided into 64-byte chunks, 40-75% of the pages have 8 or more chunks that do *not* match (Figures 1e and 1f). The fine-grained nature of the matches, interspersed with non-matching pointers, means that two pages would ideally need to be compared side-by-side, defeating the goal of efficient transfer.

3.4.1 Using Precomputed File System Context

For the reasons noted above, neither MiG nor MiG-Replay relies on *D* for compressing image pages. Instead, as shown in Figure 3, they employ a novel approach that builds on the observation that the image pages in memory are derived from the file system content. Indeed, the OS loader reads binaries from the file system and places these in memory, albeit with modifications because of ASLR. The content of these binary files provides context both similar to the target pages (image/SuperFetch) and also locally available at both *S* and *D*. Thus, the pre-computed context built using file system content offers the prospect of good compression savings without any overhead incurred in establishing the shared context.

To realize *pre-computed context based compression*, we prime the dictionary in an existing redundancy elimination algorithm, EndRE [12]. EndRE works as follows. Given a cache/dictionary of past packets that have been transferred from a source to a destination, EndRE identifies contiguous strings of bytes in the current packet that are also present in the cache. This is accomplished by 1) identifying a set of representative "fingerprints" for each packet 2) looking up these fingerprints in a "fingerprints store" that holds the fingerprints of all the past packets in the cache' and 3) for each fingerprint of the packet that is found in the store, the matching packet is retrieved and the matching region is expanded byte-by-byte in both directions to obtain the maximal region of redundant bytes. Once all matches are identified, the matched segments are replaced with fixed-size pointers into the cache, thereby suppressing redundancy. In the original EndRE, the cache starts empty and is dynamically built up as packets are transferred between source and destination. In the primed version, the cache at both ends is primed with 2 GB worth of file system content, comprising commonly-used binary and library files. The priming is done by passing these files to EndRE which builds its internal data structures for identifying redundancy. Subsequently, when the SuperFetch/image pages are passed to EndRE, contiguous byte strings that are redundant with the bytes in the primed context are identified and replaced with pointers, which are then restored at the destination using its primed cache.



(a) SuperFetch (shaded pages) impacts the ability of replay to recreate matching pages at the destination VM; MiG/MiG-Replay exploit differences with (local) file system to migrate these pages

(b) Due to ASLR, function and data pointers in image and heap pages may not match across two VMs; MiG relies on intra-VM redundancy for heap while MiG-Replay matches 64-byte heap chunks across VMs

Figure 3: **Illustration depicting how MiG and MiG-Replay combat SuperFetch and ASLR**

While we have not performed any optimization or customization of the context, we believe user-specific personalization could yield both reduction in context size as well as potentially higher compression savings.

3.5 Heap Pages

Now we turn to heap pages. Since MiG does not rely on replay, it exploits intra-VM redundancy for heap pages. We already saw that gzip was able to compress heap pages by 80% due to predominance of zero-bytes. Further, examining the heap pages of a VM instance, we found that out of the total heap of 670MB, about 170MB (non-zero bytes) was redundant. Even though over 91% of this redundancy was from within the heaps belonging to the same process, the vast majority of the redundancy was between byte strings located in different memory pages. So compression techniques such as gzip, which look for redundancy over a small window (64KB), will often not be able to identify such redundancy across disparate locations. Hence, to compress heap pages, MiG uses an intra-VM redundancy technique based on EndRE [12], that identifies redundancy over a large history (e.g., 2GB), coupled with gzip.

MiG-Replay, on the other hand, has the advantage of access to the replayed VM to eke out additional gains over MiG. Conceptually, replay should create a heap at D similar or identical to the one at S. However, in practice there is a distinction between heap *content* and heap *structure*. Replay could, in fact, help make the heap content similar. Yet, the structure of the heap could be very different across the two ends because of garbage collection and compaction, which kick in asynchronously.

Therefore, to match the process heaps across S and D, MiG-Replay looks deeper. Heaps in Windows 7 come in two forms: *managed heap*, whose structure can be parsed, and *unmanaged heap* which are private to a process. For managed heap, MiG-Replay performs a heap walk on the heap of each process at S and D, to produce the list of heap entries at each end. For each heap entry at S, MiG-Replay computes a hash of its used portion (parts of the heap entry might be unused), and sends it across to D. D looks through its heap entries for hash matches. If a matching heap entry is found, the corresponding content need not be transferred from S to D.

As shown in Figure 3b, ASLR can again result in differing pointers inside the heap entries of S and D, that make an exact match of the full heap entry less likely. To mitigate this effect of ASLR, we chunk the heap entry into n-byte blocks and compute 4-byte hashes to identify potential matches. Based on our evaluation (Section 5), we find that blocks of size $n = 64$ bytes provided us with the highest compression savings. Finally, for unmanaged heaps, since the heap structure cannot be parsed, MiG-Replay simply chunks them into 64-byte chunks at S and looks for chunk matches at D on corresponding heap pages that belong to the same process.

To keep the processing overhead manageable, we only apply the above procedure for heap entries that are larger than 1KB in allocated size. Our measurements show that heap entries that qualify as being "large" per the above criterion account for an overwhelming 80+% of the bytes in the heap.

We stress again that all of the above complexity associated with replay and parsing the heap is only for the case of MiG-Replay, which we designed solely for the purpose of comparison with the much simpler MiG scheme.

3.6 Other Pages

The pages that remain include stack pages, and also kernel pages that did not benefit from a full-page match. For such pages, both with MiG and MiG-Replay, we employ intra-VM redundancy elimination using EndRE [12].

4. IMPLEMENTATION

We now briefly discuss the implementation of MiG on Windows. MiG runs in the root partition of the Microsoft Windows Server 2008 Hyper-V system [8]. While our current prototype is targeted towards the quick migration feature of Hyper-V (suspend-migrate-resume), wherein the VM

state is saved, moved, and restored at the destination, we believe MiG can be effectively applied to the VM live migration [17] scenario as well.

To initiate migration, we use existing Hyper-V mechanisms to save the state of the VM, which yields a VM physical memory file and two small (few MB) configuration files containing the VM configuration information and the saved state of devices. MiG reads in the saved physical memory and extracts semantic information in two steps. First, it consults a system-wide data structure, the Page Frame Number (PFN) database, which has metadata information for each page (zero or free, allocated to a process or kernel, etc.), and allows the reverse mapping of a physical memory address to the virtual address of a process, where applicable. Second, MiG consults the Virtual Address Descriptor (VAD) tree process-specific data structure to determine the type of the page (`MEM_PRIVATE` for heap, `MEM_IMAGE` for code and `MEM_MAPPED` for memory mapped file pages) in the virtual address space. MiG then applies the appropriate technique from Section 3 to each of the pages and, thus, creates a compressed version of the memory file, which is migrated to the destination. Note that both PFN and VAD are publicly documented, so while MiG is intrusive in having to look into the memory state of the VM, it does not depend on access to any proprietary information.

In case of MiG-Replay, our prototype takes in two saved memory image files — one each corresponding to the original and the replayed VM — and simply performs an analysis of the compression gains of having a replayed VM. In addition to extracting the above semantic information, MiG-Replay also performs a heap walk by parsing the heap structure of each process on the two VMs, to identify the allocated heap chunks for heap compression.

For Linux, we use the libVMI tool [7] for introspection into VM memory.

5. EVALUATION

Metrics. We evaluate the performance of MiG, MiG-Replay and `rsync` primarily in terms of volume of bytes transferred relative to using gzip as the compression scheme.[1] Let gzip yield a total byte transfer requirement of b_{gzip}. For any other scheme x (e.g., MiG), let the byte requirement be b_x. The byte savings, or compression *gains*, of x over the baseline is then $g_x = \frac{(b_{gzip} - b_x)}{b_{gzip}} \times 100$. This relative savings metric captures the bytes saved compared to the scheme, namely `gzip`, that is commonly used in commercial systems such as Windows Server 2008 Hyper-V. Further, if compression processing is faster than the link speed, this relative byte savings would translate into an equivalent reduction in migration time. Thus, we also evaluate the *migration transfer time* for the various schemes. Finally, we do not present *absolute byte savings* as a separate metric since it is already captured as part of the migration transfer time metric. In general, absolute byte savings for MiG ranges between 80-95% for the various scenarios.

Workloads. We evaluate MiG performance for both Windows and Linux OSes. For Windows, we collect memory dumps from 10 desktops with real user workloads, running the *32-bit* version of Windows 7 with *2-4 GB* of RAM, 8 of which were from desktops used by researchers and 2 by admin staff. For Linux, we use *64-bit* VMs running Debian squeeze (2.6.32.5-amd64) and workload consists of a mix of document editing (openoffice word/ presentation, gedit), image manipulation (gimp, inkscape, photo manager), and web-browsing (firefox, epiphany) applications, reflecting common desktop usage.

In order to evaluate MiG-Replay, we use Windows VMs with artificially generated workloads that emulate a Windows desktop computing environment, with applications such as Outlook (email), Internet Explorer (browser), Word (document editor), Excel (spreadsheet), etc. running. We use the AutoIt scripting language [3] to automate the Windows GUI and design scripts to feed keyboard input into Word or Excel interspersed with random think-time, sync email, download pages from different websites, etc. We perform 5 runs of this emulation, with different combinations of applications used in each instance, with each experiment lasting between 30 minutes and four hours to mimic a user work session. For each of these experiments, we also performed paced and accelerated replay (same script without think-times).

5.1 MiG Byte Savings

Figures 4 and 5 show the percentage byte savings achieved by MiG relative to gzip for each of the individual Windows and Linux desktop VMs, respectively. First, we see that *MiG delivers consistent byte savings of 40%-60% for the ten Windows VMs (average 51%) and 58%-68% for the five Linux VMs (average 65%) over gzip*.

It is interesting to observe the contribution of the different MiG techniques towards achieving the overall gains. For Linux VMs, the bulk of the gains come from the use of precomputed context (30-38%), followed by Full page matches (18-25%) and intra-VM redundancy elimination (6-13%). In contrast, for Windows VMs, the majority of the gains come from Intra-VM redundancy elimination (35-44%), followed by precomputed context (3-15%), Full page matches (2-7%), and Free pages (0-7%).

The surprising finding in the above results is that while the use of precomputed context (Section 3.4) provides substantial benefits for Linux VMs (30-38%), its contribution to savings in Windows VMs is modest (3-15%). Upon examining this in more detail, we find that while precomputed context in Windows had indeed full or partial matches with over 80% of image pages and 45% of SuperFetch pages[2], many of these matches were also captured by intra-VM redundancy elimination. These intra-VM redundant matches were

[1] We do not use `rsync or 7zip` as a baseline to compare against since these are an order of magnitude slower than gzip at the settings that provide savings.

[2] The lower cache hit rate for SuperFetch pages is because SuperFetch pages can also be non-image pages.

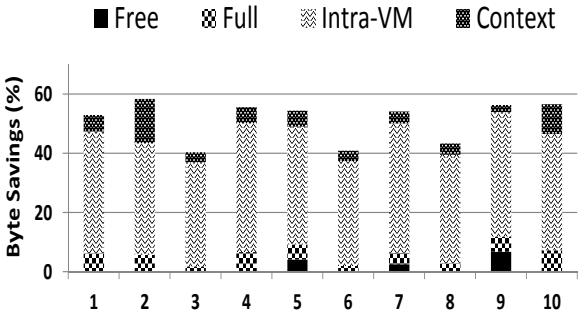


Figure 4: **MiG byte savings on Windows VMs**

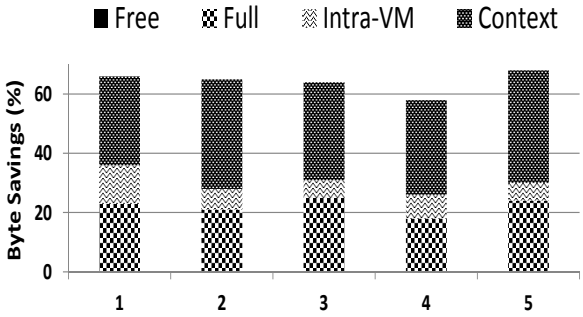


Figure 5: **MiG byte savings on Linux VMs**

between pages belonging to the same DLL that had been loaded by different processes (e.g., user32.dll was loaded by almost 50 out of the 100 processes in one Windows desktop), though, vast majority of these matches were for only small portions of a page (average match length of only 107 and 83 bytes for image and SuperFetch pages, respectively). Thus, in Windows, a substantial portion of the savings that would have accrued from using a precomputed context is already obtained by intra-VM redundancy elimination.

The other notable difference is the higher contribution of Full page matches in Linux (18-25%) versus Windows (2-7%). This is explained by the fact that Windows has much more extensive ASLR support turned on by default than Linux and agrees with the higher full page sharing numbers for Linux (Section 2).

In summary, MiG delivers significant byte savings of 51% and 65% over gzip for Windows and Linux desktop VMs, respectively. The different techniques in MiG each contribute towards achieving these savings, though, the significance of each technique's contribution varies between Windows and Linux.

5.2 Replay

Figure 6 depicts the byte savings relative to gzip for the non-semantic scheme `rsync`, and the two semantic schemes (MiG and MiG-replay) for four Windows desktop VM workloads, viz., different combinations of short/long workloads and paced/accelerated replays. In this case, short/long workloads lasted 30 mins/four hours of automated use of office applications and while paced replay took the same time as the original workload, accelerated replay took under ten minutes to complete for both workloads.

From the figure, we see that MiG provides average savings relative to gzip of 38-48% for all these workloads without relying on any replay. Using the replayed VM memory, we find that `rsync` provides about 18-34% relative savings while MiG-Replay provides 39-63% relative savings. Note that `rsync` gains over gzip are modest when the replay is accelerated, indicating that the memory image created with accelerated replay is not as close to the source image as in paced replay.

Interestingly, MiG-Replay delivers about 15% additional savings compared to MiG in cases where either the replay is paced or the workload is short; however, when the workload is long and replay is accelerated in time (as would be necessary for fast migration), we *find that MiG-Replay surprisingly performs worse than MiG by 8%*. The reason is twofold. First, the gains due to matching of SuperFetch pages disappear because accelerated replay fails to evoke the same prefetching pattern as the actual execution of the VM. Second, the degree of similarity in the heap also diminishes, so the overhead of performing heap matching (e.g., sending hash values across from *S* to *D*) overwhelms the gains obtained from the actual matches. We examine this second reason next.

Type	Excel	Outlook	Powerpoint	OneNote
Managed pages	131	212	205	347
Unmanaged pages	893	1045	1249	860
Bytes (%) (heap size $> 1KB$)	84.5	84.9	83.8	89.8
Bytes % match	84	80.6	77.4	73.3
MiG-Replay savings %	50	44	36	42

Table 4: **Heap characteristics of some office apps**

Table 4 shows some important statistics for the heap pages of a few Office applications. The heap comprises of *managed* and *unmanaged* heap and we can see that managed heap is only 13-29% of total heap for these applications. The ability of replay to recreate the source's memory state at the destination is highlighted in the next row that lists the percentage of bytes that match heap entries between source and destination VMs. For these applications, we see that between 73 to 84% of bytes do indeed match between source and destination VM. However, since the match is not exact (because of pointers affected by ASLR), we resort to dividing the heap into chunks and perform matching at the smaller granularity of chunks rather than matching at larger full heap entries. We evaluated compression savings for the entire managed heap size using the replay mechanism for different chunk sizes (not shown) and found that *64-byte chunks provide the highest savings of 40-50%*, balancing the overhead of sending 4-byte hashes for each of the chunks and the cost of losing a chunk match due to a small difference (e.g., a pointer value change) between source and destination chunks. For unmanaged heap, we use the same chunk size to divide up the heap pages and try to identify matches at the replayed VM; again, the 4-byte hash overhead for each 64-byte chunk results in decreasing the savings. Additional protection against hash collisions will further reduce these savings.

To summarize, while replay does indeed create similar

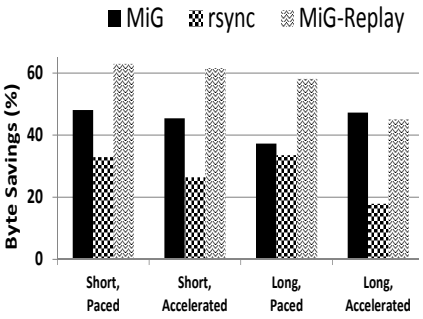


Figure 6: **Replay workloads**

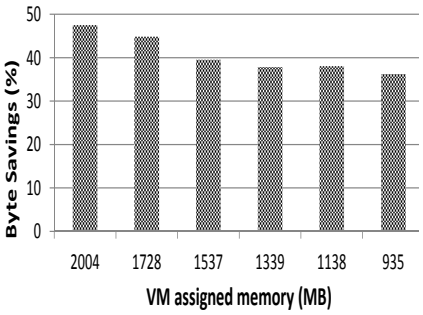


Figure 7: **Ballooning**

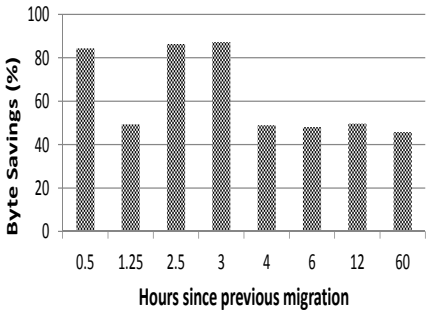


Figure 8: **Repeated migrations**

state at the destination, MiG-Replay's ability to compress heap pages effectively using replay is impeded by the reality that (a) only a small fraction (13-29%) of the heap is parseable, (b) the reconstruction of the heap using replay is far from perfect (only 73-84% of bytes match), and (c) we have to resort to matching small, 64-byte chunks to get maximum compression savings, incurring high overhead and limiting savings. Since MiG is able to perform effective intra-VM redundancy elimination for heap pages, MiG-Replay is unable to gain over MiG.

5.3 Ballooning

Ballooning is a technique to artificially introduce memory pressure in a VM, leading it to evict less important pages [30]. One application of ballooning proposed in the literature is in the context of efficient migration, wherein unnecessary pages are shed from the VM memory prior to migration [26].

In general, it is hard to estimate the amount of memory to be ballooned out; overdoing it can cause the eviction of important pages and thus adversely impact user-perceived performance. Thus, *migrating the entire memory is desirable*. However, since memory ballooning can be applied independently of MiG, in this section, we investigate the impact of memory ballooning on MiG byte savings by using Hyper-V dynamic memory feature to create different amounts of memory pressure on the VM before applying MiG.

Figure 7 shows MiG's relative savings over gzip for different amounts of assigned memory for a 2GB VM, corresponding to a memory reduction of 5-55% through ballooning. While the savings decreases as the assigned memory is reduced, MiG is still able to deliver 36% relative savings even with 55% of VM memory pages evicted. This is because while ballooning evicts low priority pages like free pages, it does not evict all SuperFetch or heap pages[3] and, thus, a substantial portion of the MiG byte savings remains.

Further, at high memory pressure, many of the evicted pages are paged out into a *pagefile (swap)* in disk which, of course, also needs to be migrated. MiG can be directly applied to the pages sitting in the pagefile just as to the pages in memory. MiG's savings on pagefile was similar to the savings achieved for in-memory pages.

[3]SuperFetch pages retain the priority of the original page.

5.4 Repeated Migrations

One of the scenarios targeted by MiG is the desktop that is always-on and is migrated repeatedly between work and home or work and cloud. In this section, we evaluate the benefit of using the memory from previously migrated state for byte savings. In these cases, both source and destination save the previously transferred VM memory and MiG uses this for its full page matches instead of a blank VM as before (all other techniques remain the same).

We had a user use a 2GB Windows 7 VM for several days; applications used included the browser and several office applications. Every once in a while, sometimes after short intervals of 30 minutes to few hours, and sometimes after long intervals of several hours to even days, we took snapshots of the VM memory, representing a checkpoint of VM state that needs to be migrated. We then used MiG with the benefit of the previous snapshot for computing byte savings.

Figure 8 depicts MiG's relative byte savings over gzip for the different migration cases corresponding to workloads that last from 30 minutes to 60 hours. From the figure, we see that short workloads of up to a couple of hours in general significantly benefit from using the previous snapshot by delivering relative savings over gzip of up to 87% (corresponding to absolute reduction of VM size by 95%). Of course, not all short workloads result in such gains (for example the 1.25 hour data point), due to windows update or other system activity that can potentially induce large change in memory, effectively reducing the effectiveness of the previous snapshot. Finally, we see that for workloads beyond a few hours, the previous snapshot is not as useful and MiG's relative gains drop down to about 50%.

5.5 Migration Time

We now evaluate the time for VM migration for the various schemes. Let us first consider the computational cost of the different MiG techniques on a 2.2 GHz CPU core for a typical 2GB Windows 7 VM.[4] In the following analysis, MiG's context cache is preloaded in memory and the VM image is also in memory.

Parsing the PFN database to extract semantic informa-

[4]Most of these numbers scale proportionately with VM size but can vary depending on the amount of compression achieved.

	Compressed Size	Compute time	Transfer time
gzip	670MB	65s	558s
MiG	330MB	67s	275s

Table 5: **Migration time for 2GB VM on 10Mbps link**

	Off	Default	Aggressive
Full	26%	23%	22%
Full+Intra-VM	32%	32%	30%
Full+Intra-VM+context	62%	69%	70%

Table 6: **Impact of preload on MiG savings (Linux)**

tion takes about 20s.[5]. MiG also creates a 4-byte hash of each page using Jenkins Hash [6] and compares these hashes against the local precomputed hashes of a blank VM for identifying full page matches. This process takes around 8s. The processing of SuperFetch and image pages using EndRE [12] with primed context takes about 15s. Finally, all remaining pages are compressed using EndRE+gzip, which takes about 24s. Thus, our MiG prototype implementation is able to reduce a 2GB VM to about 330 MB in 67s. For the same VM, gzip reduces it to 670 MB in 65s.

Migration time (Table 5) is determined by the maximum of transfer time and compression processing time. Transfer time is directly proportional to (compressed) VM size and inversely proportional to link speed. Thus, on slow links and/or for large VMs, MiG's migration is significantly faster than gzip. For example, *on a 10 Mbps link, MiG transfers the 2GB VM in about 275s, halving the transfer time of gzip (558s)*. For comparison, it takes 810s for `7zip`, 840s for `rsync` and 1665s for uncompressed transfer.

We can also take advantage of multi-core CPUs to perform many of the above operations in parallel to optimize MiG (and gzip). Using 4 cores, processing for an optimized version of MiG can easily be limited to less than 28s (and optimized gzip to less than 55s), thereby allowing *MiG to retain the 50% reduction in both bytes and migration time compared to gzip at 100 Mbps speeds*. Of course, on 1 Gbps or faster links, transferring the raw VM may be faster than using either gzip or MiG.

6. DISCUSSION

Turning off page prefetching prior to migration: Turning off page prefetching mechanisms such as SuperFetch could aid VM migration by cutting out the prefetched pages from the set that needs to be moved. However, doing so can have an adverse impact on user-perceived performance [10]. Nevertheless, it is interesting to ask how much there is to be gained from varying the amount of prefetching, in terms of the byte savings achieved by MiG.

To answer this question, we used a Linux desktop VM loaded with a few applications and tested it under three different settings for prefetching: off (preload removed), default, and aggressive (increased free memory to be used for prefetching from default of 50% to 90%). The byte savings relative to gzip is shown in Table 6. While increasing the degree of prefetching results in a reduction in the number of full-page matches relative to a blank VM, this loss is more than offset by leveraging pre-computed context to compress the prefetched (and other) pages, resulting in similar absolute byte savings for all these cases. This suggests that we do not have to turn off prefetching to obtain byte savings for migration.

Influencing randomization in ASLR: While turning off ASLR would adversely impact security, one could arguably influence ASLR's randomization policy more subtly, to make it more migration friendly while not compromising security. For instance, when a VM is migrated, the randomization seed used for ASLR at the destination could be set to be the same as that at the source, which might then make the memory state of a replayed instance match more closely with the source. However, to our knowledge, for security reasons, OSes do not make the seed available through an API or document the location of the seed in memory so that, for instance, it could be read from the hypervisor.

Nevertheless, to get a sense for the gains to be had if the same seed were used at the source and the destination, consider the evaluation presented in Section 5.4. Since a single VM instance was snapshoted repeatedly, the randomization seed remained unchanged across the snapshots. When the interval between two snapshots is short (under 2 hours), there is a high degree of match between the snapshots. However, when the interval is longer, the snapshots tend to diverge, even though the randomization seed is the same across the snapshots. The divergence is because of SuperFetch, garbage collection, etc., factors which exist independent of ASLR. Thus, any short-term gains arising from maintaining the same ASLR seed get overshadowed over longer durations by other sources of non-determinism.

7. RELATED WORK

ISR, Collective, Transient PCs. The vision of a desktop PC environment that is mobile and available anywhere was articulated by Chen and Noble [16] and in the Internet Suspend Resume (ISR) project [25]. The Collective [15] is another system that provides users with a consistent desktop environment at a computer nearby as users move. Recently, this paradigm of having the desktop environment stored in the cloud but executed on a PC close to the user has been dubbed the transient PC [27].

Efficient Migration. Prior work closest to MiG is the work on optimizing the migration of virtual computers [26]. Their system uses copy-on-write disks in order to migrate disk changes, supplanted with demand paging to fetch needed blocks, memory ballooning to zero out unused memory, and page hashing to suppress identical memory blocks. While MiG can benefit from the disk migration techniques in [26], migrating memory state is much more challenging today due to new OS features. The idea of using replay in order to migrate VMs efficiently was proposed in [28]. However, as we

[5]This is primarily due to our prototype using windbg APIs which makes disk accesses; an optimized version should take under 10s.

show in this paper, replay provides only small gains.

CloudNet [31] supports efficient Live WAN migration of VMs. It implements smart stop and copy to reduce the number of iterations/copies for Live migration which can be useful for live migration support in MiG. It also implements redundancy elimination by computing sub page-level hashes (1 KB in size) and comparing this to previously sent data. MiG's intra-VM redundancy elimination eliminates redundant chunks that are as small as 32 bytes. Remus [19] is a system that replicates VMs asynchronously. Remus uses page compression including delta and gzip compression for efficient checkpointing. Since Remus checkpoints state every 25*ms*, memory page delta-based approach works well for them. For durations comprising several hours, typical for VM migration, we find that previous memory state is not useful.

Similarity in VM memory. Looking beyond migration, the recent study by Barker et al. [13] reports that page sharing in Linux and Windows VMs running at a server is diminished because of ASLR. Our study differs from and goes beyond this prior work in several ways. First, since our goal is efficient migration, we compare two VMs running the same OSes/applications and provided with the same input, which is not a scenario considered in [13]. Second, while [13] focuses mostly on page-level sharing, we show that even at 64-byte chunk level, changes due to ASLR render sharing ineffective. Third, going beyond ASLR, we also evaluate and show the significant impact of OS prefetching (e.g., SuperFetch) on memory redundancy.

8. CONCLUSION

When we started our investigation into efficient migration of desktop VMs, we had assumed that replay and memory similarity would lead to efficiency. However, we were puzzled by the lack of similarity even in blank VMs. The culprit, as explained in this paper, is randomness and nondeterminism due to mechanisms such as ASLR and page prefetching in modern OSes. Through extensive experiments on both Windows and Linux, we have characterized and quantified the impact of these mechanisms.

Despite these hurdles, our migration solution, MiG, yields compression gains of 51% and 65% over gzip on Windows and Linux VMs, respectively, and halving of the overall migration time. Central to MiG is the idea of tailoring the compression technique to the semantics of memory pages, an approach which we believe could transcend the specific OS mechanisms and compression techniques considered here.

9. ACKNOWLEDGEMENTS

We thank our shepherd, Jason Flinn, and the anonymous reviewers for their constructive comments.

10. REFERENCES

[1] 7zip. http://www.7-zip.org.
[2] ASLR. http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx.
[3] AutoIT. http://www.autoitscript.com/site/.
[4] Bandwidth caps around the world. http://www.maximumpc.com/article/features/how_bad_do_we_really_have_it_bandwidth_caps_around_world.
[5] bzip2. http://www.bzip.org.
[6] Jenkins Hash. http://burtleburtle.net/bob/c/lookup3.c.
[7] LibVMI tool. http://code.google.com/p/vmitools/.
[8] Microsoft Hyper-V. http://www.microsoft.com/en-us/server-cloud/windows-server/hyper-vaspx.
[9] SuperFetch. http://msdn.microsoft.com/en-us/library/bb188739.aspx.
[10] SuperFetch performance. http://everythingexpress.wordpress.com/2011/11/13/how-to-adjusting-windows-7-superfetch/.
[11] US Secure Hash Algorithm 1 (SHA1). RFC 3174, September 2001.
[12] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *USENIX NSDI*, April 2010.
[13] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman. An Empirical Study of Memory Sharing in Virtual Machines. In *USENIX ATC*, June 2012.
[14] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of IEEE Compression and Complexity of Sequences*, June 1997.
[15] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The Collective: A Cache-Based System Management Architecture. In *NSDI*, May 2005.
[16] P. Chen and B. D. Noble. When virtual is better than real. In *8th IEEE Workshop on Hot Topics on Operating Systems*, May 2001.
[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, May 2005.
[18] L. Collin. A quick benchmark: Gzip vs. Bzip2 vs. LZMA, 2005. http://tukaani.org/lzma/benchmarks.html.
[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *USENIX NSDI*, April 2008.
[20] T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. Shin. LiteGreen: Saving Energy in Networked Desktops using Virtualization. In *USENIX ATC*, June 2010.
[21] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.
[22] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.
[23] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines . In *OSDI*, December 2008.
[24] A. Kochut and H. Shaikh. Desktop to cloud transformation planning. In *IEEE IPDPS*, May 2009.
[25] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *IEEE WMCSA*, June 2002.
[26] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *OSDI*, 2002.
[27] M. Satyanarayanan, S. Smaldone, B. Gilbert, J. Harkes1, and L. Iftode. Bringing the Cloud Down to Earth: Transient PCs Everywhere. In *MobiCloud 2010, Santa Clara, CA*, October 2010.
[28] A. Surie, H. A. Lagar-Cavilla, E. de Lara, and M. Satyanarayanan. Low-Bandwidth VM Migration via Opportunistic Replay. In *HotMobile*, February 2008.
[29] A. Tridgell. Efficient Algorithms for Sorting and Synchronization, 2000. PhD thesis, Australian National University.
[30] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, December 2002.
[31] T. Wood, K. Ramakrishnan, P. Shenoy, and J. V. der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Virtual Execution Environments (VEE)*, March 2011.